CICS® Transaction Server for VSE/ESA™

REXX Guide

Release 1

IBM

CICS® Transaction Server for VSE/ESA™

REXX Guide

Release 1

Second edition (September 2000)

Contents

Preface

What this book is about

This book describes REXX/CICS or REXX for CICS Transaction Server for VSE/ESA. This IBM program product provides a native REXX-based application development, customization, prototyping, and procedures language environment for REXX/CICS, along with associated runtime facilities.

Who this book is for

This book is for users who need to refer to CICS Transaction Server for VSE/ESA REXX instructions and functions, and for those who need to learn more details about REXX language items such as parsing. It is also intended for anyone who wants to learn how to write REXX programs. The type of users include: application programmers, system programmers, end users, administrators, developers, testers, and support personnel.

What you need to know to understand this book

Within this book, reference may be made to *Release 1*. **The function described in this book has been added since the release of CICS Transaction Server for VSE/ESA Release 1 and can be identified within the code as 1.1.1.**

If you are not an experienced programmer, and are new to REXX, you should read the guide part of this book, see page 1.

The book contains two parts: a user's guide and a reference. The user's guide section will help you become familiar with REXX for CICS Transaction Server for VSE/ESA. The reference section contains the CICS Transaction Server for VSE/ESA REXX instructions, functions, and commands. The instructions, functions, and commands are listed alphabetically in their own sections. Also included are details about general concepts you need to know in order to program in REXX.

The programming language described by this book is called the REstructured eXtended eXecutor language (commonly referred to as REXX). This book also describes how the CICS Transaction Server for VSE/ESA REXX language processor (shortened, hereafter, to the language processor) processes or *interprets* the REstructured eXtended eXecutor language.

Prerequisites

REXX/CICS runs under CICS Transaction Server for VSE/ESA Version 1 or above. There are no other prerequisites (other than the prerequisites that CICS TS for VSE/ESA requires).

ix

Part 1. User's Guide

Chapter 1. Introduction

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

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 under CICS. These include issuing EXEC CICS commands, SQL statements, as well as commands to the CEDA (Resource Definition Online Transaction) and CEMT (Master Terminal Transaction) utilities.

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 of 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 section 2.3, "Syntax of REXX Instructions" on page 8.

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

Debugging capabilities: When a REXX program running in REXX/CICS encounters an error, REXX writes messages describing the error. You can also use the REXX TRACE instruction and the interactive debug facility to locate errors in programs.

Interpreted language: The REXX/CICS product includes the REXX/CICS 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.

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.

5

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 (REXX/CICS and CICS commands and SQL).

  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/CICS 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.

Chapter 2. Writing and Running a REXX Program

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.

2.1 What you need to run a REXX Program?

Before you can run a REXX Program, you must configure the REXX support. Go through the following steps:

1. Create the RFS Filepools
2. Install Resource Definitions
3. Update LSRPOOL Definitions
4. Rename supplied Procedures
5. Update CICSTART.PROC
6. Update CICS Initialization JCL
7. Format the RFS Filepools
8. Create the Help Files
9. Verify the Installation
10. Configure the REXX DB2 Interface

For details on configuration, see Appendix K, "Post-Installation Configuration" on page 491.

2.2 What is a REXX Program?

A REXX program consists of REXX language instructions that the REXX interpreter interprets directly. A program can also contain commands that the host environment executes, such as CICS commands (see page 95).

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 of output, you use the REXX instruction SAY followed by the text you want written.

```
/* Sample REXX Program                                              */
  SAY 'Hello world!'
```

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 on page 13.

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

```
Hello world!
```

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: From a CICS terminal you would clear the screen and enter:

```
REXX addtwo
```

For this example, the first number you will enter is 42, and the second number is 21. Here is the ADDTWO program:

```
/*************************** REXX *******************************/
/*  This program adds two numbers and produces their sum.       */
/****************************************************************/
say 'Enter first number.'
 PULL number1                      /* Assigns: number1=42 */
say 'Enter second number.'
 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, 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 displays the output line:

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

Before you try any examples, please read the next two sections, 2.3, "Syntax of REXX Instructions" and 2.5, "Typing in a Program" on page 14.

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 terminal output device:

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

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.

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 only significant during parsing, see section 7.2, "Parsing Data" on page 86.

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 terminal output device.

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 section 4.2, "Using Conditional Instructions" on page 39.

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 section 4.4.1, "EXIT Instruction" on page 56.

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 section 3.3, "Using Expressions" on page 25.

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" on page 67 and section 4.4.3, "SIGNAL Instruction" on page 57.

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: REXX/CICS does not require that a REXX program begin with a comment. However, for portability reasons, you may want to start each REXX program with a comment that includes the word REXX. Not every language processor requires this program identifier. However, to run the same exec on MVS TSO and CICS, you should include the /* REXX */ program identifier to satisfy TSO requirements.

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 */
ADDRESS REXXCICS 'DIR'
```

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 REXXCICS. The command is the expression that follows the environment:

```
'DIR'
```

The DIR command lists the files in your current file system directory. For more details about changing the host command environment, see section 8.2.3, "Changing the Host Command Environment" on page 97.

More information about issuing commands appears in Chapter 8, "Using Commands from a Program" on page 95.

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 example.

Example

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
```

2.5 Typing in a Program

When you type in the following program, use the REXX/CICS editor for files that reside in the REXX File System (RFS).

The name of the program is HELLO EXEC (for now, assume that the file type is exec).

1. Sign on to a CICS Transaction Server for VSE/ESA terminal by entering `CESN` and supplying your user ID and password when it is requested.

2. Clear the screen.

3. Type:

   `edit hello.exec`

4. Type in the program, exactly as it is shown in Figure 4, beginning with `/* REXX HELLO EXEC */`. Then file it using the EDIT command:

   `====>  file`

Now your program is ready to run.

```
/* REXX HELLO EXEC     */

/* A conversation      */
say "Hello! What is your name?"
pull who
if who = "" then say "Hello stranger!"
else say "Hello" who
```

Figure 4. HELLO EXEC

2.5.1 Running a Program

Clear the screen before running an exec. If you want to run a program that has a file type of EXEC, you type in REXX followed by its file name. In this case, type `rexx hello` on the command line and press Enter. Try it!

Suppose your name is Sam. Type `sam` and press Enter. `Hello SAM` is displayed.

```
rexx hello
Hello! What is your name?
sam
Hello SAM
```

Here is what happens:

1. The SAY instruction displays `Hello! What is your name?`

2. The PULL instruction pauses the program, waiting for a reply.

3. You type `sam` on the command line and then press Enter.

4. The PULL instruction puts the word SAM into the variable (the place in the computer's storage) called `who`.

5. The IF instruction asks, Is `who` equal to nothing?

   ```
   who = ""
   ```

   This means, "is the value stored in `who` equal to nothing?" To find out, REXX substitutes that stored value for the variable name. So the question now is: Is SAM equal to nothing?

   ```
   "SAM" = ""
   ```

6. Not true. The instruction after `then` is not processed. Instead, REXX processes the instruction after `else`.

7. The SAY instruction displays `"Hello" who`, which is evaluated as

   ```
   Hello SAM
   ```

Now, here is what happens if you press Enter without typing a response first.

```
hello
Hello! What is your name?

Hello stranger!
```

Then again, maybe you did not understand that you had to type in your name. (Perhaps the program should make your part clearer.) Anyhow, if you just press Enter instead of typing a name:

1. The PULL instruction puts "" (nothing) into the place in the computer's storage called `who`.

2. Again, the IF instruction tests the variable

   `who = ""`

   meaning: Is the value of `who` equal to nothing? When the value of `who` is substituted, this scans as:

   `"" = ""`

   And this time, it is true.

3. So the instruction after `then` is processed, and the instruction after `else` is not.

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 5. Example of a Program with a Syntax Error

When the program runs, the language processor sends the following lines of output.

```
     7 +++ PULL who                    /* Get the person's name.IF who =
'' THEN SAY 'Hello, stranger'ELSE   SAY 'Hello,' who
CICREX453E Error 6 running HELLO EXEC, 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 terminal 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.

```
CICREX453E Error 6 running HELLO EXEC, line 7: Unmatched "/*" or quote
```

For more information about the error, you can go to the message explanations in Appendix A, "Error Numbers and Messages" on page 429.

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. */
```

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 terminal and sends this information to the terminal output device.

```
/************************** REXX ***********************************/
/* This REXX program gets the name of an animal from the input    */
/* stream and sends it to the terminal.                           */
/******************************************************************/

PULL animal                      /* Get the animal name.*/
SAY animal
```

Figure 6. 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`

2.7.1.1 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.

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 program stack or terminal input device.

- By specifying input when calling the program.

2.8.1 Getting Information from the Program Stack or Terminal Input Device

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:

`REXX addtwo`

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 7. Example of a Program That Uses PULL

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

```
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 8. Variation of an Example that Uses PULL

The PULL instruction extracts the numbers 42 and 21 from the terminal.

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 CICS command:

```
REXX add 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 9. 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:

```
REXX add 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:

```
REXX add 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`.

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 10. Example of a Program That Uses PARSE ARG

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

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.

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.

Chapter 3. Using Variables and Expressions

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

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**.

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"       */
```

3.2.1.1 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

23

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>
```

3.2.1.2 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
```

3.2.1.3 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. Incorrect, because the first character is a "$."

3. Valid

4. Valid

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

3.3 Using Expressions

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

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 section section 13.12, "NUMERIC" on page 176. 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 scientific notation (floating point numbers), see section 16.2.7, "Exponential Notation" on page 261.

-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:

Table 1. 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
   ```

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

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

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.

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 */
```

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 section 4.2.1, "IF...THEN...ELSE Instructions" on page 39.

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

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

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 */
```

I	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 */
```

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.

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
```

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.

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 section 7.2, "Parsing Data" on page 86.

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.

Table 2. 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

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 on page 32, if the parentheses were removed from the student's formula, what would be the outcome for the college?

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

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

ANSWERS

1. The results are as follows:

   a. 34   (22 + 12 = 34)

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

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

2. I'll consider it.

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

```
IF  inexpensive | scholarship & reputable | nearby  THEN
       _____/   _____/   _____/   \____/
          true         false        true      false
            \            _____/            /
             \              false              /
              _____/               /
                     true                     /
                       _____/
                               true
```
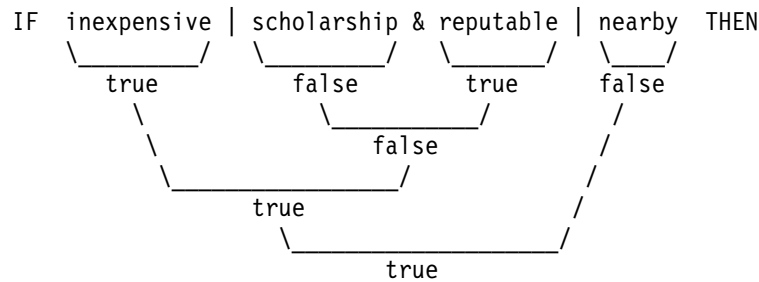
3.4 Tracing Expressions with the TRACE Instruction

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 15. 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, see section 13.23, "TRACE" on page 194.

3.4.1.1 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 16. 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.
```

Chapter 4. Controlling the Flow within a Program

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

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

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.

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:
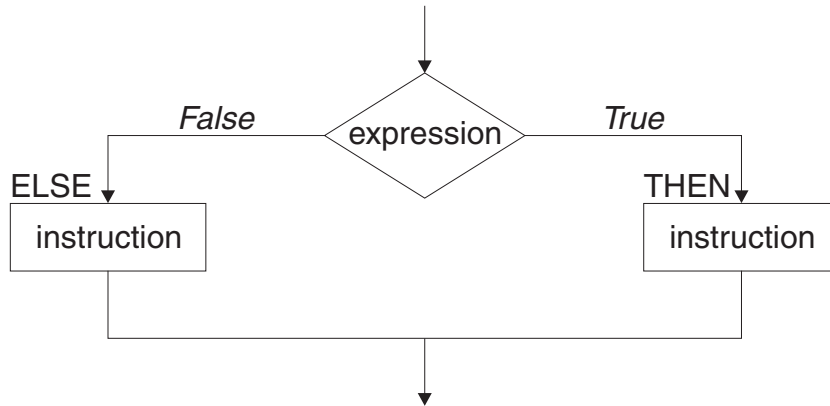
As a REXX instruction, the flowchart example looks like:

```
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.

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 17. 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!
```

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

Write the REXX instructions for the following flowchart:
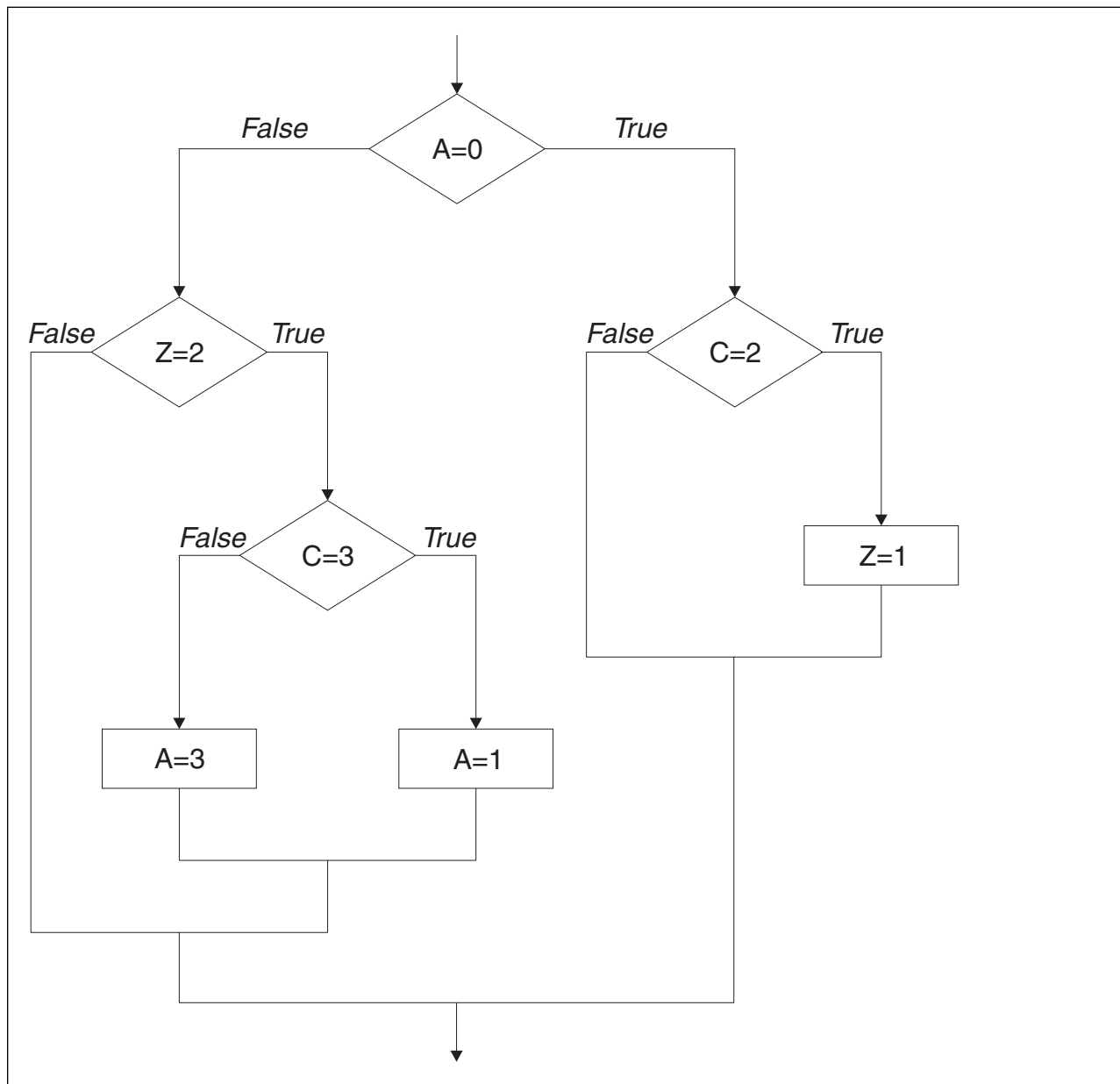
ANSWER

```
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
```

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:
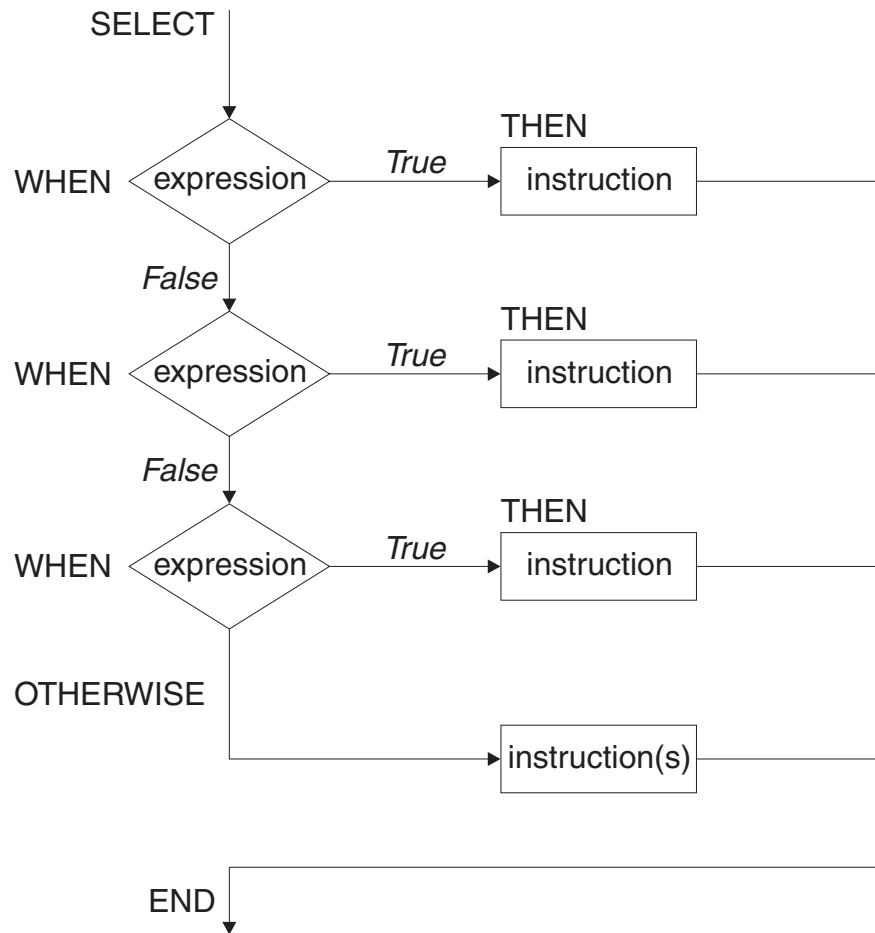
As a REXX instruction, the flowchart example looks like:

```
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 18. Example Using SELECT WHEN...OTHERWISE...END

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

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 19. Possible Solution

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.

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
```

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. An authorized user can issue the CEMT SET TASK PURGE command to halt an exec.

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 section 4.4.1, "EXIT Instruction" on page 56.

```
/****************************** 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 20. 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.

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 21. Example Using the LEAVE Instruction

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
```

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 on page 48, 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`.

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.

4.3.2.1 DO WHILE Loops

DO WHILE loops in a flowchart appear as follows:
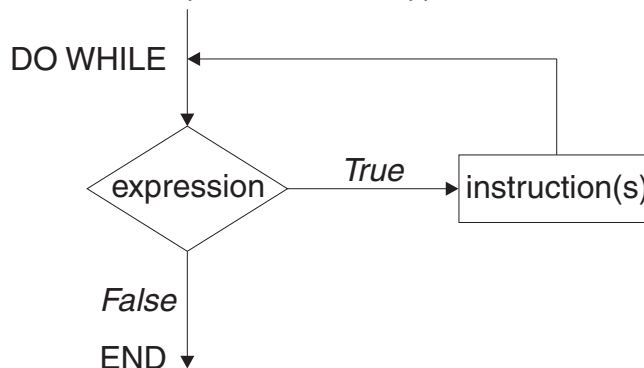
As REXX instructions, the flowchart example looks like:

```
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 on page 48. 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 22. Example Using DO WHILE

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 23. Possible Solution

4.3.2.3 DO UNTIL Loops

DO UNTIL loops in a flowchart appear as follows:

```
                    DO UNTIL

                  ┌──────────────┐
                  │ instruction(s)│
                  └──────────────┘


                      ╱╲         False
                     ╱  ╲
                    ⟨expression⟩
                     ╲  ╱
                      ╲╱

                     True

                    END
```
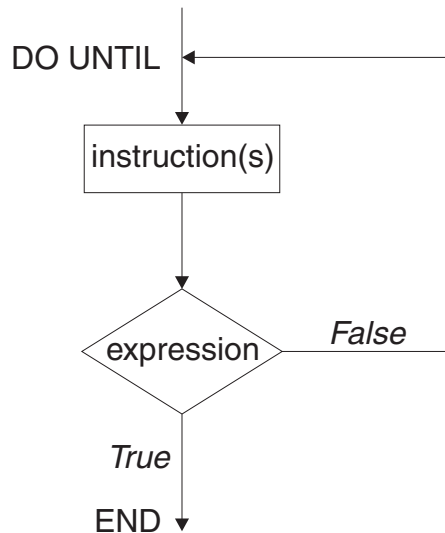
As REXX instructions, the flowchart example looks like:

```
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 24. Example Using DO UNTIL

4.3.2.4 Exercise - Using a DO UNTIL Loop

Change the program in the previous exercise on page 51 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 25. Possible Solution

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
```

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
```

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
```

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).

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" on page 67. For more detailed information on the EXIT instruction, see section 13.6, "EXIT" on page 169.

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 26. Example Using the EXIT Instruction

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. For more detailed information on the CALL and RETURN instructions, see sections 13.3, "CALL" on page 160 and 13.19, "RETURN" on page 189.

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.

```
            instruction(s)
            CALL sub1

            instruction(s)
            EXIT

            sub1:
            instruction(s)
            RETURN
```
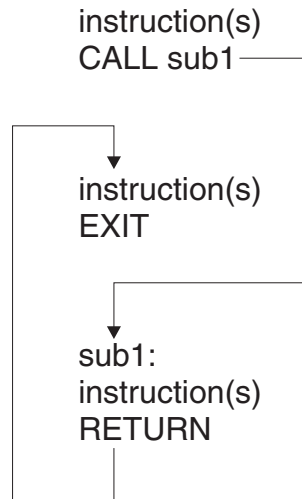
When calling an external subroutine, CALL passes control to the program name that is specified after the CALL keyword. When the external subroutine completes, you can use the RETURN instruction to return to where you left off in the calling program.

```
         MAIN

            instruction(s)
            CALL sub2

            instruction(s)
            …
            …
            ...

            sub2:

            instruction(s)
            RETURN
```
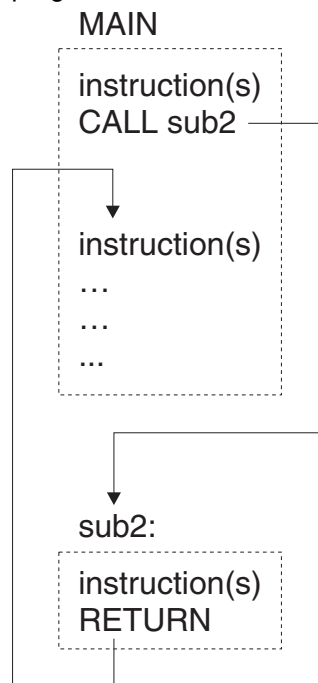
For more information about calling subroutines, see Chapter 6, "Writing Subroutines and Functions" on page 67.
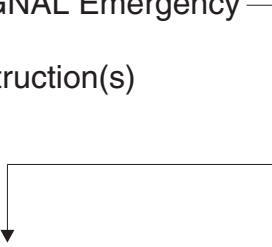
4.4.3 SIGNAL Instruction

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. For more detailed information on the SIGNAL instruction, see section 13.22, "SIGNAL" on page 192.

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

IF expression THEN
    SIGNAL Emergency
ELSE
    instruction(s)

Emergency:
instruction(s)

SIGNAL is useful for testing programs or providing 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 page 100 and section 13.22, "SIGNAL" on page 192.

Chapter 5. Using Functions

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

5.1 What is a Function?

A **function** is a sequence of instructions that can receive data, process it, 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 the calling program. More information about user-written functions appears in section 6.2, "Writing Subroutines and Functions" on page 68.

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.

5.1.1.1 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 27. 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)
```

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

  Evaluate numbers from the argument and return a particular value.

- Comparison functions

  Compare numbers, or strings, or both and return a value.

- Conversion functions

Convert one type of data representation to another type of data representation.

- Formatting functions

  Manipulate the characters and spacing in strings supplied in the argument.

- String manipulating functions

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

- Miscellaneous 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 Chapter 14, "Functions" on page 201.

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.

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).

5.2.3 Conversion Functions

Function	Description
B2X	Returns the hexadecimal representation of the input binary string. (Binary to Hexadecimal).
C2D	Returns the decimal representation of the input character string. (Character to Decimal).
C2X	Returns the hexadecimal representation of the input character string. (Character to Hexadecimal).
D2C	Returns the character representation of the input decimal string. (Decimal to Character).
D2X	Returns the hexadecimal representation of the input decimal string. (Decimal to Hexadecimal).
X2B	Returns the binary representation of the input hexadecimal string. (Hexadecimal to Binary).
X2C	Returns the character representation of the input hexadecimal string. (Hexadecimal to Character).
X2D	Returns the decimal representation of the input hexadecimal string. (Hexadecimal to Decimal).

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/CICS provides.

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[1]	Returns the word number of the first word of a specified phrase found within the input string.
INDEX[1]	Returns the character position of the first character of a specified string found in the input string.
INSERT	Returns a character string after inserting another input string into it from 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 that is the reverse of the original.
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.

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[1]	This function always returns a `0`.
LINESIZE[1]	Returns the width 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[1]	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.

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 on page 29, 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.

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 built-in functions LENGTH, see page 222, and SUBSTR, see page 227.

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 28. Possible Solution

Using Functions

Chapter 6. Writing Subroutines and Functions

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

6.1 What are Subroutines and Functions?

Subroutines and functions are routines made up of a sequence of instructions that can receive data, process it, 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 REXX subroutine that exists as a separate file.

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 arguments separated by commas. 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,...)
  ```

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.

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.

```
instruction(s)
CALL sub1

instruction(s)
EXIT

sub1:
instruction(s)
RETURN
```
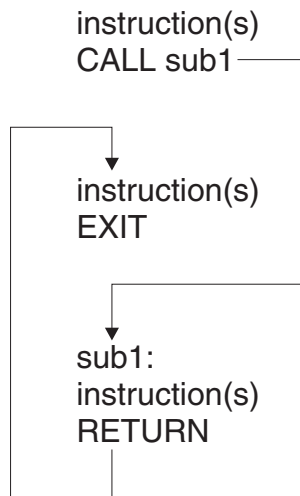
A function is a series of instructions that a program calls to perform a specific task and return a value. As Chapter 5, "Using Functions" on page 59 describes, a function can be built-in or user-written. Call a user-written function the same way as a built-in function: specify the function name immediately 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.

```
            instruction(s)
                  |
                  v
            z=func1(arg1, arg2) ─┐
                                 │
    ┌──────────────┐            │
    │              v            │
    │        instruction(s)     │
    │        EXIT               │
    │                           │
    │              v ◄──────────┘
    │        instruction(s)
    │        RETURN
    └──────────────┘
```
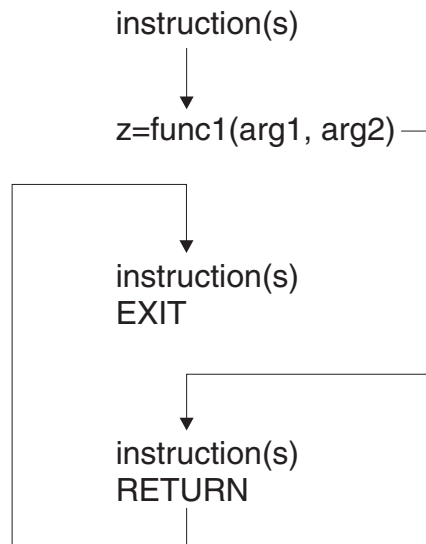
Both subroutines and functions can be **internal** (designated by a label) or **external** (designated by the subroutine or function in the REXX File System/VSE Librarian sublibrary member name). The two preceding examples illustrate an internal subroutine named `sub1` and an internal function named `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`.

```
              MAIN

          instruction(s)
          CALL sub2

          instruction(s)
          …
          …
          ..


          sub2:

          instruction(s)
          RETURN
```
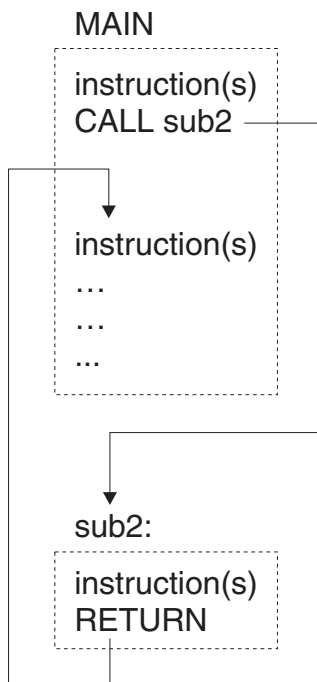
The following illustrates an external function named `func2`.

```
              MAIN

          instruction(s)

          z=func2(arg1)
          instruction(s)
          …
          …
          …
          exit


          FUNC2

          ARG var1
          instruction(s)
          RETURN value
```
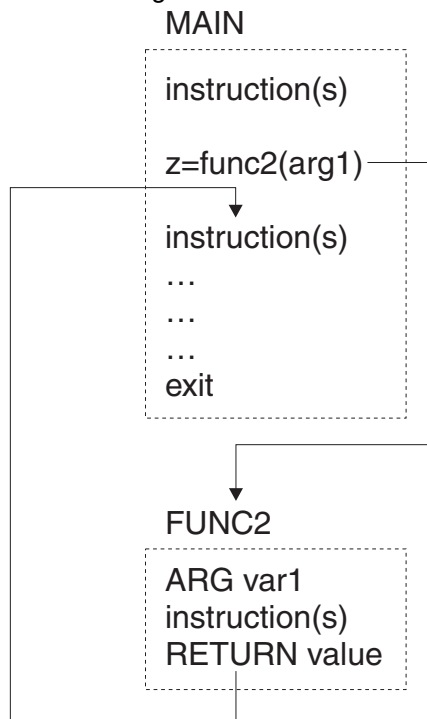
6.2.1 When to Use Internal Versus External Subroutines or Functions

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 section 14.2.1, "Search Order" on page 203.

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.)

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 29. 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 30. 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 31. 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 32. 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; section 6.2.2.2, "Passing Information by Using Arguments" on page 76 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 33. 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 34. 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 35. 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 36. 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 instruction         */
/* 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 37. 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 38. Example Using PROCEDURE EXPOSE in a Function

For more information about the PROCEDURE instruction, see section 13.15, "PROCEDURE" on page 183.

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 section 7.2, "Parsing Data" on page 86.

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 39. 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 40. 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 */
```

For more information about the ARG function see section 13.2, "ARG" on page 158.

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
  ```

6.2.3.1 Exercise - 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 41. 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 42. 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 43. Possible Solution (External Subroutine named THROW)

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 (see section 14.3.63, "WORDS" on page 235) and WORD (see section 14.3.59, "WORD" on page 234) built-in functions.

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 44. Possible Solution

6.3 Subroutines and Functions—Similarities and Differences

The following tables highlight 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.

Chapter 7. Manipulating Data

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

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.

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.

83

```
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 section 12.4.3, "Compound Symbols" on page 144.

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 and RFS commands when reading to and writing from a file. See section 25.17, "EXECIO" on page 396 for information about EXECIO. See section 25.33, "RFS" on page 416 for information about RFS. RFS is the preferred I/O method under CICS.

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

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 as simple as a list of variables to receive data. More complex templates can contain patterns; section 7.2.3, "Parsing with Patterns" on page 88 explains patterns.

7.2.1 Parsing Instructions

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

PULL Instruction: Other chapters show PULL as an instruction that reads input and assigns it to one or more variables. If the program stack contains information, the PULL instruction takes information from the program stack. When the program stack is empty, PULL takes information from the current terminal input device. See section 2.8.1, "Getting Information from the Program Stack or Terminal Input Device" on page 18 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 as a string following the exec name:

   `REXX sample 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.

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 section 13.14, "PARSE" on page 180.

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 Chapter 15, "Parsing" on page 239.

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 parsing multiple arguments that have been passed to a program or subroutine, see section 15.2.1, "Parsing Multiple Strings" on page 249.

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

Chapter 8. Using Commands from a Program

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

8.1 Types of Commands

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

REXX/CICS commands These commands provide access to miscellaneous REXX/CICS facilities. See Chapter 25, "REXX/CICS Commands" on page 369.

CICS commands These commands implement the EXEC CICS commands that application programs use to access CICS services. These commands are documented in the *CICS Transaction Server for VSE/ESA Application Programming Reference*.

SQL statements These statements are prepared and executed dynamically. See Chapter 22, "REXX/CICS DATABASE 2 Interface" on page 331

EDIT commands These commands invoke the editor facilities from the REXX/CICS macros. See Chapter 18, "REXX/CICS Text Editor" on page 271.

RFS commands These commands are for the REXX File System (RFS). See Chapter 19, "REXX/CICS File System" on page 299.

RLS commands These commands are for the REXX List System (RLS). See Chapter 20, "REXX/CICS List System" on page 317.

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. Therefore, RC contains the return code from the most recently issued command.

8.1.1 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.

Many CICS commands use single quotation marks within the command. For this reason, it is recommended that, as a matter of course, you enclose CICS commands within double quotation marks.

The following example places the word `test` in the temporary storage queue `ABC`.

```
"CICS WRITEQ TS QUEUE('ABC') FROM('test')"
```

8.1.2 Using Variables in Commands

When a command contains a variable, the value of the variable is not substituted if the variable is within quotation marks. The language processor uses the value of a variable only for variables outside quotation marks.

8.1.3 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" on page 67). You can also call a program from another program explicitly with the EXEC command. 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/CICS 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
```

8.2 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.

8.2.1 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 the commands.   . 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:

REXXCICS This is the default REXX/CICS command environment. All REXX/CICS, SQL, EDIT, RFS, or RLS commands can be issued from this environment. However, CICS commands must be prefixed with CICS, SQL statements with EXECSQL, EDIT commands with EDITSVR, RFS commands with RFS, and RLS commands with RLS.

CICS This is an optional environment that only issues CICS commands. The first word of the host command string is the command name (for example: SEND, RECEIVE).

EXECSQL This is an optional environment that issues SQL statements (SELECT) to the CICS/DB2 interface.

EDITSVR This is an optional environment that creates the edit session.

FLSTSVR This is an optional environment that executes commands for the File List Utility.

RFS This is an optional environment that executes commands for the REXX File System.

RLS This is an optional environment that executes commands for the REXX List System.

Note: It is recommended that the default environment of REXXCICS be used for all commands (that is, the ADDRESS instruction should not be specified).

8.2.2 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 prepared appropriately 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.

8.2.3 Changing the Host Command Environment

You can change the host command environment either from the default environment (REXXCICS) 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 only a single command, and one affects all commands issued after the instruction.

- **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.

- **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.

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.

8.2.3.1 Determining the Active Host Command Environment

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

Example

```
curenv = ADDRESS()
```

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

Chapter 9. Diagnosing Problems within a Program

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

9.1 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. For information about using the TRACE instruction to evaluate expressions, see section 3.4, "Tracing Expressions with the TRACE Instruction" on page 35. For information about using the TRACE instruction to evaluate host commands, see the next section, 9.1.1, "Tracing Commands with the TRACE Instruction."

- REXX/CICS 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.

9.1.1 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 execution, then executes it and sends the return code from the command to the current terminal output device. For more information on specifying the current terminal output device, refer to the SET TERMOUT command (see page 421).

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 execution and sends the return code from the command to the terminal.

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

For more information about the TRACE instruction, see section 13.23, "TRACE" on page 194.

9.1.2 Using REXX Special Variables RC and SIGL

The REXX language has three special variables: RC, SIGL, and RESULT. REXX/CICS sets these variables during particular situations and you can use them in an expression at any time. If REXX/CICS 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.

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

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, a SIGNAL or a 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 which 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.

For more information about the SIGNAL instruction, see section 13.22, "SIGNAL" on page 192.

9.1.3 Tracing with the Interactive Debug Facility

The interactive debug facility lets a user control the execution of a program. The language processor reads from the terminal, and writes output to the terminal.

9.1.3.1 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.

9.1.3.2 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. Repeated 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
```

9.1.3.3 Ending Interactive Debug

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

- Use the TRACE OFF instruction as input. 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.

9.1.4 Saving Interactive TRACE Output

REXX/CICS provides the ability to route trace output to a file. The REXX command SET TERMOUT routes linemode output (such as SAY and TRACE output) to a file instead of, or in addition to, the current terminal device (see page 421).

Chapter 10. Programming Style and Techniques

The *method* you use for constructing your programs is just as important as the *language* you use to write them.

10.1 Consider the Data

When you are faced with the task of writing a program, the first thing to consider is the data you are required to process. Make a list of the input data—what are the items and what are the possible values of each? If the items have a kind of structure or pattern, draw a diagram to illustrate it. Then do the same for the output data. Study your two diagrams and try to see if they fit together. If they do, you are well on the way to designing your program.

Next, write the specification that the user will use. This might be a written specification, a HELP file or both.

Last of all, write your program.

Here is a little example:

> You are required to write an interactive program that invites the user to play "Heads or tails." The game can be played as long as the user likes. To end the game the user should reply `Quit` in answer to the question "Heads or tails?" The program is arranged so that the computer *always* wins.

Think about how you would write this program.

The computer starts off with:

```
Let's play a game!  Type "Heads", "Tails",
or "Quit"
and press ENTER.
```

This means that there are *four* possible inputs:

- HEADS
- TAILS
- QUIT
- None of these three.

And so the corresponding outputs should be:

- Sorry. It was TAILS. Hard luck!
- Sorry. It was HEADS. Hard luck!
- That's not a valid answer. Try again!

And this sequence must be repeated indefinitely, ending with the return to CICS.

Now that you understand the specification, the input data and the output data, you are ready to write the program.

If you had started off by writing down some instructions without considering the data, it would have taken you longer.

10.1.1 Test Yourself...

Write the program. If you are careful, it should run the first time!

10.1.1.1 Answer:

```
/* CON EXEC */

/* Tossing a coin.  The machine is lucky, not the user */

do forever
   say "Let's play a game!  Type 'Heads', 'Tails'",
       "or 'Quit' and press ENTER."
   pull answer

   select
      when answer = "HEADS"
         then say "Sorry!  It was TAILS.  Hard luck!"
      when answer = "TAILS"
         then say "Sorry!  It was HEADS.  Hard luck!"
      when answer = "QUIT"
         then exit
      otherwise
         say "That's not a valid answer.  Try again!"
   end
   say
end
```

10.2 Happy Hour

Here is a chance to have some fun.

This is a very simple arcade game. Type it in and play it with your friends. Later in this chapter, you may want to improve it.

```
/* CATMOUSE EXEC                                          */

/* The user says where the mouse is to go.  But where    */
/* will the cat jump?                                     */

say "This is the mouse ---------->   @"
say "These are the cat's paws --->  ( )"
say "This is the mousehole ------>   O"
say "This is a wall ------------->   |"
say
say "You are the mouse.  You win if you reach",
    "the mousehole.  You cannot go past"
say "the cat.  Wait for him to jump over you.",
    "If you bump into him you're caught!"
say
say "The cat always jumps towards you, but he's not",
    "very good at judging distances."
say "If either player hits the wall he misses a turn."
say
say "Enter a number between 0 and 2 to say how far to",
    "the right you want to run."
say "Be careful, if you enter a number greater than 2 then",
    "the mouse will freeze and the cat will move!"
say


/*-------------------------------------------------------*/
/* Parameters that can be changed to make a different    */
/* game                                                  */
/*-------------------------------------------------------*/
len = 14              /* length of corridor             */
hole = 14             /* position of hole               */
spring = 5            /* maximum distance cat can jump  */
mouse = 1             /* mouse starts on left           */
cat = len             /* cat starts on right            */
/*-------------------------------------------------------*/
/* Main program                                          */
/*-------------------------------------------------------*/
do forever
   call display
   /*----------------------------------------------------*/
   /* Mouse's turn                                       */
   /*----------------------------------------------------*/
   pull move
   if datatype(move,whole) & move >= 0 & move <= 2
   then select
      when mouse + move > len then nop     /* hits wall */
      when cat > mouse,
         & mouse + move >= cat             /* hits cat  */
                                   /* continued ... */
```

Figure 45. CATMOUSE EXEC 1/2

```
      then mouse = cat
      otherwise                            /* moves     */
      mouse = mouse + move
      end
      if mouse = hole then leave         /* reaches hole */
  if mouse = cat then leave             /* hits cat    */
   /*------------------------------------------------*/
   /* Cat turn                                        */
   /*------------------------------------------------*/
   jump = random(1,spring)
   if cat > mouse then do     /* cat tries to jump left */
      Temp = cat - jump
      if Temp < 1 then nop           /* hits wall  */
      else cat = Temp
      end
   else do                   /* cat tries to jump right */
      if cat + jump > len then nop        /* hits wall  */
      else cat = cat + jump
      end
      if cat = mouse then leave
      end
      /*------------------------------------------------------*/
      /* Conclusion                                            */
/*------------------------------------------------------*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
/*------------------------------------------------------*/
/* Subroutine to display the state of play         */
/*                                                  */
/* Input: CAT and MOUSE                             */
/*                                                  */
/* Design note:  each position in the corridor occupies */
/* three character positions on the screen.          */
/*------------------------------------------------------*/

corridor = copies(" ",3*len)              /* corridor  */
corridor = overlay("O",corridor,3*hole-1)  /* hole     */

if mouse ¬= len                      /* mouse in hole? */
then corridor = overlay("@",corridor,3*mouse-1)/* mouse */

corridor = overlay("(",corridor,3*cat-2)      /* cat   */
corridor = overlay(")",corridor,3*cat)
say "   |"corridor"|"
return
```

Figure 46. CATMOUSE EXEC 2/2

Good job! Now, take a while to put your new skills into action, or continue reading.

10.3 Designing a Program

Still thinking about *method*, which is just as important as *language*, let us take another look at CATMOUSE EXEC.

The program is about a cat and a mouse and their positions in a corridor. At some stage their positions will have to be pictured on the screen. The whole thing is too complicated to think about all at once; the first step is to break it down into:

- **Main program**: calculate their positions
- **Display subroutine**: display their positions.

Now let us look at main program. The user (who plays the mouse) will want to see where everybody is before making a move. The cat will not. The next step is to break the main program down further, into:

```
Do forever
   call Display
   Mouse's move
   Cat's move
end
Conclusion
```

10.3.1 Methods for Designing Loops

The method for designing loops is to ask two questions:

- Will it always end?
- Whenever it terminates, will the data meet the conditions required?

Well, the loop terminates (and the game ends) when:

1. The mouse runs to the hole.
2. The mouse runs into the cat.
3. The cat catches the mouse.

10.3.2 The Conclusion

At the end of the program, the user must be told what happened.

```
call display
say who won
```

10.3.3 What Do We Have So Far?

Putting all this together, we have:

```
/*------------------------------------------------------*/
/* Main program                                         */
/*------------------------------------------------------*/
do forever
   call display
   /*------------------------------------------------------*/
   /* Mouse's turn                                         */
   /*------------------------------------------------------*/
    ...

   if mouse = hole then leave         /* reaches hole */
   if mouse = cat then leave          /* hits cat     */
   /*------------------------------------------------------*/
   /* Cat's turn                                           */
   /*------------------------------------------------------*/
    ...

   if cat = mouse then leave
end

/*------------------------------------------------------*/
/* Conclusion                                           */
/*------------------------------------------------------*/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit

/*------------------------------------------------------*/
/* Subroutine to display the state of play              */
/*                                                      */
/* Input: CAT and MOUSE                                 */
/*                                                      */
/*------------------------------------------------------*/
display:
 ...
```

The method that we have just discussed is sometimes called *stepwise refinement*.
You start with a specification (which may be incomplete). Then you divide the
proposed program into routines, such that each routine will be easier to code than
the program as a whole. Then you repeat the process for each of these routines
until you reach routines that you are sure you can code correctly at the first
attempt.

While you are doing this, keep asking yourself two questions:

- What data does this routine handle?

- Is the specification complete?

10.3.4 Stepwise Refinement: An Example

Granny is going to knit you a warm woolen garment to wear when you go sailing. This is what she might do.

1. Knit front
2. Knit back
3. Knit left arm
4. Knit right arm
5. Sew pieces together.

Each of these jobs is simpler to describe than the job of knitting a pullover. In computer jargon, breaking a job down into simpler jobs is called *stepwise refinement*.

At this stage, look at the specification again. A sailor might need to put on the pullover in the dark, quickly, without worrying about the front or back. Therefore, the front should be the same as the back; and the two sleeves should also be the same. This could be programmed:

```
do 2
   CALL Knit_body_panel
end

do 2
   CALL Knit_sleeve
end

CALL sew_pieces_together
```

In programming, the best method is to go on refining your program, working from the top, until you get down to something that is easy to code.

Top down is the best approach.

10.3.5 Reconsider the Data

When you are refining your program, your objective is to make each piece simpler. This almost certainly means:

- Simpler input data for each segment or routine
- Simpler output data for each segment or routine
- Simpler processing
- And, therefore, simpler code.

If your pieces really are simpler, they will probably have simpler names, too. For instance:

- Knit cuff

rather than

- Make ribbing for cuffs and waistband.

10.4 Correcting Your Program

If you cannot understand why your program is giving wrong results, you can:

- Modify your program so that it tells you what it is doing

- Use some of the REXX interactive trace facilities (See E.1, "Interactive Debugging of Programs" on page 469).

You will gradually learn which of these techniques suits you better.

10.4.1 Modifying Your Program

You can put extra instructions into your program, such as:

```
  ...
say "Checkpoint A.  x =" x
  ...
say "End of first routine"
  ...
```

10.4.2 Tracing Your Program

Or you can use the TRACE instruction, described on page 194.

- To find out where your program is going, use TRACE Labels. The example shows a program and the trace it gives on the screen.

```
/* ROTATE EXEC                                      */

/* Example:  two iterations of wheel, six iterations */
/* of cog.  On the first three iterations,  "x < 2"  */
/* is true. On the next three, it is false.          */
trace L
do x = 1 to 2
wheel:
   do 3
cog:
      if x < 2 then do
true:
      end
      else do
false:
      end
   end
end
done:
```

Figure 47. ROTATE EXEC

This gives the trace:

```
rotate
    6 *-*  wheel:
    8 *-*   cog:
   10 *-*     true:
    8 *-*   cog:
   10 *-*     true:
    8 *-*   cog:
   10 *-*     true:
    6 *-*  wheel:
    8 *-*   cog:
   13 *-*     false:
    8 *-*   cog:
   13 *-*     false:
    8 *-*   cog:
   13 *-*     false:
   17 *-* done:
```

- To see how the language processor is computing expressions, use TRACE Intermediates.

- To find out whether you are passing the right data to a command or subroutine, use TRACE Results.

- To make sure that you get to see nonzero return codes from commands, use TRACE Errors.

10.5 Coding Style

The only sure way of finding out whether a program is correct is to read it. Therefore, programs must be easy to read. Naturally, *easy to read* means different things to different programmers. All we can do here is to give examples of different styles, and leave you to choose the style you prefer.

A very good way to get your program checked is to ask a fellow worker to read it. Be sure to choose a coding style that your fellow workers find easy to read.

Most people would find the following program fragment difficult to read.

```
/*******************************************************/
/* SAMPLE #1:  A portion of CATMOUSE EXEC (page 105),  */
/* not divided into segments and written with no       */
/* indentation, and no comments.  This style is not    */
/* recommended.                                        */
/*******************************************************/

do forever
call display
pull move
if datatype(move,whole) & move >= 0 & move <=2
then select
when mouse+move > len then nop
when cat > mouse,
& mouse+move >= cat,
then mouse = cat
otherwise
mouse = mouse + move
end
if mouse = hole then leave
if mouse = cat  then leave
jump = random(1,spring)
if cat > mouse then do
if cat-jump < 1 then nop
else cat = cat-jump
end
else do
if cat+jump > len then nop
else cat = cat+jump
end
if cat = mouse then leave
end
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
```

This next example is easier to read. It is divided into segments, each with its own heading. The comments on the right are sometimes called *remarks*. They can help the reader get a general idea of what is going on.

```
/********************************************************/
/* SAMPLE #2:  A portion of CATMOUSE EXEC (page 105),   */
/* divided into segments and written with 'some'        */
/* indentation and 'some' comments.                     */
/********************************************************/

/********************************************************/
/* Main program                                         */
/********************************************************/
do forever
   call display
   /********************************************************/
   /* Mouse's turn                                         */
   /********************************************************/
   pull move
   if datatype(move,whole) & move >= 0 & move <=2
   then select
      when mouse+move > len then nop        /* hits wall */
      when cat > mouse,
         & mouse + move >= cat,             /* hits cat  */
      then mouse = cat
      otherwise                             /* moves     */
      mouse = mouse + move
   end
   if mouse = hole then leave            /* reaches hole */
   if mouse = cat then leave             /* hits cat     */
   /********************************************************/
   /* Cat's turn                                           */
   /********************************************************/
   jump = random(1,spring)
   if cat > mouse then do      /* cat tries to jump left */
      if cat - jump < 1 then nop            /* hits wall  */
      else cat = cat - jump
   end
   else do                    /* cat tries to jump right */
      if cat + jump > len then nop          /* hits wall  */
      else cat = cat + jump
   end
   if cat = mouse then leave
end
/********************************************************/
/* Conclusion                                           */
/********************************************************/
call display
if cat = mouse then say "Cat wins"
else say "Mouse wins"
exit
```

This next example has additional features that are popular with some programmers. Keywords written in uppercase and a different indentation style highlight the structure of the code; the abundant comments recall the detail of the specification.

```
/**********************************************************/
/* SAMPLE #3:  A portion of CATMOUSE EXEC (page 105),   */
/* divided into segments and written with 'more'         */
/* indentation and 'more' comments.                      */
/* Note commands in uppercase (to highlight logic)      */
/**********************************************************/


/**********************************************************/
/* Main program                                          */
/**********************************************************/
DO FOREVER
  CALL display
  /********************************/
  /* Mouse's turn                 */
  /********************************/
  PULL move
  IF datatype(move,whole) & move >= 0 & move <=2
    THEN SELECT
          WHEN mouse+move > len     /* mouse hits wall */
            THEN nop                /* and loses turn  */
          WHEN cat > mouse,
              & mouse+move >= cat, /* mouse hits cat  */
            THEN mouse = cat        /* and loses game  */
          OTHERWISE mouse = mouse + move /* mouse ...  */
        END                        /* moves to new location */
  IF mouse = hole THEN LEAVE   /* mouse is home safely */
  IF mouse = cat  THEN LEAVE   /* mouse hits cat (ouch) */
  /********************************/
  /* Cat's turn                   */
  /********************************/
  jump = RANDOM(1,spring)      /* determine cat's move  */
  IF cat > mouse               /* cat must jump left    */
    THEN DO
          IF cat-jump < 1       /* cat hits wall    */
            THEN nop            /* misses turn      */
            ELSE cat = cat-jump  /* cat jumps left   */
        END
    ELSE DO                     /* cat must jump right    */
          IF cat+jump > len     /* cat hits wall    */
            THEN nop            /* misses turn      */
            ELSE cat = cat+jump  /* cat jumps right  */
        END
  IF cat = mouse THEN LEAVE     /* cat catches mouse      */
END
/**********************************************************/
/* Conclusion                                            */
/**********************************************************/
CALL display                          /* on final display */
  IF cat = mouse                      /* who won?        */
    THEN say "Cat wins"               /* ... the cat     */
    ELSE say "Mouse wins"             /* ... the mouse   */
EXIT
```

Congratulations! Now you know the best ways to program your REXX execs.
Have fun writing programs that will make your life easier!

Part 2. Reference

Chapter 11. Introduction

This introductory section:

- Identifies the reference's purpose and audience
- Explains how to use the reference
- Gives an overview of product features
- Explains how to read a syntax diagram.

11.1 Who Should Read This Reference

This reference describes the CICS Transaction Server for VSE/ESA REXX Guide and Reference Interpreter (hereafter referred to as the interpreter or language processor) and the REstructured eXtended eXecutor (called REXX) language. This reference is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, or Pascal).

Descriptions include the use and syntax of the language and how the language processor "interprets" the language while a program is running under the REXX/CICS interpreter. The reference also describes:

- Programming services that let you interface with REXX and the language processor.
- Customizing services that let you customize REXX processing and how the language processor accesses and uses system services, such as storage and I/O requests.

11.2 How to Use This Reference

This part of the book is a reference rather than a tutorial. It assumes you are already familiar with REXX programming concepts. The material in this reference is arranged in chapters:

- Chapter 11, "Introduction"
- Chapter 12, "REXX General Concepts"
- Chapter 13, "Keyword Instructions" (in alphabetic order)
- Chapter 14, "Functions" (in alphabetic order)
- Chapter 15, "Parsing" (a method of dividing character strings, such as commands)
- Chapter 16, "Numbers and Arithmetic"
- Chapter 17, "Conditions and Condition Traps"
- Chapter 18, "REXX/CICS Text Editor"
- Chapter 19, "REXX/CICS File System"
- Chapter 20, "REXX/CICS List System"
- Chapter 21, "REXX/CICS Command Definition"
- Chapter 22, "REXX/CICS DATABASE 2 Interface"
- Chapter 23, "REXX/CICS High-level Client/Server Support"
- Chapter 24, "REXX/CICS Panel Facility"
- Chapter 25, "REXX/CICS Commands"

There are several appendixes covering:

- Appendix A, "Error Numbers and Messages"
- Appendix B, "Return Codes"

- Appendix C, "Double-Byte Character Set (DBCS) Support"
- Appendix D, "Reserved Keywords and Special Variables"
- Appendix E, "Debug Aids"
- Appendix F, "REXX/CICS Business Value Discussion"
- Appendix G, "System Definition/Customization/Administration"
- Appendix H, "Security"
- Appendix I, "Performance Considerations"
- Appendix J, "Basic Mapping Support Example"

11.3 Overview of Product Features

The following list is the product features of REXX/CICS described in this section.

- 11.3.1, "SAA Level 2 REXX Language Support Under REXX/CICS"
- 11.3.2, "Support for the Interpretive Execution of REXX Execs"
- 11.3.3, "CICS-Based Text Editor for REXX Execs and Data"
- 11.3.4, "VSAM-Based File System for REXX Execs and Data"
- 11.3.6, "Dynamic Support for EXEC CICS Commands"
- 11.3.7, "REXX Interface to CEDA and CEMT Transaction Programs"
- 11.3.8, "High-level Client/Server Support"
- 11.3.9, "Support for Commands Written in REXX"
- 11.3.10, "Command Definition of REXX Commands"
- 11.3.11, "Support for System and User Profile Execs"
- 11.3.12, "Shared Execs in Virtual Storage"
- 11.3.13, "SQL Interface."

11.3.1 SAA Level 2 REXX Language Support Under REXX/CICS

REXX/CICS is currently at REXX language level 3.48 and provides all Systems Application Architecture* (SAA) REXX Level 2 capability (as described in the *SAA Common Programming Interface REXX Level 2 Reference*, SC24-5549) except for stream I/O and REXX language processor exits.

Note: The POWER HOST environment is not implemented in REXX/CICS.

11.3.2 Support for the Interpretive Execution of REXX Execs

Interpretive execution of REXX execs provides the ability to create and run REXX execs without first compiling them. The use of the interpreter provides a very productive development, customization, prototyping, and command list (CLIST) processing environment. This is because it provides a fast development cycle, source level interactive debug, and a native CICS-based development environment, in one integrated package.

Note: REXX execs can freely invoke CICS programs and transactions written in any CICS supported language.

11.3.3 CICS-Based Text Editor for REXX Execs and Data

A native CICS text editor, similar to the VM/CMS XEDIT editor, is being provided as part of REXX/CICS, so execs (and other data) can be created and modified directly under CICS. Edit support is provided for files residing in the provided VSAM-based REXX file system (RFS), and for files existing in members in the VSE Librarian sublibraries.

11.3.4 VSAM-Based File System for REXX Execs and Data

REXX/CICS includes the REXX File System (RFS), a high-level file system that is hierarchically structured and is similar to the Operating System/2* (OS/2*), Advanced Interactive Executive* (AIX*), and the VM Shared File systems. The RFS automatically provides each REXX user with a file system in which to store execs and data. There is a file list utility to facilitate working with this file system, the text editor supports editing members of this file system, and execs to be run are loaded from this file system. This file system is VSAM based for performance, security, and portability reasons.

11.3.5 VSE Librarian Sublibraries

These can be used as an alternative to the RFS for REXX execs. A REXX exec in a VSE librarian sublibrary must have have a member type of .PROC.

11.3.6 Dynamic Support for EXEC CICS Commands

Support for most EXEC CICS API commands is included in REXX/CICS. This is a dynamic interface (no EXEC CICS command translation pre-processing step is needed). This support is provided through the addition of an ADDRESS CICS command environment.

11.3.7 REXX Interface to CEDA and CEMT Transaction Programs

This allows CEDA and CEMT commands to be easily issued from REXX execs, with any subsequent output placed into a REXX variable, instead of being displayed at the terminal. This facilitates the automation of many CICS administration and operation activities, and is an excellent programmer aid.

11.3.8 High-level Client/Server Support

REXX/CICS provides integrated client/server support to REXX execs by providing facilities to allow REXX execs to act as clients (which make requests to REXX/CICS servers) and by providing facilities to allow REXX execs to act as servers (with the ability to wait for and process requests from REXX/CICS clients).

REXX/CICS facilities are provided which allow REXX/CICS servers to wait on requests from clients (WAITREQ), and to retrieve (C2S) and set (S2C) the contents of client REXX variables.

Note: Servers do not execute as nested execs of clients, but rather execute as parallel entities.

Servers use Automatic Server Initiation (ASI) to start automatically when they receive their first request.

11.3.9 Support for Commands Written in REXX

REXX/CICS supports the ability for users to write new REXX/CICS commands in REXX. These commands do not function as nested REXX execs, and unlike nested REXX execs have the ability to get and set the values of REXX variables in the user exec that issued the command. Therefore, commands written in REXX can have similar capabilities as commands written in Assembler or other languages. Also, commands can be quickly written in REXX to speed systems development (in a building block structure), and then can selectively be rewritten in

Assembler (or any other CICS supported language) at a later date, if performance requirements dictate.

11.3.10 Command Definition of REXX Commands

REXX/CICS includes as one of its basic facilities, the ability for systems administrators and users to easily and dynamically define new REXX commands, either on a system-wide or user-by-user basis. One of the greatest strengths of REXX is its ability to interface cleanly with other products, applications, and system services. The goal for providing a command definition facility for new or existing commands is to facilitate the rapid and consistent high-level integration of various products and services together through the use of REXX. REXX command definition is accomplished with the REXX/CICS DEFCMD and DEFSCMD commands.

11.3.11 Support for System and User Profile Execs

To facilitate REXX/CICS system and user environment tailoring, REXX/CICS attempts to execute CICSTART, CICSPROF, and user PROFILE execs, if they exist. CICSTART is the system profile exec (STARTUP profile) and it is issued before the first user exec is run after CICS system restart. CICSPROF is the system user profile exec and it is issued when a user enters REXX/CICS for the first time since the CICS system restart. CICSPROF also invokes the user PROFILE.

11.3.12 Shared Execs in Virtual Storage

REXX/CICS supports shared copies of REXX execs residing in virtual storage. Shared execs improve the interactive response time of REXX applications, and sharing reduces the total virtual storage requirement. Execs can be pre-loaded by using the EXECLOAD command. Common system-wide execs are good candidates for pre-loading through the placement of EXECLOAD commands in the CICSTART exec.

11.3.13 SQL Interface

REXX programs may contain SQL statements. These statements are interpreted and executed dynamically. The results of the SQL statements are placed into REXX variables for use within the REXX program.

11.3.13.1 Programming Considerations

To embed SQL statements withn a REXX exec, the host command environment must be changed, The ADDRESS instruction, followed by the name of the environment, is used to change the host command environment.

The REXX/CICS command envlironment that supports SQL statements is **EXECSQL**.

11.3.13.2 Embedding SQL statments

You can pass the following SQL statements directly to the EXECSQL command environment:

    ALTER
    CREATE
    COMMENT ON
    DELETE

EXPLAIN
INSERT
GRANT
INSERT
LABEL ON
LOCK
REVOKE
SELECT
UPDATE

You cannot use the following SQL statements:

BEGIN DECLARE SECTION
CLOSE
COMMIT
CONNECT
DECLARE CURSOR
DESCRIBE
END DECLARE SECTION
EXECUTE
EXECUTE IMMEDIATE
FETCH
INCLUDE
OPEN
PREPARE
ROLLBACK
WHENEVER

11.4 How to Read the Syntax Diagrams

Throughout this book, syntax is described using the structure defined below.

* Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ►►─── symbol indicates the beginning of a statement.

  The ───► symbol indicates that the statement syntax is continued on the next line.

  The ►─── symbol indicates that a statement is continued from the previous line.

  The ───►◄ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ►─── symbol and end with the ───► symbol.

* Required items appear on the horizontal line (the main path).

  ►►──STATEMENT──*required_item*──────────────────────────────────────►◄

* Optional items appear below the main path.

  ►►──STATEMENT──┬──────────────┬──────────────────────────────────────►◄
                     └─*optional_item*─┘

* If you can choose from two or more items, they appear vertically, in a stack.

  If you *must* choose one of the items, one item of the stack appears on the main path.

```
►►──STATEMENT──┬─required_choice1─┬──────────────────────────────────►◄
               └─required_choice2─┘
```
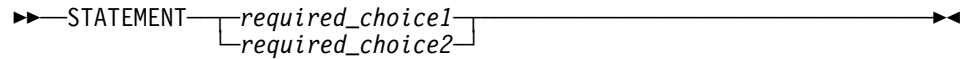
If choosing is optional, the entire stack appears below the main path.

```
►►──STATEMENT────────────────────────────────────────────────────────►◄
               ├─optional_choice1─┤
               └─optional_choice2─┘
```
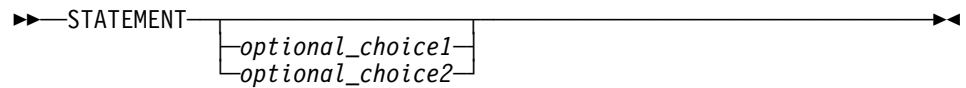
- If one of the items is the default, it appears above the main path and the
  remaining choices are shown below.

```
               ┌─default_choice─┐
►►──STATEMENT──┼────────────────┼─────────────────────────────────────►◄
               ├─optional_choice─┤
               └─optional_choice─┘
```
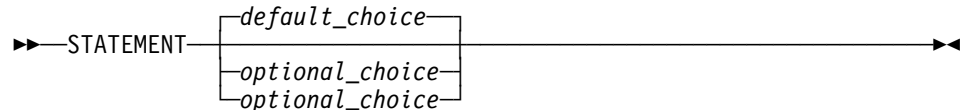
- An arrow returning to the left above the main line indicates an item that can be
  repeated.

```
               ┌─────────────────┐
►►──STATEMENT──▼─repeatable_item──┴───────────────────────────────────►◄
```
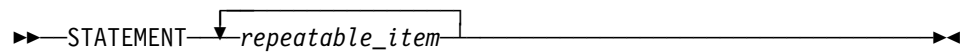
A repeat arrow above a stack indicates that you can repeat the items in the
stack.

- A set of vertical bars around an item indicates that the item is a *fragment*, a
  part of the syntax diagram that appears in greater detail below the main
  diagram.

```
►►──STATEMENT──┤ fragment ├───────────────────────────────────────────►◄
```

fragment:
```
├──expansion_provides_greater_detail──────────────────────────────────┤
```
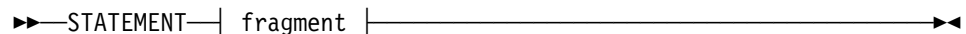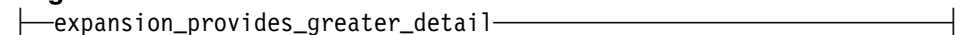
- Keywords appear in uppercase (for example, PARM1). They must be spelled
  exactly as shown but can be specified in any case. Variables appear in all
  lowercase letters (for example, *parmx*). They represent user-supplied names or
  values.

- If punctuation marks, parentheses, arithmetic operators, or such symbols are
  shown, you must enter them as part of the syntax.

Chapter 12. REXX General Concepts

The REstructured eXtended eXecutor (REXX) language is particularly suitable for:

- Command procedures
- Application front ends
- User-defined macros (such as editor subcommands)
- User-defined XEDIT subcommands
- Prototyping
- Personal computing.

REXX is a general purpose programming language like PL/I. REXX has the usual structured-programming instructions—IF, SELECT, DO WHILE, LEAVE, and so on—and a number of useful built-in functions.

The language imposes no restrictions on program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Indentation is allowed. You can, therefore, code programs in a format that emphasizes their structure, making them easier to read.

There is no limit to the length of the values of variables, as long as all variables fit into the storage available.

Implementation maximum: No single request for storage can exceed the fixed limit of 16MB. This limit applies to the size of a variable plus any control information. It also applies to buffers obtained to hold numeric results.

The limit on the length of symbols (variable names) is 250 characters.

You can use compound symbols, such as

`NAME.Y.Z`

(where `Y` and `Z` can be the names of variables or can be constant symbols), for constructing arrays and for other purposes.

REXX programs can reside in REXX File System directories or in VSE sublibraries. REXX programs usually have a file type of EXEC or, if they are in a VSE Librarian sublibrary, a member type of PROC.

A language processor (interpreter) runs REXX programs. That is, the program is processed line-by-line and word-by-word, without first being translated to another form (compiled). The advantage of this to the user is that if the program fails with a syntax error of some kind, the point of error is clearly indicated; usually, it will not take long to understand the difficulty and make a correction.

12.1 Structure and General Syntax

REXX programs are recommended to start with a comment. REXX/CICS does not require this. However, for portability reasons, you are recommended to start each REXX program with a comment that begins on the first line and includes the word `REXX`. The example in Figure 48 on page 130 illustrates this. The program starts with a comment and the characters "REXX" are in the first line (line 1).

```
              /* REXX program */
                 ...
                 ...
                 ...
              EXIT
```

Figure 48. Example of Using the REXX Program Identifier

A REXX program is built from a series of **clauses** that are composed of:

- Zero or more blanks (which are ignored)
- A sequence of tokens (see section 12.1.3, "Tokens" on page 131)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:).

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to operator characters and special characters (see page 135) are also removed.

Implementation maximum: The length of a clause cannot exceed 16K.

12.1.1 Characters

A character is a member of a defined set of elements that is used for the control or representation of data. You can usually enter a character with a single keystroke. The coded representation of a character is its representation in digital form. A character, the letter A, for example, differs from its *coded representation* or encoding. Various coded character sets (such as ASCII and EBCDIC) use different encodings for the letter A (decimal values 65 and 193, respectively). This book uses characters to convey meanings and not to imply a specific character code, except where otherwise stated. The exceptions are certain built-in functions that convert between characters and their representations. The functions C2D, C2X, D2C, X2C, and XRANGE have a dependence on the character set in use.

A code page specifies the encodings for each character in a set. You should be aware that:

- Some code pages do not contain all characters that REXX defines as valid (for example, ¬, the logical NOT character).

- Some characters that REXX defines as valid have different encodings in different code pages (for example, !, the exclamation point).

For information about Double-Byte Character Set characters, see Appendix C, "Double-Byte Character Set (DBCS) Support" on page 449.

12.1.2 Comments

A comment is a sequence of characters (on one or more lines) delimited by **/*** and ***/**. Within these delimiters any characters are allowed. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. They are called **nested comments**. Comments can be anywhere and can be of any length. They have no effect on the program, but they do act as separators. (Two tokens with only a comment in between are not treated as a single token.)

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

Take special care when commenting out lines of code containing /* or */ as part of a literal string. Consider the following program segment:

```
01    parse pull input
02    if substr(input,1,5) = '/*123'
03      then call process
04    dept = substr(input,32,5)
```

To comment out lines 2 and 3, the following change would be incorrect:

```
01    parse pull input
02 /* if substr(input,1,5) = '/*123'
03      then call process
04 */ dept = substr(input,32,5)
```

This is incorrect because the language processor would interpret the /* that is part of the literal string /*123 as the start of a nested comment. It would not process the rest of the program because it would be looking for a matching comment end (*/).

You can avoid this type of problem by using concatenation for literal strings containing /* or */; line 2 would be:

```
if substr(input,1,5) = '/' || '*123'
```

You could comment out lines 2 and 3 correctly as follows:

```
01    parse pull input
02 /* if substr(input,1,5) = '/' || '*123'
03      then call process
04 */ dept = substr(input,32,5)
```

For information about Double-Byte Character Set characters, see Appendix C, "Double-Byte Character Set (DBCS) Support" on page 449 and the OPTIONS instruction on page 178.

12.1.3 Tokens

A token is the unit of low-level syntax from which clauses are built. Programs written in REXX are composed of tokens. They are separated by blanks or comments or by the nature of the tokens themselves. The classes of tokens are:

Literal Strings:

A literal string is a sequence including *any* characters and delimited by the single quotation mark (') or the double quotation mark ("). Use two consecutive double quotation marks ("") to represent a " character within a string delimited by double quotation marks. Similarly, use two consecutive single quotation marks ('') to represent a ' character within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed.

A literal string with no characters (that is, a string of length 0) is called a **null string**.

These are valid strings:

```
'Fred'
"Don't Panic!"
'You shouldn''t'          /* Same as "You shouldn't" */
''                        /* The null string         */
```

Note that a string followed immediately by a (is considered to be the name of a function. If followed immediately by the symbol X or x, it is considered to be a hexadecimal string. If followed immediately by the symbol B or b, it is considered to be a binary string. Descriptions of these forms follow. **Implementation maximum:** A literal string can contain up to 250 characters. But note that the length of computed results is limited only by the amount of storage available. See the note on page 129 for more information.

Hexadecimal Strings:

A hexadecimal string is a literal string, expressed using a hexadecimal notation of its encoding. It is any sequence of zero or more hexadecimal digits (0–9, a–f, A–F), grouped in pairs. A single leading 0 is assumed, if necessary, at the front of the string to make an even number of hexadecimal digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by single or double quotation marks, and immediately followed by the symbol X or x. (Neither x nor X can be part of a longer symbol.) The blanks, which may be present only at byte boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them. A hexadecimal string is a literal string formed by packing the hexadecimal digits given. Packing the hexadecimal digits removes blanks and converts each pair of hexadecimal digits into its equivalent character, for example: 'C1'X to A.

Hexadecimal strings let you include characters in a program even if you cannot directly enter the characters themselves. These are valid hexadecimal strings:

```
'ABCD'x
"1d ec f8"X
"1 d8"x
```

Note: A hexadecimal string is *not* a representation of a number. Rather, it is an escape mechanism that lets a user describe a character in terms of its encoding (and, therefore, is machine-dependent). In EBCDIC, '40'X is the encoding for a blank. In every case, a string of the form '.....'x is simply an alternative to a straightforward string. In EBCDIC 'C1'x and 'A' are identical, as are '40'x and a blank, and must be treated identically. Also note that in Assembler language hexadecimal numbers are represented with the X in front of the number. REXX only accepts hexadecimal numbers as was described in the previous note. In this book you will see hexadecimal numbers represented in both ways, but when you are coding a hexadecimal string in REXX, place the X after the number.

Implementation maximum: The packed length of a hexadecimal string (the string with blanks removed) cannot exceed 250 bytes.

Binary Strings:

A binary string is a literal string, expressed using a binary representation of its encoding. It is any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles). The first group may have fewer

than four digits; in this case, up to three 0 digits are assumed to the left of the first digit, making a total of four digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by matching single or double quotation marks and immediately followed by the symbol b or B. (Neither b nor B can be part of a longer symbol.) The blanks, which may be present only at byte or nibble boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

A binary string is a literal string formed by packing the binary digits given. If the number of binary digits is not a multiple of eight, leading zeros are added on the left to make a multiple of eight before packing. Binary strings allow you to specify characters explicitly, bit by bit.

These are valid binary strings:

```
'11110000'b        /* == 'f0'x                 */
"101 1101"b        /* == '5d'x                 */
'1'b               /* == '00000001'b and '01'x */
'10000 10101010'b  /* == '0001 0000 1010 1010'b */
''b                /* == ''                    */
```

Implementation maximum: The packed length of a hexadecimal string (the string with blanks removed) cannot exceed 250 bytes.

Symbols:

Symbols are groups of characters, selected from the:

- English alphabetic characters (A–Z and a–z[2])
- Numeric characters (0–9)
- Characters . ![3] ? and _ (underscore).
- Double-Byte Character Set (DBCS) characters (X'41'–X'FE')—ETMODE must be in effect for these characters to be valid in symbols.

Any lowercase alphabetic character in a symbol is translated to uppercase (that is, lowercase a–z to uppercase A–Z) before use.

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
<.H.E.L.L.O>               /* This is DBCS */
```

For information about Double-Byte Character Set (DBCS) characters, see Appendix C, "Double-Byte Character Set (DBCS) Support" on page 449.

If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned it a value, its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a–z to uppercase A–Z). Symbols that begin with a number or a period are constant symbols and cannot be assigned a value.

---

[2] Note that some code pages do not include lowercase English characters a–z.

[3] The encoding of the exclamation point character depends on the code page in use.

One other form of symbol is allowed to support the representation of numbers in exponential format. The symbol starts with a digit (0–9) or a period, and it may end with the sequence E or e, followed immediately by an optional sign (- or +), followed immediately by one or more digits (which cannot be followed by any other symbol characters). The sign in this context is part of the symbol and is not an operator.

These are valid numbers in exponential notation:

```
17.3E-12
.03e+9
```

Implementation maximum: A symbol can consist of up to 250 characters. But note that its value, if it is a variable, is limited only by the amount of storage available. See the note on page 129 for more information.

Numbers:

These are character strings consisting of one or more decimal digits, with an optional prefix of a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 255-264 for a full definition of numbers.

Numbers can have leading blanks (before and after the sign, if any) and can have trailing blanks. Blanks may not be embedded among the digits of a number or in the exponential part. Note that a symbol (see preceding) or a literal string may be a number. A number cannot be the name of a variable.

These are valid numbers:

```
12
'-17.9'
127.0650
73e+128
' + 7.9E5 '
'0E000'
```

You can specify numbers with or without quotation marks around them. Note that the sequence −17.9 (without quotation marks) in an expression is not simply a number. It is a minus operator (which may be prefix minus if no term is to the left of it) followed by a positive number. The result of the operation is a number.

A **whole number** is a number that has a zero (or no) decimal part and that the language processor would not usually express in exponential notation. That is, it has no more digits before the decimal point than the current setting of NUMERIC DIGITS (the default is 9).

Implementation maximum: The exponent of a number expressed in exponential notation can have up to nine digits.

Operator Characters:

The characters: + - \ / % * | & = ¬ > < and the sequences >= <= \> \< \= >< <> == \== // && || ** ¬> ¬< ¬= ¬== >> << >>= \<< ¬<< \>> ¬>> <<= /= /== indicate operations (see page 137). A few of these are also used in parsing templates, and the equal sign is also used to indicate assignment. Blanks adjacent to operator characters are removed. Therefore, the following are identical in meaning:

```
345>=123
345 >=123
345 >= 123
345 > = 123
```

Some of these characters may not be available in all character sets, and, if this is the case, appropriate translations may be used. In particular, the vertical bar (|) **or** character is often shown as a split vertical bar (¦).

Throughout the language, the **not** character, ¬, is synonymous with the backslash (\). You can use the two characters interchangeably according to availability and personal preference.

Special Characters:

The following characters, together with the individual characters from the operators, have special significance when found outside of literal strings:

```
,  ;  :  )  (
```

These characters constitute the set of special characters. They all act as token delimiters, and blanks adjacent to any of these are removed. There is an exception: a blank adjacent to the outside of a parenthesis is deleted only if it is also adjacent to another special character (unless the character is a parenthesis and the blank is outside it, too). For example, the language processor does not remove the blank in `A (Z)`. This is a concatenation that is not equivalent to `A(Z)`, a function call. The language processor does remove the blanks in `(A) + (Z)` because this is equivalent to `(A)+(Z)`.

The following example shows how a clause is composed of tokens.

```
'REPEAT'   A + 3;
```

This is composed of six tokens—a literal string (`'REPEAT'`), a blank operator, a symbol (`A`, which may have a value), an operator (`+`), a second symbol (`3`, which is a number and a symbol), and the clause delimiter (`;`). The blanks between the `A` and the `+` and between the `+` and the `3` are removed. However, one of the blanks between the `'REPEAT'` and the `A` remains as an operator. Thus, this clause is treated as though written:

```
'REPEAT' A+3;
```

12.1.4 Implied Semicolons

The last element in a clause is the semicolon delimiter. The language processor implies the semicolon: at a line-end, after certain keywords, and after a colon if it follows a single symbol. This means that you need to include semicolons only when there is more than one clause on a line or to end an instruction whose last character is a comma.

A line-end usually marks the end of a clause and, thus, REXX implies a semicolon at most end of lines. However, there are the following exceptions:

- The line ends in the middle of a string.
- The line ends in the middle of a comment. The clause continues on to the next line.
- The last token was the continuation character (a comma) and the line does not end in the middle of a comment. (Note that a comment is not a token.)

REXX automatically implies semicolons after colons (when following a single symbol, a label) and after certain keywords when they are in the correct context. The keywords that have this effect are: ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

Note: The two characters forming the comment delimiters, /∗ and ∗/, must not be split by a line-end (that is, / and ∗ should not appear on different lines) because they could not then be recognized correctly; an implied semicolon would be added. The two consecutive characters forming a literal quotation mark within a string are also subject to this line-end ruling.

12.1.5 Continuations

One way to continue a clause onto the next line is to use the comma, which is referred to as the **continuation character**. The comma is functionally replaced by a blank, and, thus, no semicolon is implied. One or more comments can follow the continuation character before the end of the line. The continuation character cannot be used in the middle of a string or it will be processed as part of the string itself. The same situation holds true for comments. Note that the comma remains in execution traces.

The following example shows how to use the continuation character to continue a clause.

```
say 'You can use a comma',
    'to continue this clause.'
```

This displays:

```
You can use a comma to continue this clause.
```

12.2 Expressions and Operators

Expressions in REXX are a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data.

12.2.1 Expressions

Expressions consist of one or more **terms** (literal strings, symbols, function calls, or subexpressions) interspersed with zero or more operators that denote operations to be carried out on terms. A **subexpression** is a term in an expression bracketed within a left and a right parenthesis.

Terms include:

- **Literal Strings** (delimited by quotation marks), which are constants
- **Symbols** (no quotation marks), which are translated to uppercase. A symbol that does not begin with a digit or a period may be the name of a variable; in

this case the value of that variable is used. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.

• **Function calls** (see page 201), which are of the form:

▶▶──┬─*symbol*(──┬────────────────────┬──)──▶◀
    └─*literal_string*(─┘   └─*expression*─┘
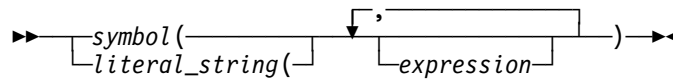
Evaluation of an expression is left to right, modified by parentheses and by operator precedence in the usual algebraic manner (see section 12.2.3, "Parentheses and Operator Precedence" on page 140). Expressions are wholly evaluated, unless an error occurs during evaluation.

All data is in the form of "typeless" character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so forth). Consequently, the result of evaluating any expression is itself a character string. Terms and results (except arithmetic and logical expressions) may be the **null string** (a string of length 0). Note that REXX imposes no restriction on the maximum length of results. However, there is a 16MB limitation on the amount of a single storage request available to the language processor. See the note on page 129 for more information.

12.2.2 Operators

An **operator** is a representation of an operation, such as addition, to be carried out on one or two terms. The following pages describe how each operator (except for the prefix operators) acts on two terms, which may be symbols, strings, function calls, intermediate results, or subexpressions. Each prefix operator acts on the term or subexpression that follows it. Blanks (and comments) adjacent to operator characters have no effect on the operator; thus, operators constructed from more than one character can have embedded blanks and comments. In addition, one or more blanks, where they occur in expressions but are not adjacent to another operator, also act as an operator. There are four types of operators:

• Concatenation
• Arithmetic
• Comparison
• Logical

12.2.2.1 String Concatenation

The concatenation operators combine two strings to form one string by appending the second string to the right-hand end of the first string. The concatenation may occur with or without an intervening blank. The concatenation operators are:

(blank)   Concatenate terms with one blank in between

‖   Concatenate without an intervening blank

(abuttal) Concatenate without an intervening blank

You can force concatenation without a blank by using the ‖ operator.

The **abuttal** operator is assumed between two terms that are not separated by another operator. This can occur when two terms are syntactically distinct, such as a literal string and a symbol, or when they are separated only by a comment.

Examples:

An example of syntactically distinct terms is: if `Fred` has the value `37.4`, then `Fred'%'` evaluates to `37.4%`.

If the variable `PETER` has the value `1`, then `(Fred)(Peter)` evaluates to `37.41`.

In EBCDIC, the two adjoining strings, one hexadecimal and one literal,

`'c1 c2'x'CDE'`

evaluate to `ABCDE`.

In the case of:

```
        Fred/* The NOT operator precedes Peter. */¬Peter
```

there is no abuttal operator implied, and the expression is not valid. However,

```
        (Fred)/* The NOT operator precedes Peter. */(¬Peter)
```

results in an abuttal, and evaluates to `37.40`.

12.2.2.2 Arithmetic

You can combine character strings that are valid numbers (see page 134) using the arithmetic operators:

+	Add
–	Subtract
*	Multiply
/	Divide
%	Integer divide (divide and return the integer part of the result)
//	Remainder (divide and return the remainder—not modulo, because the result may be negative)
**	Power (raise a number to a whole-number power)
Prefix –	Same as the subtraction: `0 - number`
Prefix +	Same as the addition: `0 + number`.

See Chapter 16, "Numbers and Arithmetic" on page 255 for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

12.2.2.3 Comparison

The comparison operators compare two terms and return the value `1` if the result of the comparison is true, or `0` otherwise.

The strict comparison operators all have one of the characters defining the operator doubled. The ==, \==, /==, and ¬== operators test for an exact match between two strings. The two strings must be identical (character by character) and of the same length to be considered strictly equal. Similarly, the strict comparison operators such as >> or << carry out a simple character-by-character comparison, with no padding of either of the strings being compared. The comparison of the two strings is from left to right. If one string is shorter than and is a leading substring of another, then it is smaller than (less than) the other. The strict comparison operators also do not attempt to perform a numeric comparison on the two operands.

For all the other comparison operators, if *both* terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so forth—see section 16.2.6, "Numeric Comparisons" on page 261) is effected. Otherwise, both terms are treated as character strings (leading and trailing blanks are ignored, and then the shorter string is padded with blanks on the right).

Character comparison and strict comparison operations are both case-sensitive, and for both the exact collating order may depend on the character set used for the implementation. For example, in an EBCDIC environment, lowercase alphabetics precede uppercase, and the digits 0–9 are higher than all alphabetics.

The comparison operators and operations are:

=	True if the terms are equal (numerically or when padded, and so forth)
\=, ¬=, /=	True if the terms are not equal (inverse of =)
>	Greater than
<	Less than
><	Greater than or less than (same as not equal)
<>	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
==	True if terms are strictly equal (identical)
\==, ¬==, /==	True if the terms are NOT strictly equal (inverse of ==)
>>	Strictly greater than
<<	Strictly less than
>>=	Strictly greater than or equal to
\<<, ¬<<	Strictly NOT less than
<<=	Strictly less than or equal to
\>>, ¬>>	Strictly NOT greater than

Note: Throughout the language, the **not** character, ¬, is synonymous with the backslash (\). You can use the two characters interchangeably, according to availability and personal preference. The backslash can appear in the following operators: \ (prefix not), \=, \==, \<, \>, \<<, and \>>.

12.2.2.4 Logical (Boolean)
A character string is taken to have the value false if it is 0, and true if it is 1. The logical operators take one or two such values (values other than 0 or 1 are not allowed) and return 0 or 1 as appropriate:

&	AND
	Returns 1 if both terms are true.

	Inclusive OR
|	Inclusive OR
	Returns `1` if either term is true.
&&	Exclusive OR
	Returns `1` if either (but not both) is true.
Prefix \, ¬	Logical NOT
	Negates; `1` becomes `0`, and `0` becomes `1`.

12.2.3 Parentheses and Operator Precedence

Expression evaluation is from left to right; parentheses and operator precedence modify this:

- When parentheses are encountered (other than those that identify function calls) the entire subexpression between the parentheses is evaluated immediately when the term is required.

- When the sequence:

  `term1 operator1 term2 operator2 term3`

  is encountered, and `operator2` has a higher precedence than `operator1`, the subexpression (`term2 operator2 term3`) is evaluated first. The same rule is applied repeatedly as necessary.

  Note, however, that individual terms are evaluated from left to right in the expression (that is, as soon as they are encountered). The precedence rules affect only the order of **operations**.

For example, `*` (multiply) has a higher priority than `+` (add), so `3+2*5` evaluates to `13` (rather than the `25` that would result if strict left to right evaluation occurred). To force the addition to occur before the multiplication, you could rewrite the expression as `(3+2)*5`. Adding the parentheses makes the first three tokens a subexpression. Similarly, the expression `-3**2` evaluates to `9` (instead of `-9`) because the prefix minus operator has a higher priority than the power operator.

The order of precedence of the operators is (highest at the top):

**+ - ¬ **	(prefix operators)
**	(power)
* / % //	(multiply and divide)
+ -	(add and subtract)
(blank) **||** (abuttal)	(concatenation with or without blank)
= > <	(comparison operators)
== >> <<	
\= ¬=	
>< <>	
\> ¬>	
\< ¬<	
\== ¬==	
\>> ¬>>	
\<< ¬<<	
>= >>=	
<= <<=	

/= /==

&   (and)

| &&   (or, exclusive or)

Examples:

Suppose the symbol A is a variable whose value is 3, DAY is a variable whose value
is Monday, and other variables are uninitialized. Then:

```
A+5                 ->    '8'
A-4*2               ->    '-5'
A/2                 ->    '1.5'
0.5**2              ->    '0.25'
(A+1)>7             ->    '0'       /* that is, False */
' '=''              ->    '1'       /* that is, True  */
' '=='''            ->    '0'       /* that is, False */
' '¬=='''           ->    '1'       /* that is, True  */
(A+1)*3=12          ->    '1'       /* that is, True  */
'077'>'11'          ->    '1'       /* that is, True  */
'077' >> '11'       ->    '0'       /* that is, False */
'abc' >> 'ab'       ->    '1'       /* that is, True  */
'abc' << 'abd'      ->    '1'       /* that is, True  */
'ab ' << 'abd'      ->    '1'       /* that is, True  */
Today is Day        ->    'TODAY IS Monday'
'If it is' day      ->    'If it is Monday'
Substr(Day,2,3)     ->    'ond'    /* Substr is a function */
'!'xxx'!'           ->    '!XXX!'
'000000' >> '0E0000' ->   '1'       /* that is, True  */
```

Note: The last example would give a different answer if the > operator had been
used rather than >>. Because '0E0000' is a valid number in exponential notation,
a numeric comparison is done; thus '0E0000' and '000000' evaluate as equal.
The REXX order of precedence usually causes no difficulty because it is the same
as in conventional algebra and other computer languages. There are two
differences from common notations:

- The prefix minus operator always has a higher priority than the power operator.

- Power operators (like other operators) are evaluated left-to-right.

For example:

```
-3**2     ==  9 /* not -9  */
-(2+1)**2 ==  9 /* not -9  */
2**2**3   == 64 /* not 256 */
```

12.3 Clauses and Instructions

Clauses can be subdivided into the following types:

12.3.1 Null Clauses

A clause consisting only of blanks or comments or both is a **null clause**. It is
completely ignored (except that if it includes a comment it is traced, if appropriate).

Note: A null clause is not an instruction; for example, putting an extra semicolon
after the THEN or ELSE in an IF instruction is not equivalent to using a dummy

instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

12.3.2 Labels

A clause that consists of a single symbol followed by a colon is a **label**. The colon in this context implies a semicolon (clause separator), so no semicolon is required. Labels identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. More than one label may precede any instruction. Labels are treated as null clauses and can be traced selectively to aid debugging.

Any number of successive clauses may be labels. This permits multiple labels before other clauses. Duplicate labels are permitted, but control passes only to the first of any duplicates in a program. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

You can use DBCS characters. See Appendix C, "Double-Byte Character Set (DBCS) Support" on page 449 for more information.

12.3.3 Instructions

An **instruction** consists of one or more clauses describing some course of action for the language processor to take. Instructions can be: assignments, keyword instructions, or commands.

12.3.4 Assignments

A single clause of the form *symbol=expression* is an instruction known as an **assignment**. An assignment gives a variable a (new) value. See section 12.4, "Assignments and Symbols" on page 143.

12.3.5 Keyword Instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Keyword instructions control the external interfaces, the flow of control, and so forth. Some keyword instructions can include nested instructions. In the following example, the DO construct (DO, the group of instructions that follow it, and its associated END keyword) is considered a single keyword instruction.

```
DO
   instruction
   instruction
   instruction
END
```

A **subkeyword** is a keyword that is reserved within the context of some particular instruction, for example, the symbols TO and WHILE in the DO instruction.

12.3.6 Commands

A **command** is a clause consisting of only an expression. The expression is evaluated and the result is passed as a command string to some external environment.

12.4 Assignments and Symbols

A **variable** is an object whose value can change during the running of a REXX program. The process of changing the value of a variable is called **assigning** a new value to it. The value of a variable is a single character string, of any length, that may contain *any* characters.

You can assign a new value to a variable with the ARG, PARSE, or PULL instructions, the VALUE built-in function, or the variable pool interface, but the most common way of changing the value of a variable is the assignment instruction itself. Any clause of the form:

symbol=expression;

is taken to be an assignment. The result of *expression* becomes the new value of the variable named by the symbol to the left of the equal sign. Currently, on VM if you omit `expression`, the variable is set to the null string. However, it is recommended that you explicitly set a variable to the null string: `symbol=''`.

Example:

```
/* Next line gives FRED the value "Frederic" */
Fred='Frederic'
```

The symbol naming the variable cannot begin with a digit (0–9) or a period. (Without this restriction on the first character of a variable name, you could redefine a number; for example 3=4; would give a variable called 3 the value 4.)

You can use a symbol in an expression even if you have not assigned it a value, because a symbol has a defined value at all times. A variable you have not assigned a value is **uninitialized**. Its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a–z to uppercase A–Z). However, if it is a compound symbol (described under section 12.4.3, "Compound Symbols" on page 144), its value is the derived name of the symbol.

Example:

```
/* If Freda has not yet been assigned a value,  */
/* then next line gives FRED the value "FREDA"   */
Fred=Freda
```

The meaning of a symbol in REXX varies according to its context. As a term in an expression (rather than a keyword of some kind, for example), a symbol belongs to one of four groups: constant symbols, simple symbols, compound symbols, and stems. Constant symbols cannot be assigned new values. You can use simple symbols for variables where the name corresponds to a single value. You can use compound symbols and stems for more complex collections of variables, such as arrays and lists.

12.4.1 Constant Symbols

A **constant symbol** starts with a digit (0–9) or a period.

You cannot change the value of a constant symbol. It is simply the string consisting of the characters of the symbol (that is, with any lowercase alphabetic characters translated to uppercase).

These are constant symbols:

```
77
827.53
.12345
12e5        /* Same as 12E5 */
3D
17E-3
```

12.4.2 Simple Symbols

A **simple symbol** does not contain any periods and does not start with a digit
(0–9).

By default, its value is the characters of the symbol (that is, translated to
uppercase). If the symbol has been assigned a value, it names a variable and its
value is the value of that variable.

These are simple symbols:

```
FRED
Whatagoodidea?    /* Same as WHATAGOODIDEA? */
?12
<.D.A.T.E>
```

12.4.3 Compound Symbols

A **compound symbol** permits the substitution of variables within its name when
you refer to it. A compound symbol contains at least one period and at least two
other characters. It cannot start with a digit or a period, and if there is only one
period in the compound symbol, it cannot be the last character.

The name begins with a **stem** (that part of the symbol up to and including the first
period). This is followed by a **tail**, parts of the name (delimited by periods) that are
constant symbols, simple symbols, or null. The **derived name** of a compound
symbol is the stem of the symbol, in uppercase, followed by the tail, in which all
simple symbols have been replaced with their values. A tail itself can be comprised
of the characters A–Z, a–z, 0–9, and @ # $ ¢ . ! ? and underscore. The value of
a tail can be any character string, including the null string and strings containing
blanks. For example:

```
taila='* ('
tailb=''
stem.taila=99
stem.tailb=stem.taila
say stem.tailb        /* Displays: 99                */
/* But the following instruction would cause an error */
/*        say stem.* (                                */
```

You cannot use constant symbols with embedded signs (for example, 12.3E+5)
after a stem; in this case, the whole symbol would not be a valid symbol.

These are compound symbols:

```
FRED.3
Array.I.J
AMESSY..One.2.
<.F.R.E.D>.<.A.B>
```

Before the symbol is used (that is, at the time of reference), the language processor substitutes the values of any simple symbols in the tail (`I`, `J`, and `One` in the examples), thus generating a new, derived name. This derived name is then used just like a simple symbol. That is, its value is by default the derived name, or (if it has been used as the target of an assignment) its value is the value of the variable named by the derived name.

The substitution into the symbol that takes place permits arbitrary indexing (subscripting) of collections of variables that have a common stem. Note that the values substituted can contain *any* characters (including periods and blanks). Substitution is done only one time.

To summarize: the derived name of a compound variable that is referred to by the symbol

`s0.s1.s2. --- .sn`

is given by

`d0.v1.v2. --- .vn`

where `d0` is the uppercase form of the symbol `s0`, and `v1` to `vn` are the values of the constant or simple symbols `s1` through `sn`. Any of the symbols `s1-sn` can be null. The values `v1-vn` can also be null and can contain *any* characters (in particular, lowercase characters are not translated to uppercase, blanks are not removed, and periods have no special significance).

Some examples follow in the form of a small extract from a REXX program:

```
a=3         /* assigns '3' to the variable A    */
z=4                /*  '4'      to Z        */
c='Fred'           /*  'Fred'   to C        */
a.z='Fred'         /*  'Fred'   to A.4      */
a.fred=5           /*  '5'      to A.FRED   */
a.c='Bill'         /*  'Bill'   to A.Fred   */
c.c=a.fred         /*  '5'      to C.Fred   */
y.a.z='Annie'      /*  'Annie'  to Y.3.4    */

say  a  z  c  a.a  a.z  a.c  c.a  a.fred y.a.4
/* displays the string:                    */
/*    "3 4 Fred A.3 Fred Bill C.3 5 Annie" */
```

You can use compound symbols to set up arrays and lists of variables in which the subscript is not necessarily numeric, thus offering great scope for the creative programmer. A useful application is to set up an array in which the subscripts are taken from the value of one or more variables, effecting a form of associative memory (content addressable).

Implementation maximum: The length of a variable name, before and after substitution, cannot exceed 250 characters.

12.4.4 Stems

A **stem** is a symbol that contains just one period, which is the last character. It cannot start with a digit or a period.

These are stems:

```
FRED.
A.
<.A.B>.
```

By default, the value of a stem is the string consisting of the characters of its symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

Further, when a stem is used as the target of an assignment, *all possible* compound variables whose names begin with that stem receive the new value, whether they previously had a value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or to the individual variable.

For example:

```
hole.  = "empty"
hole.9 = "full"

say  hole.1  hole.mouse  hole.9

/* says "empty empty full" */
```

Thus, you can give a whole collection of variables the same value. For example:

```
total. = 0
do forever
   say "Enter an amount and a name:"
   pull amount name
   if datatype(amount)='CHAR' then leave
   total.name = total.name + amount
   end
```

Note: You can always obtain the value that has been assigned to the whole collection of variables by using the stem. However, this is not the same as using a compound variable whose derived name is the same as the stem. For example:

```
total. = 0
null = ""
total.null = total.null + 5
say total. total.null             /* says "0 5" */
```

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. `DROP FRED.` drops all variables with that stem (see page 168), and `PROCEDURE EXPOSE FRED.` exposes *all possible* variables with that stem (see page 183).

Notes:

1. When the ARG, PARSE, or PULL instruction or the VALUE built-in function or the variable pool interface changes a variable, the effect is identical with an assignment. Anywhere a value can be assigned, using a stem sets an entire collection of variables.

2. Because an expression can include the operator =, and an instruction may consist purely of an expression (see section 12.5, "Commands to External Environments" on page 147), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an equal sign (=) is an **assignment**, rather than an expression (or a keyword instruction). This is not a restriction, because you can ensure the clause is processed as a command in several ways, such as by putting a null

string before the first name, or by enclosing the first part of the expression in parentheses.

Similarly, if you unintentionally use a REXX keyword as the variable name in an assignment, this should not cause confusion. For example, the clause:

```
Address='10 Downing Street';
```

is an assignment, not an ADDRESS instruction.

3. You can use the SYMBOL function (see page 228) to test whether a symbol has been assigned a value. In addition, you can set SIGNAL ON NOVALUE to trap the use of any uninitialized variables (except when they are tails in compound variables—see page 266).

12.5 Commands to External Environments

Issuing commands to the surrounding environment is an integral part of REXX.

12.5.1 Environment

The system under which REXX programs run is assumed to include at least one environment for processing commands. An environment is selected by default on entry to a REXX program. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program. The default environment for a REXX/CICS program is REXXCICS.

12.5.2 Commands

To send a command to the currently addressed environment, use a clause of the form:

```
expression;
```

The expression is evaluated, resulting in a character string (which may be the null string), which is then prepared as appropriate and submitted to the underlying system. Any part of the expression not to be evaluated should be enclosed in quotation marks.

The environment then processes the command, which may have side-effects. It eventually returns control to the language processor, after setting a return code. A **return code** is a string, typically a number, that returns some information about the command that has been processed. A return code usually indicates if a command was successful or not, but can also represent other information. The language processor places this return code in the REXX special variable RC. See section 17.4, "Special Variables" on page 269.

In addition to setting a return code, the underlying system may also indicate to the language processor if an error or failure occurred. An **error** is a condition raised by a command for which a program that uses that command would usually be expected to be prepared. (For example, a locate command to an editing system might report `requested string not found` as an error.) A **failure** is a condition raised by a command for which a program that uses that command would *not* usually be expected to recover (for example, a command that is not executable or cannot be found).

Errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is ON (see Chapter 17, "Conditions and Condition Traps" on page 265). They may also cause the command to be traced if `TRACE E` or `TRACE F` is set. `TRACE Normal` is the same as `TRACE F` and is the default—see page 194.

Here is an example of submitting a command to the default REXX/CICS command environment. The sequence:

```
TSQ1 = 'TSQUEUE1'
"EXECIO * READ" TSQ1 "MYDATA."
```

results in the string `EXECIO * READ TSQUEUE1 MYDATA.` being submitted to REXX/CICS.

On return, the return code placed in RC has the value `0` if the CICS temporary storage queue TSQUEUE1 was successfully read into MYDATA array. If TSQUEUE1 is empty the appropriate return code is placed in RC.

Note: Remember that the expression is evaluated before it is passed to the environment. Enclose in quotation marks any part of the expression that is not to be evaluated.

Example:

```
"EXECIO * READ"        /* * does not mean "multiplied by"  */
```

12.6 Basic Structure of REXX Running Under CICS

REXX/CICS support provides a main interface program named CICREXD which is used to load and issue REXX execs within a CICS region. Each REXX exec runs under a separate CICS task. Any nested REXX execs run under the CICS task of the parent exec.

12.6.1 REXX Exec Invocation

- Execs started from a terminal

  CICS uses transaction identifiers associated with programs to determine which programs to execute. REXX/CICS uses a table created by the REXX/CICS command, DEFTRNID, to associate CICS transaction identifiers with specific REXX execs. The CICS transaction identifier associated with the REXX/CICS supplied exec, CICRXTRY, is known as the default REXX/CICS transaction identifier. The supplied default is REXX. If REXX is entered from a CICS screen alone, the CICRXTRY exec is started. If other operands are specified, for example: REXX MYEXEC ABC, the exec MYEXEC is started and ABC is passed to it as an argument. CICS transaction identifiers other than the REXX/CICS default transaction identifier cause the associated exec to be started and any other operands are passed as an argument to the exec. For example: EDIT TEST.EXEC causes the REXX/CICS editor exec, CICEDIT, to start and TEST.EXEC, the argument, names the file to edit or create. All REXX/CICS transaction identifiers must have CICS definitions which associate them with the REXX/CICS main module, CICREXD.

- Execs started using a CICS START command

  REXX/CICS execs may be started using the CICS START command. The START command names the CICS transaction identified to start and the table created by the REXX/CICS DEFTRNID command names the exec to start. If

the exec is CICRXTRY and CICS start data is present, the first operand names the exec to start and any other operands are passed to the exec as an argument. If no start data is present, then CICRXTRY is started. For execs other than CICRXTRY, start data is passed to the exec as an argument. Usually, REXX/CICS execs have a terminal associated with them. However, if a REXX/CICS exec does not have a terminal associated to the transaction, any terminal output is either discarded or directed to a CICS temporary storage queue as specified by the REXX/CICS SET TERMOUT command. An error is generated if terminal input is requested for a transaction for which no terminal is associated.

- Execs started using a CICS LINK or XCTL command

  The REXX/CICS interface program CICREXD may be invoked using the CICS LINK or XCTL command. A COMMAREA must be passed to the interface program when invoked in this way. The COMMAREA must contain operands that name the exec to start and an argument to be passed. The exec name is expected as the first blank delimited token in the COMMAREA.

Note: A utility called REXXTRY (CICRXTRY exec) is provided with REXX/CICS that allows the interactive execution of REXX instructions and REXX/CICS commands. To invoke this utility, enter the REXX/CICS transaction identifier associated with CICRXTRY without any operands.

12.6.2 Where Execs Execute

REXX/CICS execs are executed as part of the CICS task that issues them, within the CICS region. The REXX interpreter is fully reentrant and runs above the 16 megabyte line (AMODE=31, RMODE=ANY).

12.6.3 Locating and Loading Execs

The following rules are used to locate and load an exec into storage so it can be started:

1. Execs loaded into storage using EXECLOAD are searched and if the exec is found, the EXECLOADed copy is used.

2. The user's current RFS directory, as defined by the REXX/CICS CD command, is searched. If found, the exec is loaded and started.

3. The user's path, as defined by the REXX/CICS PATH command, is searched. If found, the exec is loaded and started.

4. If the user is an authorized user, VSE Librarian sublibraries specified on the SETSYS AUTHCLIB command are searched. If the exec is found, it is loaded and started.

5. VSE Librarian sublibraries specified on the SETSYS AUTHELIB command are searched. If the exec is found, it is loaded and started.

6. Finally, members with a member type of PROC in sublibraries defined in the LIBDEF PROC search chain for the CICS TS partition are searched.

12.6.4 Editing Execs

REXX execs can be edited using the supplied REXX/CICS editor. Also, if REXX/CICS execs reside in a VSE Librarian member, they can be edited using other editors such as the librarian member edit component of DITTO.

12.6.5 REXX File System

Execs can be stored as members in the VSAM-based REXX File System (RFS), provided with REXX/CICS, or in VSE librarian members with a member type of PROC.

Note: If the file identifier you specified for invoking an exec does not include a file type extension, then only RFS file identifiers with a file type of EXEC are searched when you attempt to locate and issue the REXX exec. If the file identifier includes a file type extension, then only RFS files with a matching file type are searched (when you attempt to locate and run the exec).

12.6.6 Control of Exec Execution Search Order

The CD and PATH commands define the search order of user REXX libraries when you attempt to load an exec. After the current directory (set by the CD command) is searched, all directories specified with the PATH command are searched, then VSE sublibraries specified on the SETSYS AUTHCLIB and SETSYS AUTHELIB commands are searched, followed by VSE Librarian members with a member type of PROC in the LIBDEF PROC search chain for the CICS partition.

12.6.7 Adding User Written Commands

An exec may be defined so it can be invoked as a REXX/CICS command. The DEFCMD command is used to define REXX/CICS user commands. DEFCMD supports commands written in REXX as well as the standard CICS supported languages. For more information on the DEFCMD command, see section 25.10, "DEFCMD" on page 383.

12.7 Support of Standard REXX Features

This section discusses the support of standard REXX features such as SAY and TRACE statements, PULL and PARSE EXTERNAL statements, REXX stacking, and REXX functions.

12.7.1 SAY and TRACE Statements

The REXX SAY and TRACE terminal I/O output statements use CICS Terminal Control support to provide simulated line-mode output. Also, the SET TERMOUT command can be used to route line-mode output into a temporary storage queue (see page 421).

12.7.2 PULL and PARSE EXTERNAL Statements

The REXX PULL and PARSE EXTERNAL terminal I/O input statements use CICS Terminal Control support to provide simulated line-mode input.

Notes:

1. PULL (or PARSE PULL) first attempts to pull a line from the program stack and, only if it is empty, issues a read to the terminal.

2. Attempting to perform terminal line-mode input from a REXX exec that is running as part of a non-terminal attached transaction, is an error, which causes the exec to terminate with an error message.

12.7.3 REXX Stack Support

Each user has a shared program stack between multiple generations of REXX execs. This single automatic program stack is not named. If named program stacks are desired, use the RLS LPUSH, LQUEUE, and LPULL commands.

12.7.4 REXX Function Support

REXX/CICS supports the standard SAA Level 2 built-in function set, with the following exceptions:

- Stream I/O functions are not supported.

- The USERID function returns a 1 to 8 character CICS user ID if the user is signed on. If the CICS user has not signed on and a default user has been specified for the CICS region (by the CICS Systems Programmer specifying DFLTUSER in the CICS startup parameters) then that value is used.

  Note: The default user shares REXX File System and REXX List System directories.

- The STORAGE function, that allows a REXX user to display or modify the virtual storage of the CICS region. This function can only be successfully invoked from an authorized exec or by an authorized user.

12.8 REXX Command Environment Support

REXX command environments that are currently available (to use with the REXX ADDRESS command) are REXXCICS, CICS, EXECSQL, EDITSVR, FLSTSVR, RFS and RLS.

12.8.1 Adding REXX Host Command Environments

Support is provided for allowing new REXX/CICS commands and command environments to be dynamically defined. New commands may be written in REXX or in any REXX/CICS supported language (for example: Assembler, COBOL, C, PL/I). For more information on how commands and command environments are defined to REXX/CICS, see Chapter 21, "REXX/CICS Command Definition" on page 325.

12.9 Support of Standard CICS Features/Facilities

This section discusses the support of standard CICS features/facilities such as: CICS mapped I/O support, dataset I/O services, interfaces to CICS facilities and services, issuing user applications from execs, REXX interfaces to CICS temporary and transient storage queues, pseudo-conversational transaction support, and DBCS support.

12.9.1 CICS Mapped I/O Support

Support for CICS basic mapping support (BMS) I/O is provided by the CICS SEND MAP, RECEIVE MAP, and CONVERSE MAP commands and the REXX/CICS CONVTMAP and COPYS2R commands. See Appendix J, "Basic Mapping Support Example" on page 487.

Note: BMS maps must be predefined, using normal CICS procedures.

12.9.2 Dataset I/O Services

Standard CICS File I/O commands (EXEC CICS READ, WRITE, and so on) are supported. Also, high-level I/O may be done from an exec to the VSAM-based REXX File System (RFS) using the provided RFS command. In addition, standard VSE Librarian members may be used with the IMPORT and EXPORT commands and the REXX/CICS editor.

12.9.3 Interfaces to CICS Facilities and Services

From within the ADDRESS CICS command environment, support is provided for most CICS commands (as defined in the *CICS Transaction Server for VSE/ESA Application Programming Reference*. See Chapter 25, "REXX/CICS Commands" on page 369 for detailed information on the commands supported.

12.9.4 Issuing User Applications From Execs

REXX/CICS supports the EXEC CICS START, LINK, and XCTL commands to provide the ability to START CICS transactions or invoke CICS programs from within a REXX exec.

12.9.5 REXX Interfaces to CICS Storage Queues

Command support exists for reading, writing, and deleting CICS temporary storage and transient data queues from REXX/CICS.

12.9.6 Pseudo-conversational Transaction Support

CICS pseudo-conversational support for REXX execs is supported though the use of the CICS RETURN TRANSID() command, by the REXX/CICS PSEUDO command (see section 25.32, "PSEUDO" on page 415), and the SETSYS PSEUDO command (see section 25.37, "SETSYS" on page 423).

12.10 Interfaces to Other Programming Languages

REXX/CICS supports (by support for CICS LINK and CICS XCTL REXX/CICS commands) the ability to invoke CICS programs written in any REXX/CICS supported language. It likewise provides the same support for the programs used to implement new REXX commands, which are defined by using the DEFCMD and DEFSCMD commands. Support is also provided to allow an EXEC CICS START to be issued from REXX execs.

12.10.1 DBCS Support

The full range of DBCS functions and handling techniques that are included in SAA Level 2 REXX are available to the REXX/CICS user.

12.10.2 Miscellaneous Features

- A TERMID command has been provided to return the four character terminal identifier of a CICS user.

- A retrieve PF key may be specified to retrieve the last input line entered using line-mode I/O while in the REXXTRY interactive utility (CICRXTRY exec). Refer to the SET RETRIEVE command.

- The SET TERMOUT command allows line-mode terminal output (from SAY or TRACE) to be directed to a CICS temporary storage queue instead of, or in addition to, the terminal.

- The PULL instruction sets the REXX variable PULLKEY, with the name of the AID key pressed if PULL read data from the terminal.

- Line-mode output to the terminal causes `MORE` to appear in the lower right hand corner of the screen, when the screen is full. Press `Clear` or `Enter` to proceed.

- Line-mode input from the terminal causes `READ` to appear in the lower right hand corner of the screen.

Chapter 13. Keyword Instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. Some keyword instructions, like DO, can include nested instructions.

In the syntax diagrams on the following pages, symbols (words) in capitals denote keywords or subkeywords; other words (such as *expression*) denote a collection of tokens as defined previously. Note, however, that the keywords and subkeywords are not case dependent; the symbols if, If, and iF all have the same effect. Note also that you can usually omit most of the clause delimiters (;) shown because they are implied by the end of a line.

As explained in section 12.3.5, "Keyword Instructions" on page 142, a keyword instruction is recognized *only* if its keyword is the first token in a clause, and if the second token does not start with an = character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. Therefore,

```
arg(fred) rest
```

is an ARG keyword instruction, not a command that starts with a call to the ARG built-in function. A syntax error results if the keywords are not in their correct positions in a DO, IF, or SELECT instruction. (The keyword THEN is also recognized in the body of an IF or WHEN clause.) In other contexts, keywords are not reserved and can be used as labels or as the names of variables (though this is generally not recommended).

Certain other keywords, known as subkeywords, are reserved within the clauses of individual instructions. For example, the symbols VALUE and WITH are subkeywords in the ADDRESS and PARSE instructions, respectively. For details, see the description of each instruction. For a general discussion on reserved keywords, see page 467.

Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the *expression* from the subkeyword in the example following:

```
ADDRESS VALUE expression
```

However, no blank is required after the VALUE subkeyword in the following example, although it would add to the readability:

```
ADDRESS VALUE'ENVIR'||number
```

13.1 ADDRESS

```
►►──ADDRESS───┬─────────────────────────────────┬───;──────────────────►◄
              │  ┌─environment─┐                 │
              ├──┤             ├──┬───────────┐──┤
              │  │             └──└─expression─┘  │
              │  └─────────────────expression1────┤
              └──────VALUE──────────────────────────┘
```
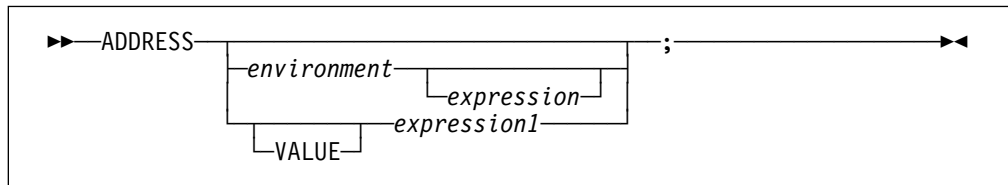
ADDRESS temporarily or permanently changes the destination of commands. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction.

The concept of alternative subcommand environments is described in section 8.2, "Issuing Commands from a Program" on page 96.

To send a single command to a specified environment, code an *environment*, a literal string or a single symbol, which is taken to be a constant, followed by an *expression*. (The environment name is the name of an external procedure or process that can process commands.) The environment name is limited to eight characters. The *expression* is evaluated, and the resulting string is routed to the *environment* to be processed as a command. (Enclose in quotation marks any part of the expression you do not want to be evaluated.) After execution of the command, *environment* is set back to whatever it was before, thus temporarily changing the destination for a single command. The special variable RC is set, just as it would be for other commands. (See page 147.) Errors and failures in commands processed in this way are trapped or traced as usual.

Example:
```
ADDRESS CICS "READQ TSQ QUEUE('QUEUE1') INTO(VAR1)" /*  CICS  */
```

If you specify only *environment*, a lasting change of destination occurs: all commands that follow (clauses that are neither REXX instructions nor assignment instructions) are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Example:
```
address cics
"READQ TSQ QUEUE('QUEUE1') INTO(VAR1)"
ADDRESS RFS
'COPY PROFILE.EXEC TEMP.EXEC'
```

Similarly, you can use the VALUE form to make a lasting change to the environment. Here *expression1* (which may be simply a variable name) is evaluated, and the result forms the name of the environment. You can omit the subkeyword VALUE if *expression1* does not begin with a literal string or symbol (that is, if it starts with a special character, such as an operator character or parenthesis).

Example:
```
ADDRESS ('ENVIR'||number)  /* Same as ADDRESS VALUE 'ENVIR'||number */
```

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. After changing the environment, repeated execution of ADDRESS alone, therefore, switches the command destination between two environments alternately.

The two environment names are automatically saved across internal and external subroutine and function calls. See the CALL instruction (page 160) for more details.

The address setting is the currently selected environment name. You can retrieve the current address setting by using the ADDRESS built-in function (see page 206).

13.2 ARG

```
►►──ARG──────────────────;─────────────────────────►◄
         └─template_list─┘
```
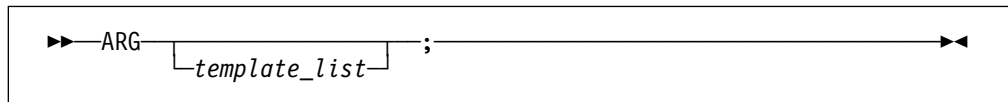
ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is a short form of the instruction:

```
►►──PARSE UPPER ARG──────────────────;─►◄
                    └─template_list─┘
```
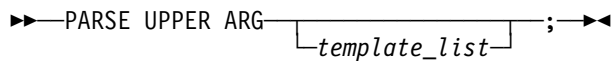
The *template_list* is often a single template but can be several templates separated by commas. If specified, each template is a list of symbols separated by blanks or patterns or both.

Unless a subroutine or internal function is being processed, the strings passed as parameters to the program are parsed into variables according to the rules described in the section on parsing (page 239).

If a subroutine or internal function is being processed, the data used will be the argument strings that the caller passes to the routine.

In either case, the language processor translates the passed strings to uppercase (that is, lowercase a–z to uppercase A–Z) before processing them. Use the PARSE ARG instruction if you do not want uppercase translation.

You can use the ARG and PARSE ARG instructions repeatedly on the same source string or strings (typically with different templates). The source string does not change. The only restrictions on the length or content of the data parsed are those the caller imposes.

Example:

```
/* String passed is "Easy Rider"  */

Arg adjective noun .

/* Now:  ADJECTIVE  contains 'EASY'        */
/*       NOUN       contains 'RIDER'       */
```

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

Example:

```
/* Function is called by  FRED('data X',1,5)   */

Fred:  Arg string, num1, num2

/* Now:   STRING  contains 'DATA X'        */
/*        NUM1    contains '1'             */
/*        NUM2    contains '5'             */
```

Notes:

1. The ARG built-in function can also retrieve or check the argument strings to a REXX program or internal routine, see page 207.

2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) on page 181 for details.

13.3 CALL

```
►►──CALL──┬─name──┬──────────────┬─────────────────────┬──;──────────────►◄
          │       │     ┌──,──┐   │                     │
          │       └──▼──┴──────────────┘                │
          │              └─expression─┘                 │
          │                                             │
          ├──OFF──┬─ERROR───┬───────────────────────────┤
          │       ├─FAILURE─┤                           │
          │       └─HALT────┘                           │
          │                                             │
          └──ON───┬─ERROR───┬──────────────────────────┘
                  ├─FAILURE─┤      ┌─NAME──trapname─┐
                  └─HALT────┘      └────────────────┘
```
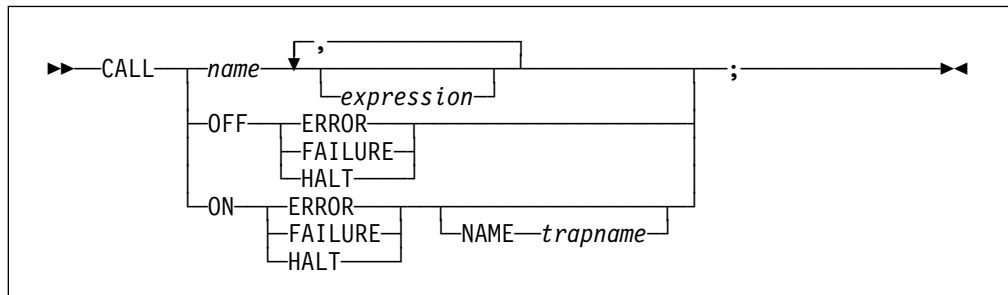
CALL calls a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 17, "Conditions and Condition Traps" on page 265.

To call a routine, a literal string or symbol that is taken as a constant, specify *name*. The *name* must be a symbol, which is treated literally, or a literal string. The routine called can be:

An internal routine   A function or subroutine that is in the same program as the CALL instruction or function call that calls it.

A built-in routine   A function (which may be called as a subroutine) that is defined as part of the REXX language.

An external routine   A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that calls it.

If *name* is a string (that is, you specify *name* in quotation marks), the search for internal routines is bypassed, and only a built-in function or an external routine is called. Note that the names of built-in functions (and generally the names of external routines, too) are in uppercase; therefore the name in the literal string should also be in uppercase.

The called routine can optionally return a result, and when it does, the CALL instruction is functionally identical with the clause:

```
►►──result=name(──┬──────────────┬──)──;──►◄
                  │    ┌──,──┐    │
                  └──▼──┴──────────┘
                         └─expression─┘
```
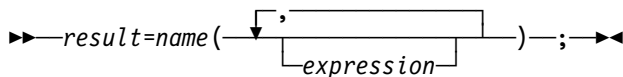
If the called routine does not return a result, you get an error if you call it as a function (as previously shown).

VM supports specifying up to 20 expressions, separated by commas. The *expression*s are evaluated in order from left to right and form the argument strings during execution of the routine. Any ARG or PARSE ARG instruction or ARG built-in function in the called routine accesses these strings rather than any previously active in the calling program, until control returns to the CALL instruction. You can omit expressions, if appropriate, by including extra commas.

The CALL then causes a branch to the routine called *name*, using exactly the same mechanism as function calls, see Chapter 14, "Functions" on page 201. The search order is in the section on functions (see section 14.2.1, "Search Order" on page 203) but briefly is as follows:

Internal routines:

These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. You can use SIGNAL and CALL together to call an internal routine whose name was determined at the time of execution; this is known as a multi-way call (see page 193). The RETURN instruction completes the execution of an internal routine.

Built-in routines:

These are routines built into the language processor for providing various functions. They always return a string that is the result of the routine. (See page 205.)

External routines:

Users can write or use routines that are external to the language processor and the calling program. External routines must be coded in REXX. If the CALL instruction calls an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG or PARSE ARG instructions or the ARG built-in function.

During execution of an internal routine, all variables previously known are generally accessible. However, the PROCEDURE instruction can set up a local variables environment to protect the subroutine and caller from each other. The EXPOSE option on the PROCEDURE instruction can expose selected variables to a routine.

Calling an external program as a subroutine is similar to calling an internal routine. The external routine, however, is an implicit PROCEDURE in that all the caller's variables are always hidden. The status of internal values (NUMERIC settings, and so forth) start with their defaults (rather than inheriting those of the caller). In addition, you can use EXIT to return from the routine.

When control reaches an internal routine, the line number of the CALL instruction is available in the variable SIGL (in the caller's variable environment). This may be used as a debug aid, as it is, therefore, possible to find out how control reached a routine. Note that if the internal routine uses the PROCEDURE instruction, then it needs to EXPOSE SIGL to get access to the line number of the CALL.

Eventually the subroutine should process a RETURN instruction, and at that point control returns to the clause following the original CALL. If the RETURN instruction specified an expression, the variable RESULT is set to the value of that expression. Otherwise, the variable RESULT is dropped (becomes uninitialized).

An internal routine can include calls to other internal routines, as well as recursive calls to itself.

Example:

```
/* Recursive subroutine execution... */
arg z
call factorial z
say z'! =' result
exit

factorial: procedure     /* Calculate factorial by  */
  arg n                  /*  recursive invocation.  */
  if n=0 then return 1
  call factorial n-1
  return  result * n
```

During internal subroutine (and function) execution, all important pieces of information are automatically saved and then restored upon return from the routine. These are:

- **The status of DO loops and other structures**: Executing a SIGNAL while within a subroutine is safe because DO loops, and so forth, that were active when the subroutine was called are not ended. (But those currently active within the subroutine are ended.)

- **Trace action**: After a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this does not affect the tracing of the caller. Conversely, if you simply wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) and ! (command inhibition) are saved across routines.

- **NUMERIC settings**: The DIGITS, FUZZ, and FORM of arithmetic operations (in section 13.12, "NUMERIC" on page 176) are saved and then restored on return. A subroutine can, therefore, set the precision, and so forth, that it needs to use without affecting the caller.

- **ADDRESS settings**: The current and previous destinations for commands (see section 13.1, "ADDRESS" on page 156) are saved and then restored on return.

- **Condition traps**: (CALL ON and SIGNAL ON) are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.

- **Condition information**: This information describes the state and origin of the current trapped condition. The CONDITION built-in function returns this information. See section section 14.3.11, "CONDITION" on page 210.

- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see section 14.3.52, "TIME" on page 228), but because the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.

- **OPTIONS settings**: ETMODE and EXMODE are saved and then restored on return. For more information, see section 13.13, "OPTIONS" on page 178.

Implementation maximum: The total nesting of control structures, which includes internal routine calls, is dependent upon available storage.

13.4 DO

```
►►──DO──┬──────────────┬──┬──────────────┬──;────────────────────►
         └─┤ repetitor ├─┘  └─┤ conditional ├─┘

►──┬───────────────────────┬──END──┬────────┬──;────────────────►◄
   │  ┌──────────────────┐  │       └─ name ─┘
   └──┴─── instruction ───┴─┘
```

repetitor:
```
├──┬─ name=expri ──┬───────────┬──┬───────────┬──┬────────────┬──┤
│                  └─TO─ exprt ─┘  └─BY─ exprb ─┘  └─FOR─ exprf ─┘
├──FOREVER──────────────────────────────────────────────────────
└── exprr ───────────────────────────────────────────────────────
```

conditional:
```
├──┬─WHILE─ exprw ─┬─────────────────────────────────────────────┤
   └─UNTIL─ expru ─┘
```
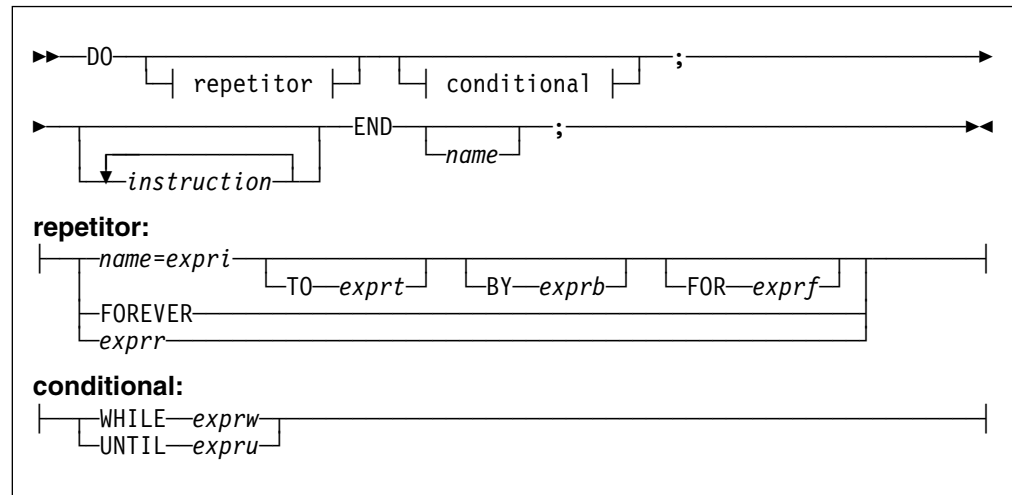
DO groups instructions together and optionally processes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

Syntax Notes:

- The *exprr*, *expri*, *exprb*, *exprt*, and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a positive whole number or zero. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.

- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.

- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.

- The *instruction* can be any instruction, including assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).

- The subkeywords WHILE and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. Similarly, TO, BY, and FOR cannot be used in *expri*, *exprt*, *exprb*, or *exprf*. FOREVER is also reserved, but only if it immediately follows the keyword DO and an equal sign does not follow it.

- The *exprb* option defaults to 1, if relevant.

13.4.1 Simple DO Group

If you specify neither *repetitor* nor *conditional*, the construct merely groups a number of instructions together. These are processed one time.

In the following example, the instructions are processed one time.

Example:

```
                      /* The two instructions between DO and END are both  */
                      /* processed if A has the value "3".                 */
                      If a=3 then Do
                              a=a+2
                              Say 'Smile!'
                            End
```

13.4.2 Repetitive DO Loops

If a DO instruction has a repetitor phrase or a conditional phrase or both, the group of instructions forms a **repetitive DO loop**. The instructions are processed according to the repetitor phrase, optionally modified by the conditional phrase. (See section 13.4.3, "Conditional Phrases (WHILE and UNTIL)" on page 166).

13.4.2.1 Simple Repetitive Loops

A simple repetitive loop is a repetitive DO loop in which the repetitor phrase is an expression that evaluates to a count of the iterations.

If *repetitor* is omitted but there is a *conditional* or if the *repetitor* is FOREVER, the group of instructions is nominally processed "forever," that is, until the condition is satisfied or a REXX instruction is processed that ends the loop (for example, LEAVE).

Note: For a discussion on conditional phrases, see section 13.4.3, "Conditional Phrases (WHILE and UNTIL)" on page 166.

In the simple form of a repetitive loop, *exprr* is evaluated immediately (and must result in a positive whole number or zero), and the loop is then processed that many times.

Example:

```
/* This displays "Hello" five times */
Do 5
  say 'Hello'
  end
```

Note that, similar to the distinction between a command and an assignment, if the first token of *exprr* is a symbol and the second token is (or starts with) =, the controlled form of *repetitor* is expected.

13.4.2.2 Controlled Repetitive Loops

The controlled form specifies *name*, a **control variable** that is assigned an initial value (the result of *expri*, formatted as though 0 had been added) before the first execution of the instruction list. The variable is then stepped (by adding the result of *exprb*) before the second and subsequent times that the instruction list is processed.

The instruction list is processed repeatedly while the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or 0, the loop is ended when *name* is greater than *exprt*. If negative, the loop is ended when *name* is less than *exprt*.

The *expri*, *exprt*, and *exprb* options must result in numbers. They are evaluated only one time, before the loop begins and before the control variable is set to its initial value. The default value for *exprb* is 1. If *exprt* is omitted, the loop runs indefinitely unless some other condition stops it.

Example:

```
Do I=3 to -2 by -1        /* Displays:   */
   say i                  /*      3      */
   end                    /*      2      */
                          /*      1      */
                          /*      0      */
                          /*     -1      */
                          /*     -2      */
```

The numbers do not have to be whole numbers:

Example:

```
I=0.3                     /* Displays:   */
Do Y=I to I+4 by 0.7      /*     0.3     */
   say Y                  /*     1.0     */
   end                    /*     1.7     */
                          /*     2.4     */
                          /*     3.1     */
                          /*     3.8     */
```

The control variable can be altered within the loop, and this may affect the iteration of the loop. Altering the value of the control variable is not usually considered good programming practice, though it may be appropriate in certain circumstances.

Note that the end condition is tested at the start of each iteration (and after the control variable is stepped, on the second and subsequent iterations). Therefore, if the end condition is met immediately, the group of instructions can be skipped entirely. Note also that the control variable is referred to by name. If (for example) the compound name A.I is used for the control variable, altering I within the loop causes a change in the control variable.

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, you must specify *exprf*, and it must evaluate to a positive whole number or zero. This acts just like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition stops it. Like the TO and BY expressions, it is evaluated only one time—when the DO instruction is first processed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

```
Do Y=0.3 to 4.3 by 0.7 for 3  /* Displays:   */
   say Y                      /*     0.3     */
   end                        /*     1.0     */
                              /*     1.7     */
```

In a controlled loop, the *name* describing the control variable can be specified on the END clause. This *name* must match *name* in the DO clause in all respects except case (note that no substitution for compound variables is carried out); a syntax error results if it does not. This enables the nesting of loops to be checked automatically, with minimal overhead.

Example:

```
Do K=1 to 10
  ...
  ...
  End k  /* Checks that this is the END for K loop */
```

> **Note:** The NUMERIC settings may affect the successive values of the control variable, because REXX arithmetic rules apply to the computation of stepping the control variable.

13.4.3 Conditional Phrases (WHILE and UNTIL)

A conditional phrase can modify the iteration of a repetitive DO loop. It may cause the termination of a loop. It can follow any of the forms of *repetitor* (none, FOREVER, simple, or controlled). If you specify WHILE or UNTIL, *exprw* or *expru*, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either 0 or 1), and the loop is ended if *exprw* evaluates to 0 or *expru* evaluates to 1.

For a WHILE loop, the condition is evaluated at the top of the group of instructions. For an UNTIL loop, the condition is evaluated at the bottom—before the control variable has been stepped.

Example:

```
Do I=1 to 10 by 2 until i>6
  say i
  end
/* Displays: "1" "3" "5" "7" */
```

Note: Using the LEAVE or ITERATE instructions can also modify the execution of repetitive loops.

```
┌─────────────────────────────────┐
│ Evaluate exprr + 0      or       │
│ evaluate expri + 0      and      │
│ exprt + 0, exprb + 0,   and      │
│ exprf + 0       in order written │
└─────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────┐
│ Assign start value to control   │
│ variable                         │
└─────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────┐      Discontinue execution of DO
│ Use TO value (expr) to test     │────▶ group if TO value is exceeded
│ control variable for termination │
└─────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────┐      Discontinue execution of DO
│ Use count of iterations (exprr) │────▶ group if number of iterations
│ to test for termination          │      is exceeded
└─────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────┐      Discontinue execution of DO
│ Use FOR value (exprf) to test   │────▶ group if FOR value (number of
│ for termination                  │      iterations through the loop) is
└─────────────────────────────────┘      exceeded
              │
              ▼
┌─────────────────────────────────┐      Discontinue execution of DO
│ Use WHILE expression (exprw)    │────▶ group if WHILE condition is
│ to test for termination          │      not met
└─────────────────────────────────┘
              │
              ▼
        ╭───────────────╮
       (  Execute instruction(s) )
        (  in the DO group        )
        ╰───────────────╯
              │
              ▼
┌─────────────────────────────────┐      Discontinue execution of DO
│ Use UNTIL expression (expru)    │────▶ group if UNTIL condition is
│ to test for termination          │      met
└─────────────────────────────────┘
              │
              ▼
┌─────────────────────────────────┐
│ Use BY expression (exprb) to    │
│ update control variable          │
└─────────────────────────────────┘
```

Figure 49. Concept of a DO Loop

13.5 DROP

```
▶▶──DROP──┬─name───┬──;─────────────────────────────────────────▶◀
          └─(name)─┘
```
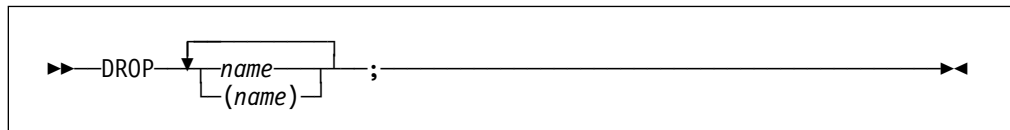
DROP "unassigns" variables, that is, restores them to their original uninitialized
state. If *name* is not enclosed in parentheses, it identifies a variable you want to
drop and must be a symbol that is a valid variable name, separated from any other
name by one or more blanks or comments.

If parentheses enclose a single *name*, then its value is used as a subsidiary list of
variables to drop. (Blanks are not necessary either inside or outside the
parentheses, but you can add them if desired.) This subsidiary list must follow the
same rules as the original list (that is, be valid variable names, separated by
blanks) except that no parentheses are allowed.

Variables are dropped in sequence from left to right. It is not an error to specify a
name more than one time or to DROP a variable that is not known. If an exposed
variable is named (see section 13.15, "PROCEDURE" on page 183), the variable in
the older generation is dropped.

Example:

```
j=4
Drop  a z.3 z.j
/* Drops the variables: A, Z.3, and Z.4          */
/* so that reference to them returns their names.   */
```

Here, a variable name in parentheses is used as a subsidiary list.

Example:

```
mylist='c d e'
drop (mylist) f
/* Drops the variables C, D, E, and F          */
/* Does not drop MYLIST                         */
```

Specifying a stem (that is, a symbol that contains only one period, as the last
character), drops all variables starting with that stem.

Example:

```
Drop  z.
/* Drops all variables with names starting with Z. */
```

13.6 EXIT

```
►►──EXIT─────────────────;────────────────────────────────►◄
        └─expression─┘
```
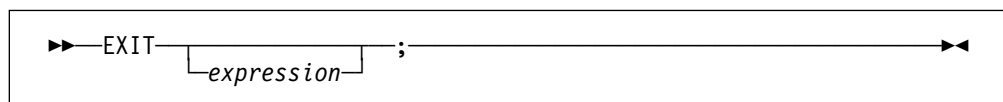
EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is stopped immediately, even if an internal routine is currently being run. If no internal routine is active, RETURN (see page 189) and EXIT are identical in their effect on the program that is being run.

If you specify *expression*, it is evaluated and the string resulting from the evaluation is passed back to the caller when the program stops.

Example:

```
j=3
Exit j*4
/* Would exit with the string '12' */
```

If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error—either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

"Running off the end" of the program is always equivalent to the instruction EXIT, in that it stops the whole program and returns no result string.

Note: If the program was called through a command interface, an attempt is made to convert the returned value to a return code acceptable by the underlying operating system. If the conversion fails, it is deemed to be unsuccessful due to the underlying operating system and thus is not subject to trapping with SIGNAL ON SYNTAX. The returned string must be a whole number whose value fits in a general register (that is, must be in the range -2**31 through 2**31-1).

13.7 IF

```
►►──IF─expression──┬──┬──THEN─instruction──┬──┬──────────────►
                   └─;─┘                    └─;─┘

►──┬─────────────────────────────┬──────────────────────────►◄
   └─ELSE─instruction──┬──┬──────┘
                       └─;─┘
```
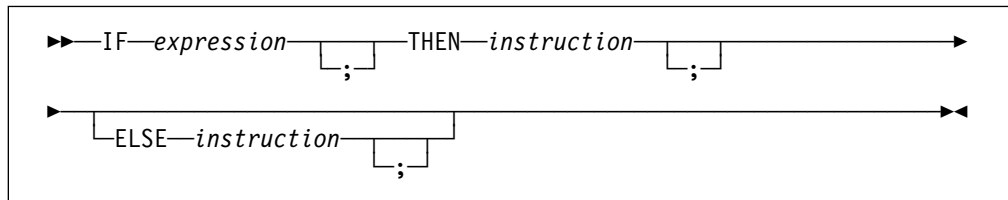
IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* is evaluated and must result in 0 or 1.

The instruction after the THEN is processed only if the result is 1 (true). If you specify an ELSE, the instruction after the ELSE is processed only if the result of the evaluation is 0 (false).

Example:

```
if answer='YES' then say 'OK!'
               else say 'Why not?'
```

Remember that if the ELSE clause is on the same line as the last clause of the THEN part, you need a semicolon before the ELSE.

Example:

```
if answer='YES' then say 'OK!';  else say 'Why not?'
```

The ELSE binds to the nearest IF at the same level. You can use the NOP instruction to eliminate errors and possible confusion when IF constructs are nested, as in the following example.

Example:

```
If answer = 'YES' Then
   If name = 'FRED' Then
      say 'OK, Fred.'
   Else
      nop
Else
   say 'Why not?'
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon (or label) after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.

2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be ended by the THEN, without a ; being required. If this were not so, people who are accustomed to other computer languages would experience considerable difficulties.

13.8 INTERPRET

```
►►──INTERPRET──expression──;──────────────────────────────────────────►◄
```

INTERPRET processes instructions that have been built dynamically by evaluating *expression*.

The *expression* is evaluated and is then processed (interpreted) just as though the resulting string were a line inserted into the program (and bracketed by a DO; and an END;).

Any instructions (including INTERPRET instructions) are allowed, but note that constructions such as DO...END and SELECT...END must be complete. For example, a string of instructions being interpreted cannot contain a LEAVE or ITERATE instruction (valid only within a repetitive DO loop) unless it also contains the whole repetitive DO...END construct.

A semicolon is implied at the end of the expression during execution, if one was not supplied.

Example:

```
data='FRED'
interpret data '= 4'
/* Builds the string  "FRED = 4" and      */
/* Processes:  FRED = 4;                   */
/* Thus the variable FRED is set to "4"    */
```

Example:

```
data='do 3; say "Hello there!"; end'
interpret data        /* Displays:        */
                      /*  Hello there!    */
                      /*  Hello there!    */
                      /*  Hello there!    */
```

Notes:

1. Label clauses are not permitted in an interpreted character string.

2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with `TRACE R` or `TRACE I` in effect is helpful.

   Example:

   ```
   /* Here is a small REXX program. */
   Trace Int
   name='Kitty'
   indirect='name'
   interpret 'say "Hello"' indirect'"!"'
   ```

   When this is run, it gives the trace:

```
kitty
    3 *-* name='Kitty'
      >L>    "Kitty"
    4 *-* indirect='name'
      >L>    "name"
    5 *-* interpret 'say "Hello"' indirect'"!"'
      >L>    "say "Hello""
      >V>    "name"
      >O>    "say "Hello" name"
      >L>    ""!""
      >O>    "say "Hello" name"!""
      *-*  say "Hello" name"!"
      >L>      "Hello"
      >V>      "Kitty"
      >O>      "Hello Kitty"
      >L>      "!"
      >O>      "Hello Kitty!"
Hello Kitty!
```

Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (INDIRECT), and another literal string. The resulting pure character string is then interpreted, just as though it were actually part of the original program. Because it is a new clause, it is traced as such (the second *-* trace flag under line 5) and is then processed. Again a literal string is concatenated to the value of a variable (NAME) and another literal, and the final result (Hello Kitty!) is then displayed.

3. For many purposes, you can use the VALUE function (see page 232) instead of the INTERPRET instruction. The following line could, therefore, have replaced line 5 in the last example:

```
say "Hello" value(indirect)"!"
```

INTERPRET is usually required only in special cases, such as when two or more statements are to be interpreted together, or when an expression is to be evaluated dynamically.

13.9 ITERATE

```
►►──ITERATE──────────;────────────────────────────────────►◄
            └─name─┘
```
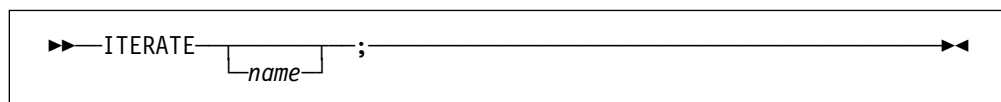
ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO).

Execution of the group of instructions stops, and control is passed to the DO instruction just as though the END clause had been encountered. The control variable (if any) is incremented and tested, as usual, and the group of instructions is processed again, unless the DO instruction ends the loop.

The *name* is a symbol, taken as a constant. If *name* is not specified, ITERATE steps the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and this is the loop that is stepped. Any active loops inside the one selected for iteration are ended (as though by a LEAVE instruction).

Example:

```
do i=1 to 4
  if i=2 then iterate
  say i
  end
/* Displays the numbers:  "1" "3" "4" */
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.

2. A loop is active if it is currently being processed. If a subroutine is called (or an INTERPRET instruction is processed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.

3. If more than one active loop uses the same control variable, ITERATE selects the innermost loop.

13.10 LEAVE

```
►►──LEAVE────────────;──────────────────────────────────────►◄
             └─name─┘
```
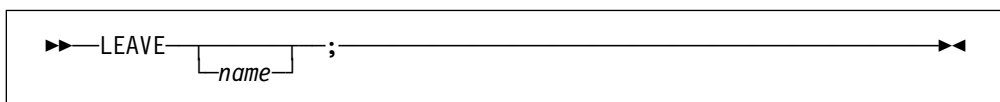
LEAVE causes an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO).

Processing of the group of instructions is ended, and control is passed to the instruction following the END clause, just as though the END clause had been encountered and the termination condition had been met. However, on exit, the control variable (if any) will contain the value it had when the LEAVE instruction was processed.

The *name* is a symbol, taken as a constant. If *name* is not specified, LEAVE ends the innermost active repetitive loop. If *name* is specified, it must be the name of the control variable of a currently active loop (which may be the innermost), and that loop (and any active loops inside it) is then ended. Control then passes to the clause following the END that matches the DO clause of the selected loop.

Example:

```
do i=1 to 5
  say i
  if i=3 then leave
  end
/* Displays the numbers:  "1" "2" "3" */
```

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.

2. A loop is active if it is currently being processed. If a subroutine is called (or an INTERPRET instruction is processed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to end an inactive loop.

3. If more than one active loop uses the same control variable, LEAVE selects the innermost loop.

13.11 NOP

```
►►──NOP──;─────────────────────────────────────────────►◄
```

NOP is a dummy instruction that has no effect. It can be useful as the target of a
THEN or ELSE clause:

Example:

```
Select
   when a=c then nop           /* Do nothing */
   when a>c then say 'A > C'
   otherwise    say 'A < C'
end
```

Note: Putting an extra semicolon instead of the NOP would merely insert a null
clause, which would be ignored. The second WHEN clause would be seen as the
first instruction expected after the THEN, and would, therefore, be treated as a
syntax error. NOP is a true instruction, however, and is, therefore, a valid target for
the THEN clause.

13.12 NUMERIC

```
►►──NUMERIC──┬─DIGITS────┬───────────────────────┬──────────;──────►◄
             │           └─expression1─┘         │
             │         ┌─SCIENTIFIC──────┐        │
             ├─FORM────┼─ENGINEERING─────┤        │
             │         │           ┌─expression2─┘│
             │         └─VALUE─────┘              │
             └─FUZZ────┬───────────────┬──────────┘
                       └─expression3─┘
```
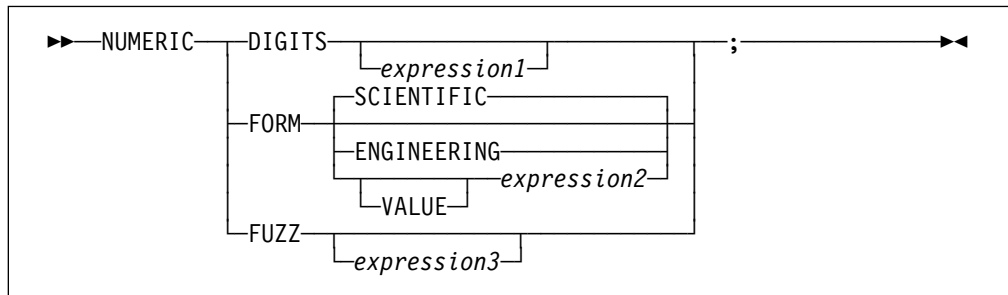
NUMERIC changes the way in which a program carries out arithmetic operations. The options of this instruction are described in detail on pages 255-264, but in summary:

NUMERIC DIGITS

controls the precision to which arithmetic operations and arithmetic built-in functions are evaluated. If you omit *expression1*, the precision defaults to 9 digits. Otherwise, *expression1* must evaluate to a positive whole number and must be larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available—see the note on page 129 for more information) but note that high precisions are likely to require a good deal of processing time. It is recommended that you use the default value wherever possible.

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function, see section 14.3.20, "DIGITS" on page 216.

NUMERIC FORM

controls which form of exponential notation REXX uses for the result of arithmetic operations and arithmetic built-in functions. This may be either SCIENTIFIC (in which case only one, nonzero digit appears before the decimal point) or ENGINEERING (in which case the power of 10 is always a multiple of 3). The default is SCIENTIFIC. The subkeywords SCIENTIFIC or ENGINEERING set the FORM directly, or it is taken from the result of evaluating the expression (*expression2*) that follows VALUE. The result in this case must be either SCIENTIFIC or ENGINEERING. You can omit the subkeyword VALUE if *expression2* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator character or parenthesis).

You can retrieve the current NUMERIC FORM setting with the FORM built-in function, see section 14.3.26, "FORM" on page 218.

NUMERIC FUZZ

controls how many digits, at full precision, are ignored during a numeric comparison operation. (See page 261.) If you omit *expression3*, the default is 0 digits. Otherwise, *expression3* must evaluate to 0 or a positive whole number, rounded if necessary according to the current NUMERIC DIGITS setting, and must be smaller than the current NUMERIC DIGITS setting.

NUMERIC FUZZ temporarily reduces the value of NUMERIC DIGITS by the NUMERIC FUZZ value during every numeric comparison. The numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison and are then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function, see section 14.3.28, "FUZZ" on page 220.

Note: The three numeric settings are automatically saved across internal and external subroutine and function calls. See the CALL instruction (page 160) for more details.

13.13 OPTIONS

```
▶▶──OPTIONS──expression──;──────────────────────────────────────▶◀
```

OPTIONS passes special requests or parameters to the language processor. For example, these may be language processor options or perhaps define a special character set.

The *expression* is evaluated, and the result is examined one word at a time. The language processor converts the words to uppercase. If the language processor recognizes the words, then they are obeyed. Words that are not recognized are ignored and assumed to be instructions to a different processor.

The language processor recognizes the following words:

ETMODE   specifies that literal strings and symbols and comments containing DBCS characters are checked for being valid DBCS strings. If you use this option, it must be the first instruction of the program.

If the *expression* is an external function call, for example `OPTIONS 'GETETMOD'()`, and the program contains DBCS literal strings, enclose the name of the function in quotation marks to ensure that the entire program is not scanned before the option takes effect. It is not recommended to use internal function calls to set ETMODE because of the possibility of errors in interpreting DBCS literal strings in the program.

NOETMODE   specifies that literal strings and symbols and comments containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default. The language processor ignores this option unless it is the first instruction in a program.

EXMODE   specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained.

NOEXMODE   specifies that any data in strings is handled on a byte basis. The integrity of DBCS characters, if any, may be lost. NOEXMODE is the default.

Notes:

1. Because of the language processor's scanning procedures, you must place an `OPTIONS 'ETMODE'` instruction as the first instruction in a program containing DBCS characters in literal strings, symbols, or comments. If you do not place `OPTIONS 'ETMODE'` as the first instruction and you use it later in the program, you receive error message CICREXX488E. If you do place it as the first instruction of your program, all subsequent uses are ignored. If the expression contains anything that would start a label search, all clauses tokenized during the label search process are tokenized within the current setting of ETMODE. Therefore, if this is the first statement in the program, the default is NOETMODE.

2. To ensure proper scanning of a program containing DBCS literals and DBCS comments, enter the words ETMODE, NOETMODE, EXMODE, and

NOEXMODE as literal strings (that is, enclosed in quotation marks) in the OPTIONS instruction.

3. The EXMODE setting is saved and restored across subroutine and function calls.

4. To distinguish DBCS characters from 1-byte EBCDIC characters, sequences of DBCS characters are enclosed with a shift-out (SO) character and a shift-in (SI) character. The hexadecimal values of the SO and SI characters are X'0E' and X'0F', respectively.

5. When you specify OPTIONS 'ETMODE', DBCS characters within a literal string are excluded from the search for a closing quotation mark in literal strings.

6. The words ETMODE, NOETMODE, EXMODE, and NOEXMODE can appear several times within the result. The one that takes effect is determined by the last valid one specified between the pairs ETMODE-NOETMODE and EXMODE-NOEXMODE.

13.14 PARSE
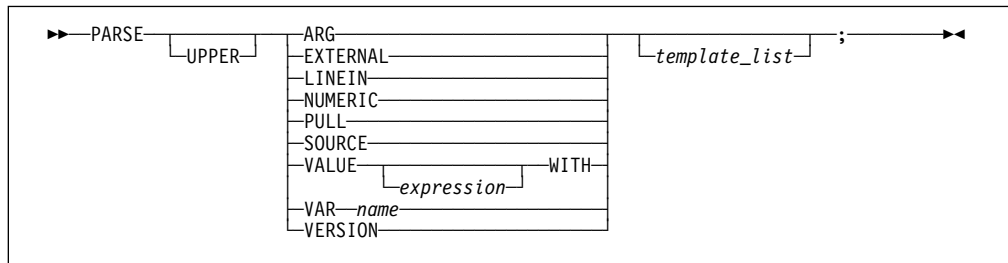
PARSE assigns data (from various sources) to one or more variables according to the rules of parsing. (See Chapter 15, "Parsing" on page 239.)

The *template_list* is often a single template but may be several templates separated by commas. If specified, each template is a list of symbols separated by blanks or patterns or both.

Each template is applied to a single source string. Specifying multiple templates is never a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See page 249 for information on parsing multiple source strings.

If you do not specify a template, no variables are set but action is taken to prepare the data for parsing, if necessary. Thus for PARSE EXTERNAL and PARSE PULL, a data string is removed from the queue, for PARSE LINEIN (and PARSE PULL if the queue is empty), a line is taken from the default input stream, and for PARSE VALUE, *expression* is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition (if it is enabled) is raised.

If you specify the UPPER option, the data to be parsed is first translated to uppercase (that is, lowercase a–z to uppercase A–Z). Otherwise, no uppercase translation takes place during the parsing.

The following list describes the data for each variant of the PARSE instruction.

PARSE ARG

parses the string or strings passed to a program or internal routine as input arguments. (See the ARG instruction on page 158 for details and examples.)

Note: You can also retrieve or check the argument strings to a REXX program or internal routine with the ARG built-in function (see page 207).

PARSE EXTERNAL

This is a non-SAA subkeyword provided in REXX/CICS. The next string from the terminal input buffer is parsed. This queue may contain data that is the result of external asynchronous events—such as user console input, or messages. If that queue is empty, a console read results. Note that this mechanism should not be used for typical console input, for which PULL is more general, but rather for special applications (such as debugging) where the program stack cannot be disturbed.

PARSE LINEIN

This is the same as PARSE EXTERNAL.

PARSE NUMERIC

This is a non-SAA subkeyword provided in VM. The current numeric controls (as set by the NUMERIC instruction, see page 176) are available. These controls are in the order DIGITS FUZZ FORM.

Example:

```
Parse Numeric Var1
```

After this instruction, `Var1` would be equal to: 9 0 SCIENTIFIC. See section 13.12, "NUMERIC" on page 176 and the built-in functions in section 14.3.20, "DIGITS" on page 216, and section 14.3.26, "FORM" on page 218, and section 14.3.28, "FUZZ" on page 220.

PARSE PULL

parses the next string from the external data queue. If the external data queue is empty, PARSE PULL reads a line from the default input stream (the user's terminal), and the program pauses, if necessary, until a line is complete. You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions. You can find the number of lines currently in the queue with the QUEUED built-in function, see page 224. Other programs in the system can alter the queue and use it as a means of communication with programs written in REXX. See also the PULL instruction on page 186.

Note: PULL and PARSE PULL read from the program stack. If that is empty, they read from the terminal input buffer; and if that too is empty, they read from the console. (See the PULL instruction, on page 186, for further details.)

PARSE SOURCE

parses data describing the source of the program running. The language processor returns a string that is fixed (does not change) while the program is running. It returns a source string containing the following tokens:

1. The characters CICS.

2. The string COMMAND, FUNCTION, or SUBROUTINE depending on whether the program was invoked as some kind of host command, from a function call in an expression, by a CALL instruction, or as a server process.

3. The name of the exec in uppercase. The name of the file (RFS), or VSE Librarian Sublibrary from which the exec was originally loaded. The three formats are:

   • Library.sublibrary(member)
   • RFS fully qualified file identifier
   • Dataset name (member)

4. Initial (default) host command environment that is always REXXCICS.

5. Identifier of the specific CICS/ESA environment, which in this case is REXX/CICS.

PARSE VALUE

parses the data that is the result of evaluating *expression*. If you specify no *expression*, the null string is used. Note that WITH is a subkeyword in this context and cannot be used as a symbol within *expression*.

Thus, for example:

```
PARSE VALUE time() WITH  hours ':' mins ':' secs
```

gets the current time and splits it into its constituent parts.

PARSE VAR *name*

parses the value of the variable *name*. The *name* must be a symbol that is valid as a variable name (that is, it cannot start with a period or a digit). Note that the variable *name* is not changed unless it appears in the template, so that for example:

```
PARSE VAR string word1 string
```

removes the first word from *string*, puts it in the variable *word1*, and assigns the remainder back to *string*. Similarly

```
PARSE UPPER VAR string word1 string
```

in addition translates the data from *string* to uppercase before it is parsed.

PARSE VERSION

parses information describing the language level and the date of the language processor. This information consists of five words delimited by blanks:

1. The string `REXX370`, signifying the 370 implementation.
2. The language level description (for example, `3.48`).
3. The language processor release date (for example, `05 April 2000`).

13.15 PROCEDURE

```
►►──PROCEDURE──────────────────────────────────;──────────────────►◄
                 └─EXPOSE──┬──name──┬─┘
                           └─(name)─┘
```
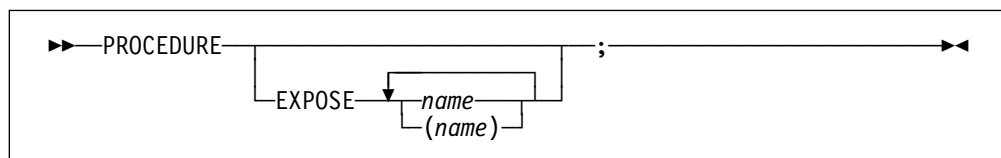
PROCEDURE, within an internal routine (subroutine or function), protects variables by making them unknown to the instructions that follow it. After a RETURN instruction is processed, the original variables environment is restored and any variables used in the routine (that were not exposed) are dropped. (An exposed variable is one belonging to a caller of a routine that the PROCEDURE instruction has exposed. When the routine refers to, or alters, the variable, the original (caller's) copy of the variable is used.) An internal routine need not include a PROCEDURE instruction; in this case the variables it is manipulating are those the caller "owns." If used, the PROCEDURE instruction must be the first instruction processed after the CALL or function invocation; that is, it must be the first instruction following the label.

If you use the EXPOSE option, any variable specified by *name* is exposed. Any reference to it (including setting and dropping) refers to the variables environment the caller owns. Hence, the values of existing variables are accessible, and any changes are persistent even on RETURN from the routine. If *name* is not enclosed in parentheses, it identifies a variable you want to expose and must be a symbol that is a valid variable name, separated from any other *name* with one or more blanks.

If parentheses enclose a single *name*, then, after the variable *name* is exposed, the value of *name* is immediately used as a subsidiary list of variables. (Blanks are not necessary either inside or outside the parentheses, but you can add them if desired.) This subsidiary list must follow the same rules as the original list (that is, valid variable names, separated by blanks) except that no parentheses are allowed.

Variables are exposed in sequence from left to right. It is not an error to specify a name more than one time, or to specify a name that the caller has not used as a variable.

Any variables in the main program that are not exposed are still protected. Therefore, some limited set of the caller's variables can be made accessible, and these variables can be changed (or new variables in this set can be created). All these changes are visible to the caller upon RETURN from the routine.

Example:

```
/* This is the main REXX program */
j=1; z.1='a'
call toft
say j k m        /* Displays "1 7 M"          */
exit

/* This is a subroutine      */
toft: procedure expose j k z.j
   say j k z.j /* Displays "1 K a"           */
   k=7; m=3    /* Note: M is not exposed     */
   return
```

Note that if Z.J in the EXPOSE list had been placed before J, the caller's value of J would not have been visible at that time, so Z.1 would not have been exposed.

The variables in a subsidiary list are also exposed from left to right.

Example:

```
/* This is the main REXX program */
j=1;k=6;m=9
a ='j k m'
call test
exit

/* This is a subroutine      */
test: procedure expose (a)   /* Exposes A, J, K, and M     */
  say a j k m                /* Displays "j k m 1 6 9"     */
  return
```

You can use subsidiary lists to more easily expose a number of variables at one time or, with the VALUE built-in function, to manipulate dynamically named variables.

Example:

```
/* This is the main REXX program */
c=11; d=12; e=13
Showlist='c d'     /* but not E                */
call Playvars
say c d e f        /* Displays "11 New 13 9" */
exit

/* This is a subroutine      */
Playvars: procedure expose (showlist) f
 say word(showlist,2)              /* Displays "d"                    */
 say value(word(showlist,2),'New') /* Displays "12" and sets new value */
 say value(word(showlist,2))       /* Displays "New"                  */
 e=8                               /* E is not exposed                */
 f=9                               /* F was explicitly exposed        */
 return
```

Specifying a **stem** as *name* exposes this stem and *all possible* compound variables whose names begin with that stem. (See page 145 for information about stems.)

Example:

```
/* This is the main REXX program */
a.=11; i=13; j=15
i = i + 1
C.5 = 'FRED'
call lucky7
say a. a.1 i j c. c.5
say 'You should see 11 7 14 15 C. FRED'
exit
lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose    */
/* names start with A. or C.                      */
A.1='7'  /* This sets A.1 in the caller's          */
         /* environment, even if it did not        */
         /* previously exist.                      */
return
```

Variables may be exposed through several generations of routines, if desired, by ensuring that they are included on all intermediate PROCEDURE instructions.

See the CALL instruction and function descriptions on pages 160 and 201 for details and examples of how routines are called.

13.16 PULL

```
▶▶──PULL─────────────────────;──────────────────────────▶◀
          └─template_list─┘
```
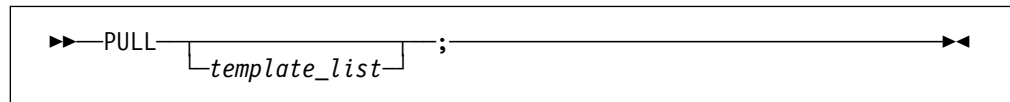
PULL reads a string from the program stack. If the program stack is empty, PULL then tries reading a line from the current terminal input device. It is just a short form of the instruction:

```
▶▶──PARSE UPPER PULL─────────────────;──▶◀
                  └─template_list─┘
```
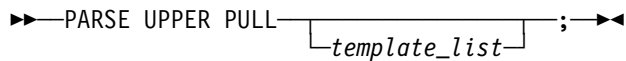
The current head-of-queue is read as one string. Without a *template_list* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template_list* is usually a single template, which is a list of symbols separated by blanks or patterns or both. (The *template_list* can be several templates separated by commas, but PULL parses only one source string; if you specify several comma-separated templates, variables in templates other than the first one are assigned the null string.) The string is translated to uppercase (that is, lowercase a–z to uppercase A–Z) and then parsed into variables according to the rules described in the section on parsing (page 239). Use the PARSE PULL instruction if you do not desire uppercase translation.

Notes:

1. The REXX/CICS implementation of the external data queue is the program stack. The language processor reads a line from the program stack. If the program stack is empty, a terminal read occurs. The program stack for you is in an RLS queue named \SYSTEM*rexxtnum**PROGSTACK* where *rexxtrnum* is an internal REXX task number.

2. If the PULL causes a read from the terminal, the variable PULLKEY is set upon completion of the PULL command. It will contain the name of the aid key pressed in response to the PULL command (for example: ENTER, PFKEY 1, PAKEY 1, MSR, PEN, or CLEAR).

For information on named queues, see the REXX List System LPULL command in section 20.4.3, "LPULL" on page 320.

Example:

```
Say 'Do you want to erase the file?  Answer Yes or No:'
Pull answer .
if answer='NO' then say 'The file will not be erased.'
```

Here the dummy placeholder, a period (.), is used on the template to isolate the first word the user enters.

If the external data queue is empty, a console read is issued and the program pauses, if necessary, until a line is complete.

The QUEUED built-in function (see page 224) returns the number of lines currently in the program stack.

13.17 PUSH

```
►►──PUSH──────────────────;──────────────────────────────────►◄
           └─expression─┘
```
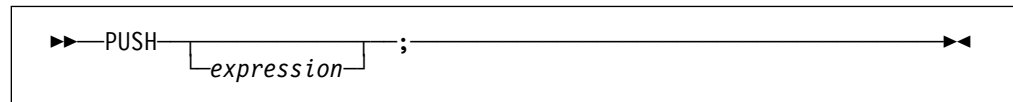
PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue.

If you do not specify *expression*, a null string is stacked.

Note: The REXX/CICS implementation of the external data queue is the program stack. The language processor reads a line from the program stack. If the program stack is empty, a terminal read occurs. The program stack for you is in an RLS queue named \SYSTEM*rexxtnum**PROGSTACK* where *rexxtrnum* is an internal REXX task number.

For information on named queues, see the REXX List System LPUSH command, section 20.4.4, "LPUSH" on page 320.

Example:

```
a='Fred'
push        /* Puts a null line onto the queue */
push a 2    /* Puts "Fred 2"    onto the queue */
```

The QUEUED built-in function (described on page 224) returns the number of lines currently in the external data queue.

13.18 QUEUE

```
►►──QUEUE──────────────────;─────────────────────────────────►◄
            └─expression─┘
```
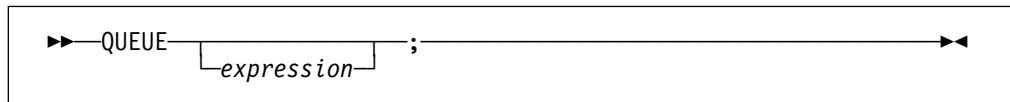
QUEUE appends the string resulting from *expression* to the tail of the external data queue. That is, it is added FIFO (First In, First Out).

If you do not specify *expression*, a null string is queued.

Note: The REXX/CICS implementation of the external data queue is the program stack. The language processor reads a line from the program stack. If the program stack is empty, a terminal read occurs. The program stack for you is in an RLS queue named \SYSTEM*rexxtnum**PROGSTACK* where *rexxtrnum* is an internal REXX task number.

Example:

```
a='Toft'
queue a 2  /* Enqueues "Toft 2" */
queue      /* Enqueues a null line behind the last */
```

The QUEUED built-in function (described on page 224) returns the number of lines currently in the external data queue.

13.19 RETURN

```
►►──RETURN─────────────────;──────────────────────────►◄
            └─expression─┘
```
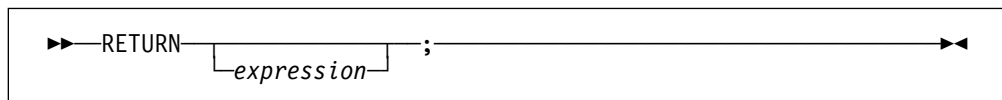
RETURN returns control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is being run, (see page 169.)

If a *subroutine* is being run (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so forth) are also restored, (see page 160.)

If a *function* is being processed, the action taken is identical, except that *expression* **must** be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was called. See the description of functions on page 201 for more details.

If a PROCEDURE instruction was processed within the routine (subroutine or internal function), all variables of the current generation are dropped (and those of the previous generation are exposed) after *expression* is evaluated and before the result is used or assigned to RESULT.

13.20 SAY

```
►►──SAY────────────;──────────────────────────────────────►◄
         └─expression─┘
```
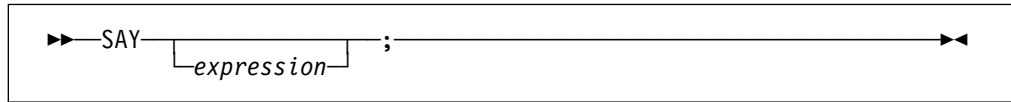
SAY writes a line to the default output stream (the terminal) so the user sees it displayed. The result of *expression* may be of any length. If you omit *expression*, the null string is written.

You can use the SET TERMOUT command to redirect SAY output (see page 421).

Example:

```
data=100
Say data 'divided by 4 =>' data/4
/* Displays: "100 divided by 4 => 25"  */
```

13.21 SELECT
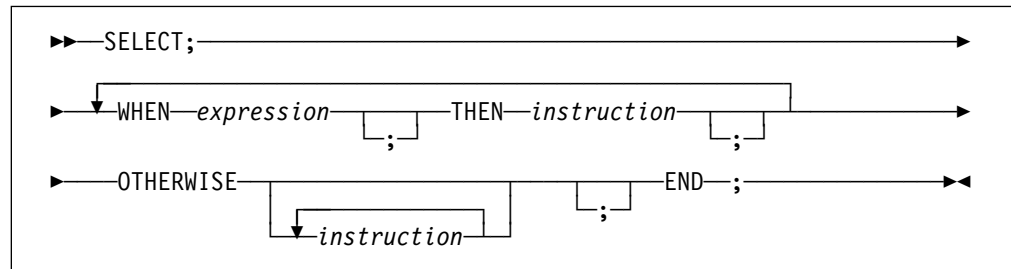
SELECT conditionally calls one of several alternative instructions.

Each *expression* after a WHEN is evaluated in turn and must result in 0 or 1. If the result is 1, the instruction following the associated THEN (which may be a complex instruction such as IF, DO, or SELECT) is processed and control then passes to the END. If the result is 0, control passes to the next WHEN clause.

If none of the WHEN expressions evaluates to 1, control passes to the instructions, if any, after OTHERWISE. In this situation, the absence of an OTHERWISE causes an error (but note that you can omit the instruction list that follows OTHERWISE).

Example:

```
balance=100
check=50
balance = balance - check
Select
  when balance > 0 then
       say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
       say 'Warning, Balance is now zero!  STOP all spending.'
       say "You cut it close this month! Hope you do not have any"
       say "checks left outstanding."
       end
  Otherwise
       say "You have just overdrawn your account."
       say "Your balance now shows" balance "dollars."
       say "Oops!  Hope the bank does not close your account."
end  /* Select */
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, IF, or the SELECT instruction itself.

2. A null clause is not an instruction, so putting an extra semicolon (or label) after a THEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.

3. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the WHEN clause to be ended by the THEN without a ; (delimiter) being required.

13.22 SIGNAL

```
►►──SIGNAL──┬─labelname─────────────────────────────────┬──;──►◄
            │        ┌─expression─┐                      │
            │  └─VALUE─┘                                 │
            ├─OFF──┬─ERROR───┬────────────────────────────┤
            │      ├─FAILURE─┤                            │
            │      ├─HALT────┤                            │
            │      ├─NOVALUE─┤                            │
            │      └─SYNTAX──┘                            │
            └─ON───┬─ERROR───┬──┬──────────────────────┐
                   ├─FAILURE─┤  └─NAME──trapname─┘
                   ├─HALT────┤
                   ├─NOVALUE─┤
                   └─SYNTAX──┘
```
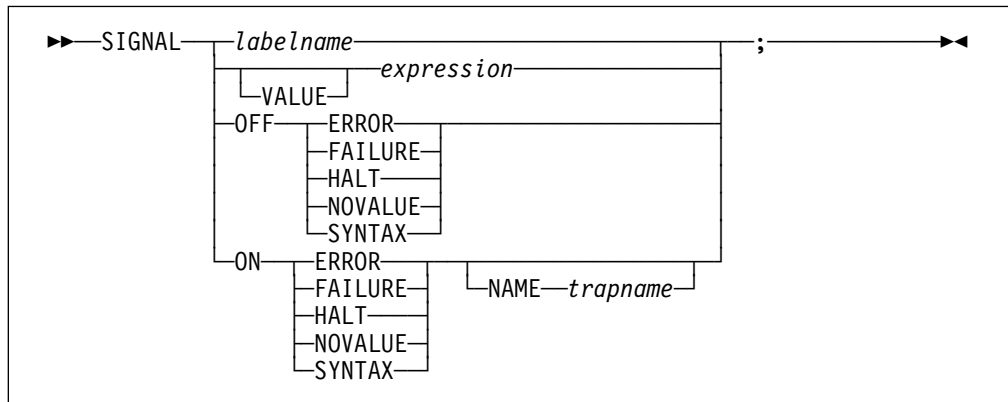
SIGNAL causes an *unusual* change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 17, "Conditions and Condition Traps" on page 265.

To change the flow of control, a label name is derived from *labelname* or taken from the result of evaluating the *expression* after VALUE. The *labelname* you specify must be a literal string or symbol that is taken as a constant. If you use a symbol for *labelname*, the search is independent of alphabetic case. If you use a literal string, the characters should be in uppercase. This is because the language processor translates all labels to uppercase, regardless of how you enter them in the program. Similarly, for SIGNAL VALUE, the *expression* must evaluate to a string in uppercase or the language processor does not find the label. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator character or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then ended (that is, they cannot be resumed). Control then passes to the first label in the program that matches the given name, as though the search had started from the top of the program.

Example:

```
Signal fred;  /* Transfer control to label FRED below */
  ....
  ....
Fred: say 'Hi!'
```

Because the search effectively starts at the top of the program, control always passes to the first occurrence, if duplicates are present, of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because you can use SIGL to determine the source of a transfer of control to a label.

For information about using SIGNAL with the INTERPRET instruction, see Note 1 on page 171.

Using SIGNAL VALUE

The VALUE form of the SIGNAL instruction allows a branch to a label whose name is determined at the time of execution. This can safely effect a multi-way CALL (or function call) to internal routines because any DO loops, and so forth, in the calling routine are protected against termination by the call mechanism.

Example:

```
fred='PETE'
call multiway fred, 7
   ....
   ....
exit
Multiway: procedure
   arg label .              /* One word, uppercase                 */
                           /* Can add checks for valid labels here */
   signal value label      /* Transfer control to wherever        */
   ....
Pete: say arg(1) '!' arg(2) /* Displays: "PETE ! 7"               */
   return
```

13.23 TRACE

```
►►──TRACE─────────────────────────────────────────────────;──────────►◄
           ┌─number─┐
           │        ├────────────────────────┐
           │        │         ┌─Normal────────┐
           │        │         ├─All───────────┤
           │        │  ┌─?─┐  ├─Commands──────┤
           └────────┴──┤   ├──├─Error─────────┤
                       └─!─┘  ├─Failure───────┤
                              ├─Intermediates─┤
                              ├─Labels────────┤
                              ├─Off───────────┤
                              ├─Results───────┤
                              └─Scan──────────┘
```

Or, alternatively:

```
►►──TRACE───────────────────────────;──────────────────►◄
          ┌─string─────────────┐
          ├─symbol─────────────┤
          └───────┬─expression─┘
               └─VALUE─┘
```
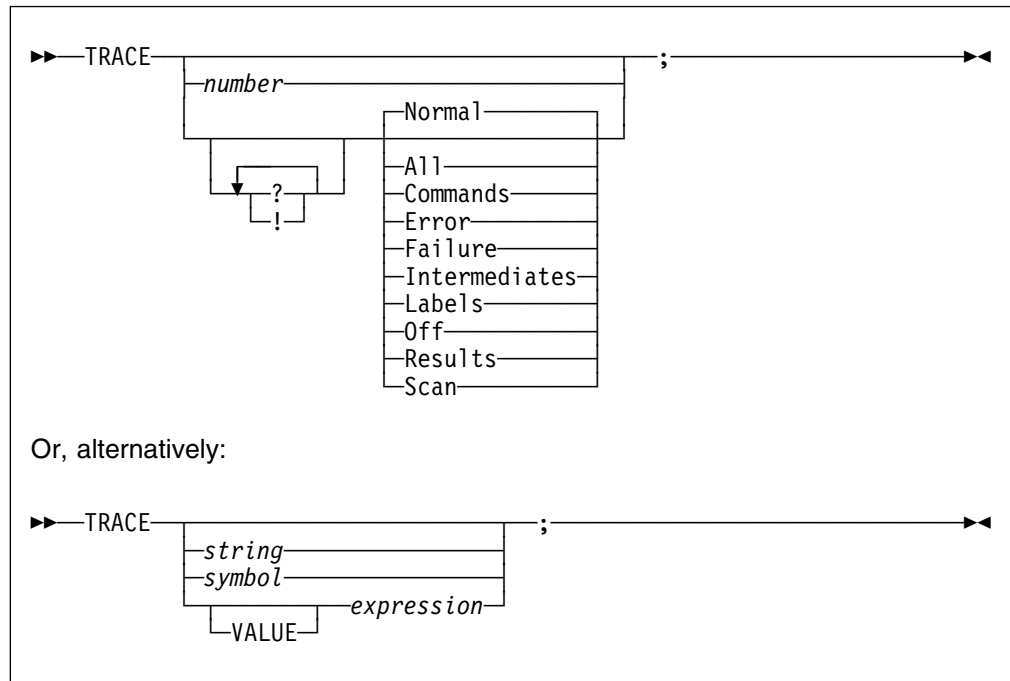
TRACE controls the tracing action (that is, how much is displayed to the user) during processing of a REXX program. (Tracing describes some or all of the clauses in a program, producing descriptions of clauses as they are processed.) TRACE is mainly used for debugging. Its syntax is more concise than that of other REXX instructions because TRACE is usually entered manually during interactive debugging. (This is a form of tracing in which the user can interact with the language processor while the program is running.) For this use, economy of key strokes is especially convenient.

If specified, the *number* must be a whole number.

The *string* or *expression* evaluates to:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later
- Null.

The *symbol* is taken as a constant, and is, therefore:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later.

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator or parenthesis).

13.23.1 Alphabetic Character (Word) Options

Although you can enter the word in full, only the capitalized and highlighted letter is needed; all characters following it are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All Traces (that is, displays) all clauses before execution.

Commands Traces all commands before execution. If the command results in an error or failure[4] ,tracing also displays the return code from the command.

Error Traces any command resulting in an error or failure[4] after execution, together with the return code from the command.

Failure Traces any command resulting in a failure[4] after execution, together with the return code from the command. This is the same as the `Normal` option.

Intermediates Traces all clauses before execution. Also traces intermediate results during evaluation of expressions and substituted names.

Labels Traces only labels passed during execution. This is especially useful with debug mode, when the language processor pauses after each label. It also helps the user to note all internal subroutine calls and transfers of control because of the SIGNAL instruction.

Normal Traces any command resulting in a negative return code after execution, together with the return code from the command. **This is the default setting**.

Off Traces nothing and resets the special prefix options (described later) to OFF.

Results Traces all clauses before execution. Displays final results (contrast with `Intermediates`, preceding) of evaluating an expression. Also displays values assigned during PULL, ARG, and PARSE instructions. **This setting is recommended for general debugging.**

Scan Traces all remaining clauses in the data without them being processed. Basic checking (for missing ENDs and so forth) is carried out, and the trace is formatted as usual. This is valid only if the TRACE S clause itself is not nested in any other instruction (including INTERPRET or interactive debug) or in an internal routine.

---

[4] See page 147 for definitions of error and failure.

13.23.2 Prefix Options

The prefixes ! and ? are valid either alone or with one of the alphabetic character options. You can specify both prefixes, in any order, on one TRACE instruction. You can specify a prefix more than one time, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix(es) must immediately precede the option (no intervening blanks).

The prefixes ! and ? modify tracing and execution as follows:

? Controls interactive debug. During usual execution, a TRACE option with a prefix of ? causes interactive debug to be switched on. (See E.1, "Interactive Debugging of Programs" on page 469 for full details of this facility.) While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction `TRACE ?E` makes the language processor pause for input after executing any command that returns an error (that is, a nonzero return code).

Any TRACE instructions in the program being traced are ignored. (This is so that you are not taken out of interactive debug unexpectedly.)

You can switch off interactive debug in several ways:

- Entering `TRACE 0` turns off all tracing.
- Entering `TRACE` with no options restores the defaults—it turns off interactive debug but continues tracing with TRACE Normal (which traces any failing command after execution) in effect.
- Entering `TRACE ?` turns off interactive debug and continues tracing with the current option.
- Entering a TRACE instruction with a ? prefix before the option turns off interactive debug and continues tracing with the new option.

Using the ? prefix, therefore, switches you alternately in or out of interactive debug. (Because the language processor ignores any further TRACE statements in your program after you are in interactive debug, use `CALL TRACE '?'` to turn off interactive debug.)

! Inhibits host command execution. During regular execution, a TRACE instruction with a prefix of ! suspends execution of all subsequent host commands. For example, `TRACE !C` causes commands to be traced but not processed. As each command is bypassed, the REXX special variable RC is set to `0`. You can use this action for debugging potentially destructive programs. (Note that this does not inhibit any commands entered manually while in interactive debug. These are always processed.)

You can switch off command inhibition, when it is in effect, by issuing a TRACE instruction with a prefix !. Repeated use of the ! prefix, therefore, switches you alternately in or out of command inhibition mode. Or, you can turn off command inhibition at any time by issuing `TRACE 0` or `TRACE` with no options.

13.23.3 Numeric Options

If interactive debug is active *and* if the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped over. (See separate section in E.1, "Interactive Debugging of Programs" on page 469, for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, `TRACE -100` means that

the next 100 clauses that would usually be traced are not, in fact, displayed. After that, tracing resumes as before.

13.23.3.1 Tracing Tips

1. When a loop is being traced, the DO clause itself is traced on every iteration of the loop.

2. You can retrieve the trace actions currently in effect by using the TRACE built-in function (see section 14.3.53, "TRACE" on page 230).

3. If available at the time of execution, comments associated with a traced clause are included in the trace, as are comments in a null clause, if you specify TRACE A, R, I, or S.

4. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.

5. Trace actions are automatically saved across subroutine and function calls. See the CALL instruction (page 160) for more details.

13.23.4 A Typical Example

One of the most common traces you will use is:

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/*  and tracing Results of expressions begins.    */
```

13.23.5 Format of TRACE Output

Every clause traced appears with automatic formatting (indentation) according to its logical depth of nesting and so forth. The language processor may replace any control codes in the encoding of data (for example, EBCDIC values less than `'40'x`) with a question mark (?) to avoid console interference. Results (if requested) are indented an extra two spaces and are enclosed in double quotation marks so that leading and trailing blanks are apparent.

A line number precedes the first clause traced on any line. If the line number is greater than 99999, the language processor truncates it on the left, and the ? prefix indicates the truncation. For example, the line number 100354 appears as `?00354`. All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

`*-*`   Identifies the source of a single clause, that is, the data actually in the program.

`+++`   Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).

`>>>`   Identifies the result of an expression (for `TRACE R`) or the value assigned to a variable during parsing, or the value returned from a subroutine call.

`>.>`   Identifies the value "assigned" to a placeholder during parsing (see page 241).

The following prefixes are used only if `TRACE Intermediates` is in effect:

>C>   The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

>F>   The data traced is the result of a function call.

>L>   The data traced is a literal (string, uninitialized variable, or constant symbol).

>O>   The data traced is the result of an operation on two terms.

>P>   The data traced is the result of a prefix operation.

>V>   The data traced is the contents of a variable.

If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N , command inhibition (!) off, and interactive debug (?) off.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced. Any CALL or INTERPRET or function invocations active at the time of the error are also traced. If an attempt to transfer control to a label that could not be found caused the error, that label is also traced. The special trace prefix +++ identifies these traceback lines.

13.24 UPPER

This is a non-SAA instruction provided in REXX/CICS.

```
►►──UPPER──┬─▼─variable─┬──;──────────────────────────────►◄
```
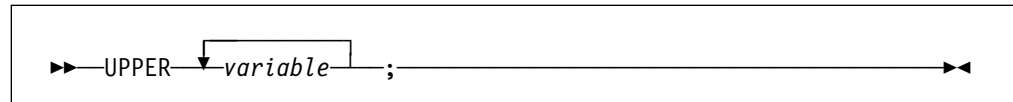
UPPER translates the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

The *variable* is a symbol, separated from any other *variable*s by one or more blanks or comments. Specify only simple symbols and compound symbols. (See page 144.)

Using this instruction is more convenient than repeatedly invoking the TRANSLATE built-in function.

Example:

```
a1='Hello';  b1='there'
Upper a1 b1
say a1 b1     /* Displays "HELLO THERE" */
```

An error is signalled if a constant symbol or a stem is encountered. Using an uninitialized variable is *not* an error, and has no effect, except that it is trapped if the NOVALUE condition (SIGNAL ON NOVALUE) is enabled.

UPPER

Chapter 14. Functions

A **function** is an internal, built-in, or external routine that returns a single result string. (A **subroutine** is a function that is an internal, built-in, or external routine that may or may not return a result and that is called with the CALL instruction.)

14.1 Syntax

A **function call** is a term in an expression that calls a routine that carries out some procedures and returns a string. This string replaces the function call in the continuing evaluation of the expression. You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:

►►──*function_name*(──┬──────,──────┬──)──►◄
                      └─*expression*─┘
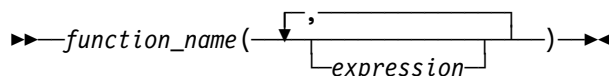
The *function_name* is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. In VM, the implementation maximum is up to 20 expressions. These expressions are called the **arguments** to the function. Each argument expression may include further function calls.

Note that the left parenthesis must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A blank operator would be assumed at this point instead.) Only a comment (which has no effect) can appear between the name and the left parenthesis.

The arguments are evaluated in turn from left to right and the resulting strings are all then passed to the function. This then runs some operation (usually dependent on the argument strings passed, though arguments are not mandatory) and eventually returns a single character string. This string is then included in the original expression just as though the entire function reference had been replaced by the name of a variable whose value is that returned data.

For example, the function SUBSTR is built-in to the language processor (see page 227) and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'substr(N1,2,7)
/* Sets Z1 to 'Part of N1 is: bcdefgh' */
```

A function may have a variable number of arguments. You need to specify only those that are required. For example, SUBSTR('ABCDEF',4) would return DEF.

14.2 Functions and Subroutines

The function calling mechanism is identical with that for subroutines. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not.

The following types of routines can be called as functions:

Internal   If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine called by the CALL instruction, various other status information (TRACE and NUMERIC settings and so forth) is saved too. See the CALL instruction (page 160) for details about this. You can use SIGNAL and CALL together to call an internal routine whose name is determined at the time of execution; this is known as a multi-way call (see page 193).

If you are calling an internal routine as a function, you *must* specify an expression in any RETURN instruction to return from it. This is not necessary if it is called as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x'! =' factorial(x)
exit

factorial: procedure   /* Calculate factorial by   */
  arg n                /*   recursive invocation.  */
  if n=0 then return 1
  return  factorial(n-1) * n
```

While searching for an internal label, syntax checking is performed and the exec is tokenized. See Appendix I, "Performance Considerations" on page 485 for more details. FACTORIAL is unusual in that it calls itself (this is recursive invocation). The PROCEDURE instruction ensures that a new variable n is created for each invocation.

Note: When there is a search for a routine, the language processor currently scans the statements in the REXX program to locate the internal label. During the search, the language processor may encounter a syntax error. As a result, a syntax error may be raised on a statement different from the original line being processed.

Built-in   These functions are always available and are defined in the next section of this manual. (See pages 205—237.)

External   You can write or use functions that are external to your program and to the language processor. External routines must be written in REXX. You can call a REXX program as a function and, in this case, pass more than one argument string. The ARG or PARSE ARG instructions or the ARG built-in function can retrieve these argument strings. When called as a function, a program must return data to the caller.

Notes:

1. External REXX functions can easily perform an EXEC CICS LINK to a program written in any CICS-supported language. Also, REXX/CICS command routines may be written in assembler.

2. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller's variables are always hidden and the status of internal values (NUMERIC settings and so forth) start with their defaults (rather than inheriting those of the caller).

3. Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the called REXX program, and in either case you must specify an expression.

4. With care, you can use the INTERPRET instruction to process a function with a variable function name. However, you should avoid this if possible because it reduces the clarity of the program.

14.2.1 Search Order

The search order for functions is: internal routines take precedence, then built-in functions, and finally external functions.

Internal routines are *not* used if the function name is given as a literal string (that is, specified in quotation marks); in this case the function must be built-in or external. This lets you usurp the name of, say, a built-in function to extend its capabilities, yet still be able to call the built-in function when needed.

Example:
```
/* This internal DATE function modifies the      */
/* default for the DATE function to standard date.  */
date: procedure
      arg in
      if in='' then in='Standard'
      return 'DATE'(in)
```

Built-in functions have uppercase names, and so the name in the literal string must be in uppercase for the search to succeed, as in the example. The same is usually true of external functions.

External functions and **subroutines** have a system-defined search order. The search order for **external functions** and **subroutines** follows.

Whenever an exec, command, external function, or subroutine, written in REXX is invoked by REXX/CICS (for example: from the CICS command line, CALL instruction, EXEC command, or a command from within an exec), the search order for locating the target exec is as follows:

- Search order for a DEFCMD defined command exec or other called exec:

  Note: If this is an attempt to execute an authorized command, a check is made to see if this is an authorized user or the command is in an exec loaded from a VSE Librarian sublibrary specified on a SETSYS AUTHCLIB or SETSYS AUTHELIB command. If none of these are true, the command fails with a return code of -4.

  1. Within the current exec for an internal function or subroutine.

> **Note:** This search can be bypassed by explicitly identifying the function (or subroutine) as external by enclosing its name in quotes.

2. Execs in storage (from an earlier EXECLOAD).

3. The current RFS directory.

4. The current PATH. RFS directories and VSE Librarian sublibraries are searched in the order listed in the most recent PATH command, if it was executed.

5. VSE Librarian sublibraries in the order listed in the most recent SETSYS AUTHCLIB command.

   If the user is an authorized user or if the current exec was loaded from a sublibrary specified on the last SETSYS AUTHELIB command and the search is for an authorized command's program, then check the sublibraries specified on the most recent SETSYS AUTHCLIB command.

6. VSE Librarian sublibraries in the order listed in the most recent SETSYS AUTHELIB command.

7. VSE Librarian members with a member type of PROC in the LIBDEF PROC search chain for the CICS partition.

14.2.2 Errors During Execution

If an external or built-in function detects an error of any kind, the language processor is informed, and a syntax error results. Execution of the clause that included the function call is, therefore, ended. Similarly, if an external function fails to return data correctly, the language processor detects this and reports it as an error.

If a syntax error occurs during the execution of an internal function, it can be trapped (using SIGNAL ON SYNTAX) and recovery may then be possible. If the error is not trapped, the program is ended.

14.3 Built-in Functions

REXX provides a rich set of built-in functions, including character manipulation, conversion, and information functions.

Other built-in and external functions are generally available—see section 14.4, "External Functions Provided in REXX/CICS" on page 237.

The following are general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.

- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated. Any argument named as a *number* is rounded, if necessary, according to the current setting of NUMERIC DIGITS (just as though the number had been added to 0) and checked for validity before use. This occurs in the following functions: ABS, FORMAT, MAX, MIN, SIGN, and TRUNC, and for certain options of DATATYPE. This is not true for RANDOM.

- Any argument named as a *string* may be a null string.

- If an argument specifies a *length*, it must be a positive whole number or zero. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.

- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, DATATYPE(1`,`), like DATATYPE(1), would return NUM. You can include any number of trailing commas; they are ignored. (Where there are actual parameters, the default values apply.)

- If you specify a *pad* character, it must be exactly one character long. (A pad character extends a string, usually on the right. For an example, see the LEFT built-in function on page 222.)

- If a function has an *option* you can select by specifying the first character of a string, that character can be in upper- or lowercase.

- A number of the functions described in this chapter support DBCS. A complete list and descriptions of these functions are in Appendix C, "Double-Byte Character Set (DBCS) Support" on page 449.

14.3.1 ABBREV (Abbreviation)

```
►►──ABBREV(information,info──────────────)──────────────►◄
                          └─,length─┘
```
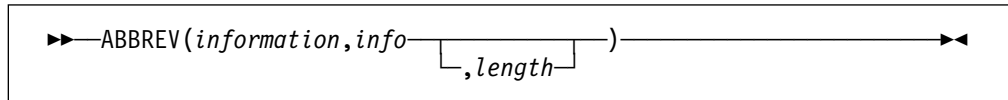
returns 1 if *info* is equal to the leading characters of *information* **and** the length of *info* is not less than *length*. Returns 0 if either of these conditions is not met.

If you specify *length*, it must be a positive whole number or zero. The default for *length* is the number of characters in *info*.

Here are some examples:

```
ABBREV('Print','Pri')       ->    1
ABBREV('PRINT','Pri')       ->    0
ABBREV('PRINT','PRI',4)     ->    0
ABBREV('PRINT','PRY')       ->    0
ABBREV('PRINT','')          ->    1
ABBREV('PRINT','',1)        ->    0
```

Note: A null string always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired; for example:

```
say 'Enter option:';   pull option .
select  /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
  otherwise nop;
end;
```

14.3.2 ABS (Absolute Value)

```
►►──ABS(number)──────────────────────────────►◄
```
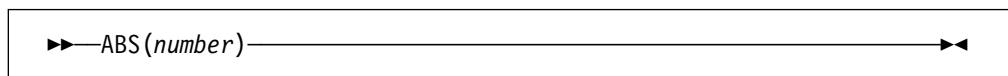
returns the absolute value of *number*. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

```
ABS('12.3')      ->    12.3
ABS(' -0.307')   ->    0.307
```

14.3.3 ADDRESS

```
►►──ADDRESS()─────────────────────────────────►◄
```
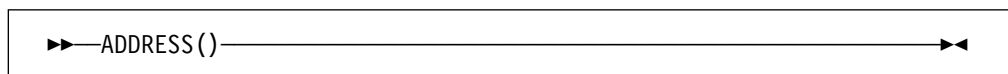
returns the name of the environment to which commands are currently being submitted. The environment may be a name of a subcommand environment. See the ADDRESS instruction (page 156) for more information. Trailing blanks are removed from the result.

Here are some examples:

```
ADDRESS()    ->    'CICS'      /* default under CICS   */
ADDRESS()    ->    'EDITSVR'   /* default under CICS editor */
```

14.3.4 ARG (Argument)
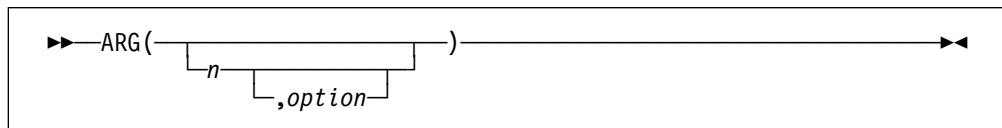
```
►►──ARG(─┬──────────────┬─)────────────────────►◄
         └─n─┬────────┬─┘
             └─,option─┘
```

returns an argument string or information about the argument strings to a program or internal routine.

If you do not specify *n*, the number of arguments passed to the program or internal routine is returned.

If you specify only *n*, the *n*th argument string is returned. If the argument string does not exist, the null string is returned. The *n* must be a positive whole number.

If you specify *option*, ARG tests for the existence of the *n*th argument string. The following are valid *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Exists   returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted   returns 1 if the *n*th argument was omitted; that is, if it was *not* explicitly specified when the routine was called. Returns 0 otherwise.

Here are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()        ->    0
ARG(1)       ->    ''
ARG(2)       ->    ''
ARG(1,'e')   ->    0
ARG(1,'O')   ->    1

/*  following "Call name 'a',,'b';" */
ARG()        ->    3
ARG(1)       ->    'a'
ARG(2)       ->    ''
ARG(3)       ->    'b'
ARG(n)       ->    ''     /* for n>=4 */
ARG(1,'e')   ->    1
ARG(2,'E')   ->    0
ARG(2,'O')   ->    1
ARG(3,'o')   ->    0
ARG(4,'o')   ->    1
```

Notes:

1. The number of argument strings is the largest number *n* for which `ARG(n,'e')` would return 1 or 0 if there are no explicit argument strings. That is, it is the position of the last explicitly specified argument string.

2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1

argument string if anything else (including blanks) is included with the command.

3. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See pages 158, 180, and 239.)

14.3.5 BITAND (Bit by Bit AND)
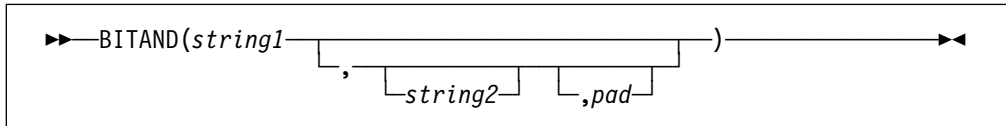
```
>>--BITAND(string1----------------------------------)----------><
                   |    _____    _____   |
                   |___|            |__|        |__|
                    ,    string2       ,pad
```

returns a string composed of the two input strings logically ANDed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITAND('12'x)                    ->     '12'x
BITAND('73'x,'27'x)              ->     '23'x
BITAND('13'x,'5555'x)            ->     '1155'x
BITAND('13'x,'5555'x,'74'x)      ->     '1154'x
BITAND('pQrS',,'BF'x)            ->     'pqrs'        /* EBCDIC */
```

14.3.6 BITOR (Bit by Bit OR)

```
>>--BITOR(string1-----------------------------------)----------><
                  |    _____    _____   |
                  |___|            |__|        |__|
                   ,    string2       ,pad
```
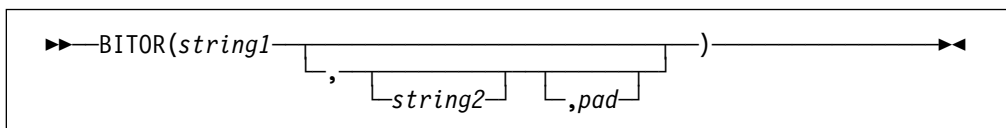
returns a string composed of the two input strings logically inclusive-ORed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITOR('12'x)                     ->     '12'x
BITOR('15'x,'24'x)               ->     '35'x
BITOR('15'x,'2456'x)             ->     '3556'x
BITOR('15'x,'2456'x,'F0'x)       ->     '35F6'x
BITOR('1111'x,,'4D'x)            ->     '5D5D'x
BITOR('pQrS',,'40'x)             ->     'PQRS' /* EBCDIC */
```

14.3.7 BITXOR (Bit by Bit Exclusive OR)

```
►►──BITXOR(string1──────────────────────)──────────►◄
             └─,─┬─────────┬──┬──────┬─┘
                 └─string2─┘  └─,pad─┘
```
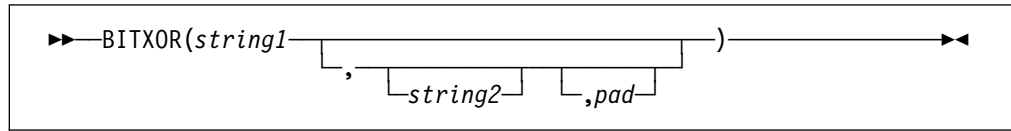
returns a string composed of the two input strings logically eXclusive-ORed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the XOR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITXOR('12'x)                    -> '12'x
BITXOR('12'x,'22'x)              -> '30'x
BITXOR('1211'x,'22'x)           -> '3011'x
BITXOR('1111'x,'444444'x)       -> '555544'x
BITXOR('1111'x,'444444'x,'40'x) -> '555504'x
BITXOR('1111'x,,'4D'x)          -> '5C5C'x
BITXOR('C711'x,'222222'x,' ')   -> 'E53362'x  /* EBCDIC */
```

14.3.8 B2X (Binary to Hexadecimal)

```
►►──B2X(binary_string)──────────────────────────────►◄
```
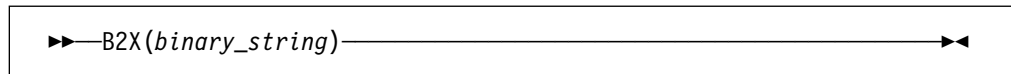
returns a string, in character format, that represents *binary_string* converted to hexadecimal.

The *binary_string* is a string of binary (0 or 1) digits. It can be of any length. You can optionally include blanks in *binary_string* (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string uses uppercase alphabetics for the values A–F, and does not include blanks.

If *binary_string* is the null string, B2X returns a null string. If the number of binary digits in *binary_string* is not a multiple of four, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

Here are some examples:

```
B2X('11000011')   ->  'C3'
B2X('10111')      ->  '17'
B2X('101')        ->  '5'
B2X('1 1111 0000') ->  '1F0'
```

You can combine B2X with the functions X2D and X2C to convert a binary number into other forms. For example:

```
X2D(B2X('10111')) ->  '23'  /* decimal 23 */
```

14.3.9 CENTER/CENTRE
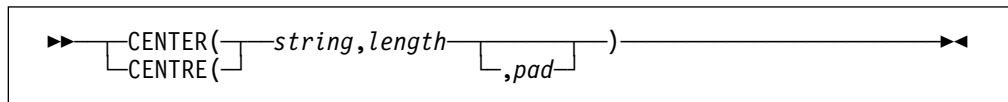
```
 ►►──┬─CENTER(─┬─string,length─┬───────┬──)──────────────────►◄
     └─CENTRE(─┘               └─,pad─┘
```

returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The *length* must be a positive whole number or zero. The default *pad* character is blank. If the string is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

Here are some examples:

```
CENTER(abc,7)               ->     '  ABC  '
CENTER(abc,8,'-')           ->     '--ABC---'
CENTRE('The blue sky',8)    ->     'e blue s'
CENTRE('The blue sky',7)    ->     'e blue '
```

Note: To avoid errors because of the difference between British and American spellings, this function can be called either CENTRE or CENTER.

14.3.10 COMPARE

```
 ►►──COMPARE(string1,string2─┬───────┬──)──────────────────►◄
                            └─,pad─┘
```
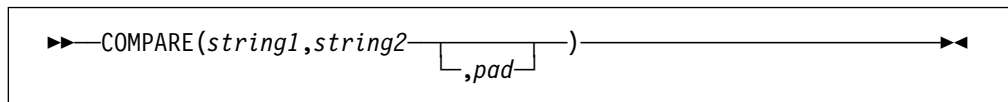
returns 0 if the strings, *string1* and *string2*, are identical. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with *pad* if necessary. The default *pad* character is a blank.

Here are some examples:

```
COMPARE('abc','abc')        ->     0
COMPARE('abc','ak')         ->     2
COMPARE('ab ','ab')         ->     0
COMPARE('ab ','ab',' ')     ->     0
COMPARE('ab ','ab','x')     ->     3
COMPARE('ab-- ','ab','-')   ->     5
```

14.3.11 CONDITION

```
 ►►──CONDITION(─┬──────────┬──)──────────────────►◄
               └─option─┘
```
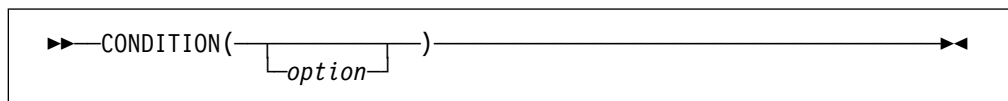
returns the condition information associated with the current trapped condition. (See Chapter 17, "Conditions and Condition Traps" on page 265 for a description of condition traps.) You can request the following pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

To select the information to return, use the following *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Condition name   returns the name of the current trapped condition.

Description   returns any descriptive string associated with the current trapped condition. See page 269 for the list of possible strings. If no description is available, returns a null string.

Instruction   returns either `CALL` or `SIGNAL`, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit *option*.

Status   returns the status of the current trapped condition. This can change during processing, and is either:

> `ON` - the condition is enabled

> `OFF` - the condition is disabled

> `DELAY` - any new occurrence of the condition is delayed or ignored.

If no condition has been trapped, then the CONDITION function returns a null string in all four cases.

Here are some examples:

```
CONDITION()            ->    'CALL'         /* perhaps */
CONDITION('C')         ->    'FAILURE'
CONDITION('I')         ->    'CALL'
CONDITION('D')         ->    'FailureTest'
CONDITION('S')         ->    'OFF'          /* perhaps */
```

Note: The CONDITION function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, after a subroutine called with CALL ON *trapname* has returned, the current trapped condition reverts to the condition that was current before the CALL took place (which may be none). CONDITION returns the values it returned before the condition was trapped.

14.3.12 COPIES

```
►►──COPIES(string,n)──────────────────────────────────────────►◄
```
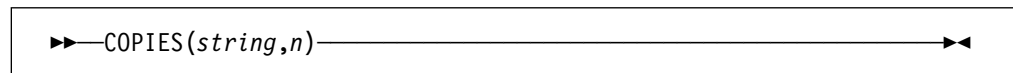
returns *n* concatenated copies of *string*. The *n* must be a positive whole number or zero.

Here are some examples:

```
COPIES('abc',3)    ->    'abcabcabc'
COPIES('abc',0)    ->    ''
```

14.3.13 C2D (Character to Decimal)

```
►►──C2D(string──────┬───)────────────────────────────────────►◄
                    └─,n─┘
```
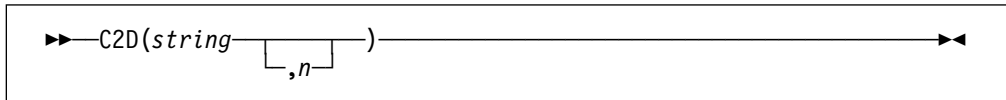
returns the decimal value of the binary representation of *string*. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify *n*, it is the length of the returned result. If you do not specify *n*, *string* is processed as an unsigned binary number.

If *string* is null, returns 0.

Here are some examples:

```
C2D('09'X)        ->          9
C2D('81'X)        ->        129
C2D('FF81'X)      ->      65409
C2D('')           ->          0
C2D('a')          ->        129     /*  EBCDIC  */
```

If you specify *n*, the string is taken as a signed number expressed in *n* characters. The number is positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. In both cases, it is converted to a whole number, which may, therefore, be negative. The *string* is padded on the left with '00'x characters (note, not "sign-extended"), or truncated on the left to *n* characters. This padding or truncation is as though RIGHT(*string*,*n*,'00'x) had been processed. If *n* is 0, C2D always returns 0.

Here are some examples:

```
C2D('81'X,1)      ->       -127
C2D('81'X,2)      ->        129
C2D('FF81'X,2)    ->       -127
C2D('FF81'X,1)    ->       -127
C2D('FF7F'X,1)    ->        127
C2D('F081'X,2)    ->      -3967
C2D('F081'X,1)    ->       -127
C2D('0031'X,0)    ->          0
```

Implementation maximum: The input string cannot have more than 250 characters that are significant in forming the final result. Leading sign characters ('00'x and 'FF'x) do not count toward this total.

14.3.14 C2X (Character to Hexadecimal)

```
►►──C2X(string)────────────────────────────────────────────────►◄
```
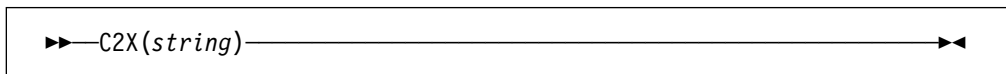
returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string. For example, on an EBCDIC system, C2X(1) returns F1 because the EBCDIC representation of the character 1 is 'F1'X.

The string returned uses uppercase alphabetics for the values A–F and does not include blanks. The *string* can be of any length. If *string* is null, returns a null string.

Here are some examples:

```
C2X('72s')      ->     'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X)    ->     '0123'   /* 'F0F1F2F3'X     in EBCDIC */
```

14.3.15 DATATYPE

```
►►──DATATYPE(string───────┬──────)──────────────────►◄
                          └─,type─┘
```
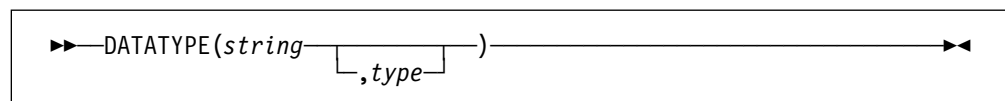
returns NUM if you specify only *string* and if *string* is a valid REXX number that can be added to 0 without error; returns CHAR if *string* is not a valid number.

If you specify *type*, returns 1 if *string* matches the type; otherwise returns 0. If *string* is null, the function returns 0 (except when *type* is X, which returns 1 for a null string). The following are valid *type*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored. Note that for the hexadecimal option, you must start your string specifying the name of the option with x rather than h.)

Alphanumeric   returns 1 if *string* contains only characters from the ranges a–z, A–Z, and 0–9.

Binary   returns 1 if *string* contains only the characters 0 or 1 or both.

C   returns 1 if *string* is a mixed SBCS/DBCS string.

Dbcs   returns 1 if *string* is a DBCS-only string enclosed by SO and SI bytes.

Lowercase   returns 1 if *string* contains only characters from the range a–z.

Mixed case   returns 1 if *string* contains only characters from the ranges a–z and A–Z.

Number   returns 1 if *string* is a valid REXX number.

Symbol   returns 1 if *string* contains only characters that are valid in REXX symbols. (See page 133.) Note that both uppercase and lowercase alphabetics are permitted.

Uppercase   returns 1 if *string* contains only characters from the range A–Z.

Whole number   returns 1 if *string* is a REXX whole number under the current setting of NUMERIC DIGITS.

he**X**adecimal   returns 1 if *string* contains only characters from the ranges a–f, A–F, 0–9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if *string* is a null string, which is a valid hexadecimal string.

Here are some examples:

```
DATATYPE(' 12 ')          ->    'NUM'
DATATYPE('')              ->    'CHAR'
DATATYPE('123*')          ->    'CHAR'
DATATYPE('12.3','N')      ->    1
DATATYPE('12.3','W')      ->    0
DATATYPE('Fred','M')      ->    1
DATATYPE('','M')          ->    0
DATATYPE('Fred','L')      ->    0
DATATYPE('?20K','s')      ->    1
DATATYPE('BCd3','X')      ->    1
DATATYPE('BC d3','X')     ->    1
```

Note: The DATATYPE function tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC).

14.3.16 DATE
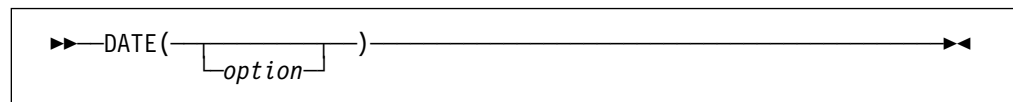
returns, by default, the local date in the format: *dd mon yyyy* (day month year—for example, 13 Mar 1992), with no leading zero or blank on the day. If the active language has an abbreviated form of the month name, then it is used (for example, Jan, Feb, and so on).

You can use the following *option*s to obtain specific formats. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Base   returns the number of complete days (that is, not including the current day) since and including the base date, 1 January 0001, in the format: *dddddd* (no leading zeros or blanks). The expression DATE('B')//7 returns a number in the range 0–6 that corresponds to the current day of the week, where 0 is Monday and 6 is Sunday.

Thus, this function can be used to determine the day of the week independent of the national language in which you are working.

Note: The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Century   returns the number of days, including the current day, since and including January 1 of the last year that is a multiple of 100 in the format: *ddddd* (no leading zeros). Example: A call to DATE(C) on 13 March 1992 returns 33675, the number of days from 1 January 1900 to 13 March 1992. Similarly, a call to DATE(C) on 2 January 2000 returns 2, the number of days from 1 January 2000 to 2 January 2000.

Days   returns the number of days, including the current day, so far in this year in the format: *ddd* (no leading zeros or blanks).

European   returns date in the format: dd/mm/yy.

Julian   returns date in the format: yyddd.

Month   returns full English name of the current month, for example, `August`.

Normal   returns date in the format: *dd mon yyyy*. **This is the default**.

Ordered   returns date in the format: *yy/mm/dd* (suitable for sorting, and so forth).

Standard   returns date in the format: *yyyymmdd* (suitable for sorting, and so forth).

Usa   returns date in the format: *mm/dd/yy*.

Weekday   returns the English name for the day of the week, in mixed case, for example, `Tuesday`.

Here are some examples, assuming today is 13 March 1992:

```
DATE()        ->    '13 Mar 1992'
DATE('B')     ->    727269
DATE('C')     ->    33675
DATE('D')     ->    73
DATE('E')     ->    '13/03/92'
DATE('J')     ->    92073
DATE('M')     ->    'March'
DATE('N')     ->    '13 Mar 1992'
DATE('O')     ->    '92/03/13'
DATE('S')     ->    '19920313'
DATE('U')     ->    '03/13/92'
DATE('W')     ->    'Friday'
```

Note: The first call to DATE or TIME in one clause causes a time stamp to be made that is then used for *all* calls to these functions in that clause. Therefore, multiple calls to any of the DATE or TIME functions or both in a single expression or clause are guaranteed to be consistent with each other.

14.3.17 DBCS (Double-Byte Character Set Functions)

The following are all part of DBCS processing functions. See page 449.

DBADJUST	DBRIGHT	DBUNBRACKET
DBBRACKET	DBRLEFT	DBVALIDATE
DBCENTER	DBRRIGHT	DBWIDTH
DBCJUSTIFY	DBTODBCS	
DBLEFT	DBTOSBCS	

14.3.18 DELSTR (Delete String)

```
►►──DELSTR(string,n─────┬─────────┬──)──────────────────────►◄
                        └─,length─┘
```
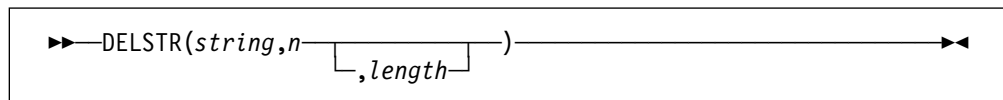
returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the length of *string*, the function returns *string* unchanged.

Here are some examples:

```
DELSTR('abcd',3)      ->    'ab'
DELSTR('abcde',3,2)   ->    'abe'
DELSTR('abcde',6)     ->    'abcde'
```

14.3.19 DELWORD (Delete Word)

```
►►──DELWORD(string,n──────────)────────────────►◄
                    └─,length─┘
```
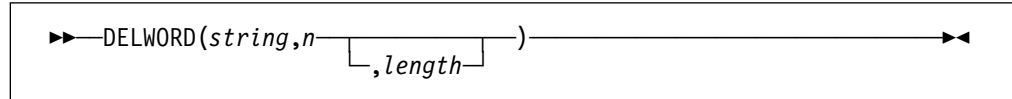
returns *string* after deleting the substring that starts at the *n*th word and is of *length* blank-delimited words. If you omit *length*, or if *length* is greater than the number of words from *n* to the end of *string*, the function deletes the remaining words in *string* (including the *n*th word). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Here are some examples:

```
DELWORD('Now is the  time',2,2)  ->  'Now time'
DELWORD('Now is the time ',3)    ->  'Now is '
DELWORD('Now is the  time',5)    ->  'Now is the  time'
DELWORD('Now is  the time',3,1) ->  'Now is  time'
```

14.3.20 DIGITS

```
►►──DIGITS()─────────────────────────────────────►◄
```
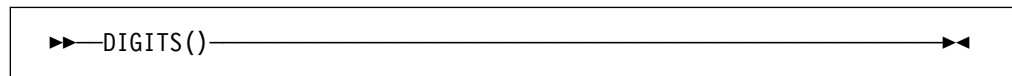
returns the current setting of NUMERIC DIGITS. See the NUMERIC instruction on page 176 for more information.

Here is an example:

```
DIGITS()   ->   9    /* by default */
```

14.3.21 D2C (Decimal to Character)
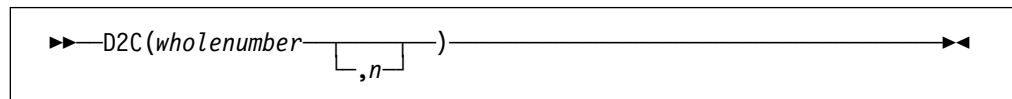
```
►►──D2C(wholenumber──────)────────────────────►◄
                   └─,n─┘
```

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to binary. If you specify *n*, it is the length of the final result in characters; after conversion, the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, then the result is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n, wholenumber* must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading '00'x characters.

Here are some examples:

```
D2C(9)        ->   ' '    /* '09'x is unprintable in EBCDIC      */
D2C(129)      ->   'a'    /* '81'x is an EBCDIC 'a'              */
D2C(129,1)    ->   'a'    /* '81'x is an EBCDIC 'a'              */
D2C(129,2)    ->   ' a'   /* '0081'x is EBCDIC ' a'             */
D2C(257,1)    ->   ' '    /* '01'x is unprintable in EBCDIC      */
D2C(-127,1)   ->   'a'    /* '81'x is EBCDIC 'a'                */
D2C(-127,2)   ->   ' a'   /* 'FF'x is unprintable EBCDIC;       */
                          /* '81'x is EBCDIC 'a'                */
D2C(-1,4)     ->   '    ' /* 'FFFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)     ->   ''     /* '' is a null string               */
```

Implementation maximum: The output string may not have more than 250 significant characters, though a longer result is possible if it has additional leading sign characters ('00'x and 'FF'x).

14.3.22 D2X (Decimal to Hexadecimal)
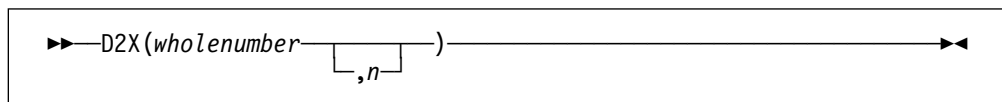
```
►►──D2X(wholenumber─┬──────┬─)──────────────────────►◄
                    └─,n─┘
```

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabetics for the values A–F and does not include blanks.

If you specify *n*, it is the length of the final result in characters; after conversion the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n*, *wholenumber* must be a positive whole number or zero, and the returned result has no leading zeros.

Here are some examples:

```
D2X(9)        ->   '9'
D2X(129)      ->   '81'
D2X(129,1)    ->   '1'
D2X(129,2)    ->   '81'
D2X(129,4)    ->   '0081'
D2X(257,2)    ->   '01'
D2X(-127,2)   ->   '81'
D2X(-127,4)   ->   'FF81'
D2X(12,0)     ->   ''
```

Implementation maximum: The output string may not have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

14.3.23 ERRORTEXT

```
►►──ERRORTEXT(n)────────────────────────────────►◄
```

returns the REXX error message associated with error number *n*. The *n* must be in
the range `0–99`, and any other value is an error. Returns the null string if *n* is in
the allowed range but is not a defined REXX error number. See Appendix A,
"Error Numbers and Messages" on page 429 for a complete description of error
numbers and messages.

Here are some examples:

```
ERRORTEXT(16)    ->     'Label not found'
ERRORTEXT(60)    ->     ''
```

14.3.24 EXTERNALS

```
►►──EXTERNALS()─────────────────────────────────►◄
```

always returns a 0. For example:

```
EXTERNALS()    ->    0    /* Always */
```

The EXTERNALS function returns the number of elements in the terminal input
buffer (system external event queue). In CICS, there is no equivalent buffer.
Therefore, in the CICS implementation of REXX, the EXTERNALS function always
returns a 0.

14.3.25 FIND

WORDPOS is the preferred built-in function for this type of word search. See page
234 for a complete description.

```
►►──FIND(string,phrase)─────────────────────────►◄
```

returns the word number of the first word of *phrase* found in *string* or returns `0` if
phrase is not found or if there are no words in *phrase*. The *phrase* is a sequence
of blank-delimited words. Multiple blanks between words in *phrase* or *string* are
treated as a single blank for the comparison.

Here are some examples:

```
FIND('now is the time','is the time')    ->    2
FIND('now is  the time','is    the')     ->    2
FIND('now is  the time','is  time ')     ->    0
```

14.3.26 FORM

```
►►──FORM()──────────────────────────────────────►◄
```

returns the current setting of NUMERIC FORM. See the NUMERIC instruction on
page 176 for more information.

Here is an example:

```
FORM()    ->    'SCIENTIFIC'  /* by default */
```

14.3.27 FORMAT

expnt:
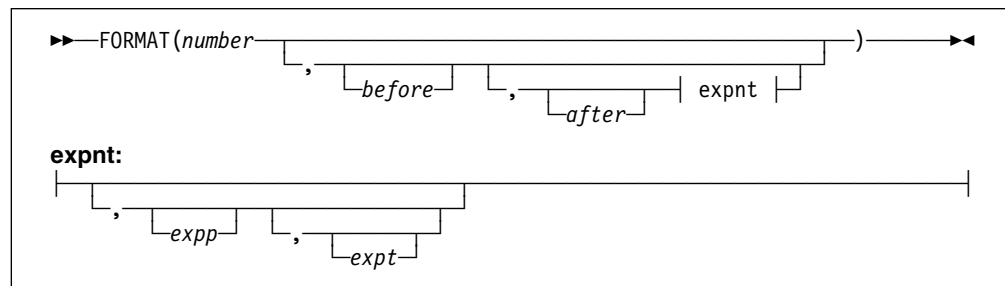
returns *number*, rounded and formatted.

The *number* is first rounded according to standard REXX rules, just as though the operation `number+0` had been carried out. The result is precisely that of this operation if you specify only *number*. If you specify any other options, the *number* is formatted as follows.

The *before* and *after* options describe how many characters are used for the integer and decimal parts of the result, respectively. If you omit either or both of these, the number of characters used for that part is as needed.

If *before* is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If *before* is larger than needed for that part, the number is padded on the left with blanks. If *after* is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying `0` causes the number to be rounded to an integer.

Here are some examples:

```
FORMAT('3',4)              ->    '   3'
FORMAT('1.73',4,0)         ->    '   2'
FORMAT('1.73',4,3)         ->    '   1.730'
FORMAT('-.76',4,1)         ->    ' -0.8'
FORMAT('3.03',4)           ->    '   3.03'
FORMAT(' - 12.73',,4)      ->    '-12.7300'
FORMAT(' - 12.73')         ->    '-12.73'
FORMAT('0.000')            ->    '0'
```

The first three arguments are as described previously. In addition, *expp* and *expt* control the exponent part of the result, which, by default, is formatted according to the current NUMERIC settings of DIGITS and FORM. The *expp* sets the number of places for the exponent part; the default is to use as many as needed (which may be zero). The *expt* sets the trigger point for use of exponential notation. The default is the current setting of NUMERIC DIGITS.

If *expp* is `0`, no exponent is supplied, and the number is expressed in *simple* form with added zeros as necessary. If *expp* is not large enough to contain the exponent, an error results.

If the number of places needed for the integer or decimal part exceeds *expt* or twice *expt*, respectively, exponential notation is used. If *expt* is `0`, exponential

notation is always used unless the exponent would be 0. (If *expp* is 0, this
overrides a 0 value of *expt*.) If the exponent would be 0 when a nonzero *expp* is
specified, then *expp*+2 blanks are supplied for the exponent part of the result. If
the exponent would be 0 and *expp* is not specified, simple form is used. The *expp*
must be less than 10, but there is no limit on the other arguments.

Here are some examples:

```
FORMAT('12345.73',,,2,2)   ->    '1.234573E+04'
FORMAT('12345.73',,3,,0)   ->    '1.235E+4'
FORMAT('1.234573',,3,,0)   ->    '1.235'
FORMAT('12345.73',,,3,6)   ->    '12345.73'
FORMAT('1234567e5',,3,0)   ->    '123456700000.000'
```

14.3.28 FUZZ

```
►►──FUZZ()──────────────────────────────────────►◄
```
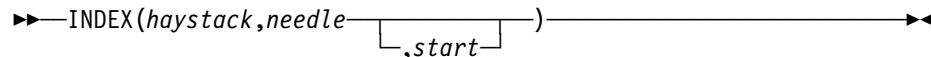
returns the current setting of NUMERIC FUZZ. See the NUMERIC instruction on
page 176 for more information.

Here is an example:

```
FUZZ()    ->    0     /* by default */
```

14.3.29 INDEX

POS is the preferred built-in function for obtaining the position of one string in
another. See page 224 for a complete description.

```
►►──INDEX(haystack,needle──┬──────────┬──)──────────►◄
                           └─,start──┘
```
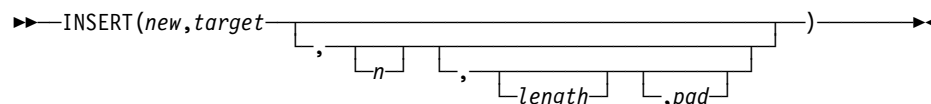
returns the character position of one string, *needle*, in another, *haystack*, or returns
0 if the string *needle* is not found or is a null string. By default the search starts at
the first character of *haystack* (*start* has the value 1). You can override this by
specifying a different *start* point, which must be a positive whole number.

Here are some examples:

```
INDEX('abcdef','cd')      ->    3
INDEX('abcdef','xd')      ->    0
INDEX('abcdef','bc',3)    ->    0
INDEX('abcabc','bc',3)    ->    5
INDEX('abcabc','bc',6)    ->    0
```

14.3.30 INSERT

```
►►──INSERT(new,target──┬──────────────────────────────────┬──)──►◄
                       └─,─┬───┬─┬──────────────────────┬─┘
                           └─n─┘ └─,─┬────────┬─┬──────┬─┘
                                     └─length─┘ └─,pad─┘
```

inserts the string *new*, padded or truncated to length *length*, into the string *target* after the *n*th character. The default value for *n* is 0, which means insert before the beginning of the string. If specified, *n* and *length* must be positive whole numbers or zero. If *n* is greater than the length of the target string, padding is added before the string *new* also. The default value for *length* is the length of *new*. If *length* is less than the length of the string *new*, then INSERT truncates *new* to length *length*. The default *pad* character is a blank.

Here are some examples:

```
INSERT(' ','abcdef',3)        ->    'abc def'
INSERT('123','abc',5,6)       ->    'abc  123   '
INSERT('123','abc',5,6,'+')   ->    'abc++123+++'
INSERT('123','abc')           ->    '123abc'
INSERT('123','abc',,5,'-')    ->    '123--abc'
```

14.3.31 JUSTIFY

```
►►──JUSTIFY(string,length──────────)────────────────►◄
                        └─,pad─┘
```
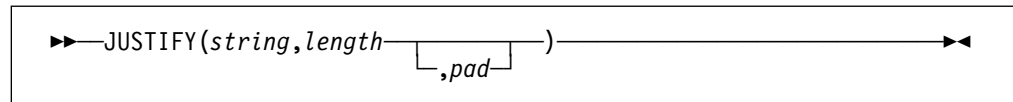
returns *string* formatted by adding *pad* characters between blank-delimited words to justify to both margins. This is done to width *length* (*length* must be a positive whole number or zero). The default *pad* character is a blank.

The first step is to remove extra blanks as though SPACE(*string*) had been run (that is, multiple blanks are converted to single blanks, and leading and trailing blanks are removed). If *length* is less than the width of the changed string, the string is then truncated on the right and any trailing blank is removed. Extra *pad* characters are then added evenly from left to right to provide the required length, and the *pad* character replaces the blanks between words.

Here are some examples:

```
JUSTIFY('The blue sky',14)    ->    'The  blue  sky'
JUSTIFY('The blue sky',8)     ->    'The blue'
JUSTIFY('The blue sky',9)     ->    'The  blue'
JUSTIFY('The blue sky',9,'+') ->    'The++blue'
```

14.3.32 LASTPOS (Last Position)

```
►►──LASTPOS(needle,haystack──────────)────────────────►◄
                          └─,start─┘
```
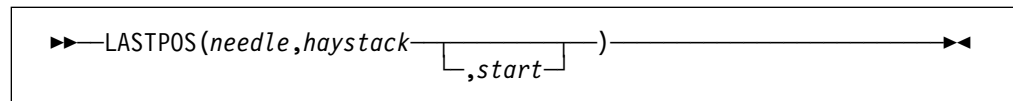
returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns 0 if *needle* is the null string or is not found. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying *start*, the point at which the backward scan starts. *start* must be a positive whole number and defaults to LENGTH(*haystack*) if larger than that value or omitted.

Here are some examples:

```
LASTPOS(' ','abc def ghi')      ->    8
LASTPOS(' ','abcdefghi')        ->    0
LASTPOS('xy','efgxyz')          ->    4
LASTPOS(' ','abc def ghi',7)    ->    4
```

14.3.33 LEFT

```
►►──LEFT(string,length──────────)──────────────────────────►◄
                      └─,pad─┘
```

returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a positive whole number or zero. The LEFT function is exactly equivalent to:

```
►►──SUBSTR(string,1,length──────────)──►◄
                        └─,pad─┘
```

Here are some examples:

```
LEFT('abc d',8)         ->    'abc d   '
LEFT('abc d',8,'.')     ->    'abc d...'
LEFT('abc  def',7)      ->    'abc  de'
```

14.3.34 LENGTH

```
►►──LENGTH(string)─────────────────────────────────────────►◄
```

returns the length of *string*.

Here are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

14.3.35 LINESIZE

```
►►──LINESIZE()─────────────────────────────────────────────►◄
```

returns the current terminal line width (the point at which the language processor breaks lines displayed using the SAY instruction). Returns a default value of 80 if:

- No terminal is attached.
- Output is being redirected.

Note: To determine if a terminal is attached, specify the REXX/CICS command SCRNINFO (see page 420). The values for screen height and screen width are 0 if there is no terminal attached.

14.3.36 MAX (Maximum)

```
►►──MAX(──┬─┬─────┬─number─┬──)────────────────────────────────►◄
              └─',──┘
```

returns the largest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

```
MAX(12,6,7,9)                                                    ->     12
MAX(17.3,19,17.03)                                              ->     19
MAX(-7,-3,-4.3)                                                 ->     -3
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21)) ->     21
```

Implementation maximum: You can specify up to 20 *number*s, and can nest calls to MAX if more arguments are needed.

14.3.37 MIN (Minimum)
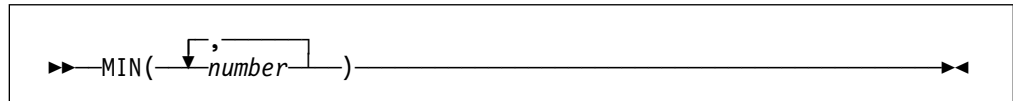
```
►►──MIN(──┬─┬─────┬─number─┬──)────────────────────────────────►◄
              └─',──┘
```

returns the smallest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

```
MIN(12,6,7,9)                                                    ->      6
MIN(17.3,19,17.03)                                             ->    17.03
MIN(-7,-3,-4.3)                                                ->     -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1)) ->      1
```

Implementation maximum: You can specify up to 20 *number*s, and can nest calls to MIN if more arguments are needed.

14.3.38 OVERLAY

```
►►──OVERLAY(new,target──┬──────────────────────────────────┬──)──►◄
                        └─,─┬────┬─┬──────────────────────┬─┘
                            └─n──┘ └─,─┬────────┬─┬──────┬┘
                                       └─length─┘ └─,pad─┘
```
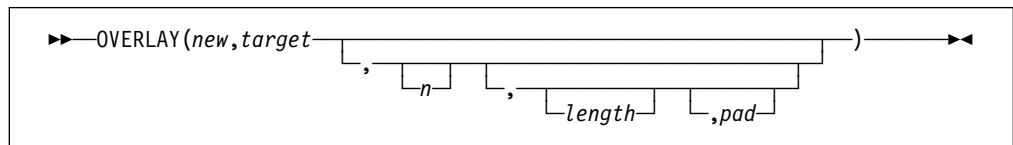
returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. (The overlay may extend beyond the end of the original *target* string.) If you specify *length*, it must be a positive whole number or zero. The default value for *length* is the length of *new*. If *n* is greater than the length of the target string, padding is added before the *new* string. The default *pad* character is a blank, and the default value for *n* is 1. If you specify *n*, it must be a positive whole number.

Here are some examples:

```
OVERLAY(' ','abcdef',3)          ->     'ab def'
OVERLAY('.','abcdef',3,2)        ->     'ab. ef'
OVERLAY('qq','abcd')             ->     'qqcd'
OVERLAY('qq','abcd',4)           ->     'abcqq'
OVERLAY('123','abc',5,6,'+')     ->     'abc+123+++'
```

14.3.39 POS (Position)

```
►►──POS(needle,haystack───────────────)──────────────────►◄
                        └─,start─┘
```
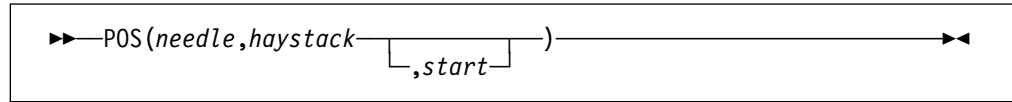
returns the position of one string, *needle*, in another, *haystack*. (See also the INDEX and LASTPOS functions.) Returns 0 if *needle* is the null string or is not found or if *start* is greater than the length of *haystack*. By default the search starts at the first character of *haystack* (that is, the value of *start* is 1). You can override this by specifying *start* (which must be a positive whole number), the point at which the search starts.

Here are some examples:

```
POS('day','Saturday')       ->    6
POS('x','abc def ghi')      ->    0
POS(' ','abc def ghi')      ->    4
POS(' ','abc def ghi',5)    ->    8
```

14.3.40 QUEUED

```
►►──QUEUED()──────────────────────────────────────────────►◄
```

returns the number of lines remaining in the external data queue when the function is called. If no lines are remaining, a PULL or PARSE PULL reads from the terminal input buffer. If no terminal input is waiting, this causes a console read.

Here is an example:

```
QUEUED()     ->    5    /* Perhaps */
```

14.3.41 RANDOM

```
►►──RANDOM(─────────────────────────────)──────────────────►◄
          ├─max─────────────────┤
          ├─min,──────┬─────┬───┤
          └─,──────┘  └─max─┘ └─,seed─┘
```
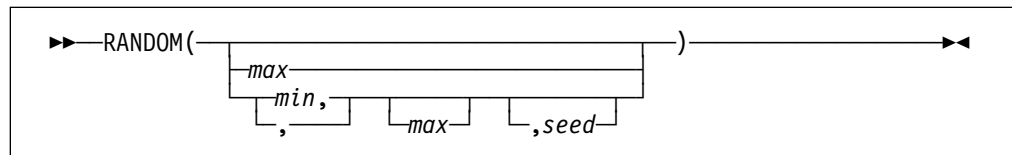
returns a quasi-random nonnegative whole number in the range *min* to *max* inclusive. If you specify *max* or *min* or both, *max* minus *min* cannot exceed 100000. The *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument, as described in Note 1 on page 225. This *seed* must be a positive whole number ranging from 0 to 999999999.

Here are some examples:

```
RANDOM()          ->     305
RANDOM(5,8)       ->       7
RANDOM(2)         ->       0  /*  0  to 2    */
RANDOM(,,1983)    ->     123  /* reproducible */
```

Notes:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

```
sequence = RANDOM(1,6,12345)  /* any number would */
                              /* do for a seed    */
do 39
   sequence = sequence RANDOM(1,6)
   end
say sequence
```

The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence. If you do not supply a *seed*, the first time RANDOM is called, the microsecond field of the time-of-day clock is used as the *seed*; and hence your program almost always gives different results each time it is run.

2. The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

14.3.42 REVERSE

```
►►──REVERSE(string)──────────────────────────────────────►◄
```
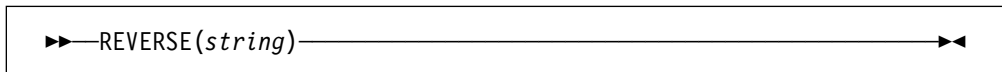
returns *string*, swapped end for end.

Here are some examples:

```
REVERSE('ABc.')   ->     '.cBA'
REVERSE('XYZ ')   ->     ' ZYX'
```

14.3.43 RIGHT
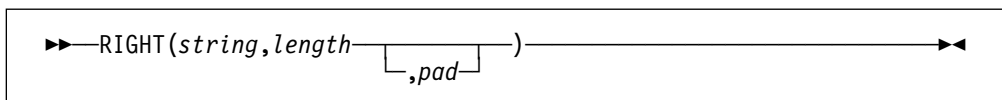
```
►►──RIGHT(string,length──────────)────────────────────────►◄
                       └─,pad─┘
```

returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. The *length* must be a positive whole number or zero.

Here are some examples:

```
RIGHT('abc  d',8)    ->     '  abc  d'
RIGHT('abc def',5)   ->     'c def'
RIGHT('12',5,'0')    ->     '00012'
```

14.3.44 SIGN

```
►►──SIGN(number)─────────────────────────────────────────────►◄
```
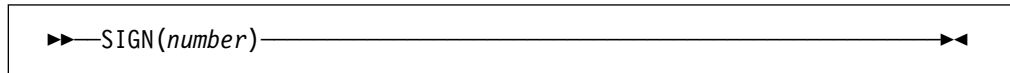
returns a number that indicates the sign of *number*. The *number* is first rounded according to standard REXX rules, just as though the operation `number+0` had been carried out. Returns `-1` if *number* is less than `0`; returns `0` if it is `0`; and returns `1` if it is greater than `0`.

Here are some examples:

```
SIGN('12.3')       ->     1
SIGN(' -0.307')    ->    -1
SIGN(0.0)          ->     0
```

14.3.45 SOURCELINE

```
►►──SOURCELINE(──┬───┬──)─────────────────────────────────────►◄
                 └─n─┘
```
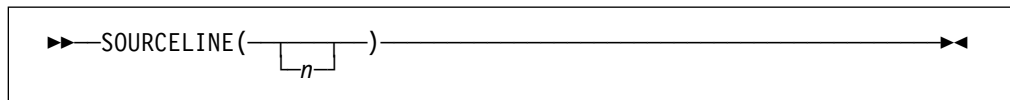
returns the line number of the final line in the program if you omit *n*, or returns the *n*th line in the program if you specify *n*. If specified, *n* must be a positive whole number and must not exceed the number of the final line in the program.

Here are some examples:

```
SOURCELINE()    ->    10
SOURCELINE(1)   ->    '/* This is a 10-line REXX program */'
```

14.3.46 SPACE

```
►►──SPACE(string──┬──────────────────────┬──)────────────────►◄
                  └─,──┬───┬──┬──────┬────┘
                       └─n─┘  └─,pad─┘
```
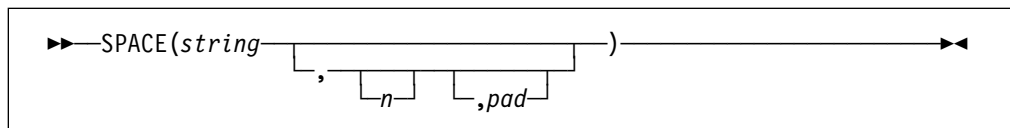
returns the blank-delimited words in *string* with *n pad* characters between each word. If you specify *n*, it must be a positive whole number or zero. If it is `0`, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is `1`, and the default *pad* character is a blank.

Here are some examples:

```
SPACE('abc  def  ')         ->     'abc def'
SPACE('  abc def',3)        ->     'abc   def'
SPACE('abc  def  ',1)       ->     'abc def'
SPACE('abc  def  ',0)       ->     'abcdef'
SPACE('abc  def  ',2,'+')   ->     'abc++def'
```

14.3.47 STORAGE

See section 14.4, "External Functions Provided in REXX/CICS" on page 237.

14.3.48 STRIP

```
►►──STRIP(string──┬───────────────────────┬──)──────────►◄
                  └─,──┬────────┬──┬───────┬─┘
                       └─option─┘  └─,char─┘
```
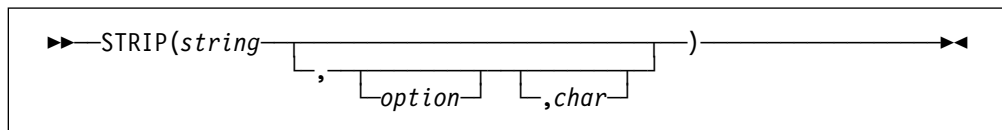
returns *string* with leading or trailing characters or both removed, based on the *option* you specify. The following are valid *option*s. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Both   removes both leading and trailing characters from *string*. This is the default.

Leading   removes leading characters from *string*.

Trailing   removes trailing characters from *string*.

The third argument, *char*, specifies the character to be removed, and the default is a blank. If you specify *char*, it must be exactly one character long.

Here are some examples:

```
STRIP('  ab c  ')        ->    'ab c'
STRIP('  ab c  ','L')    ->    'ab c  '
STRIP('  ab c  ','t')    ->    '  ab c'
STRIP('12.7000',,0)      ->    '12.7'
STRIP('0012.700',,0)     ->    '12.7'
```

14.3.49 SUBSTR (Substring)

```
►►──SUBSTR(string,n──┬───────────────────────┬──)──────────►◄
                     └─,──┬────────┬──┬──────┬─┘
                          └─length─┘  └─,pad─┘
```
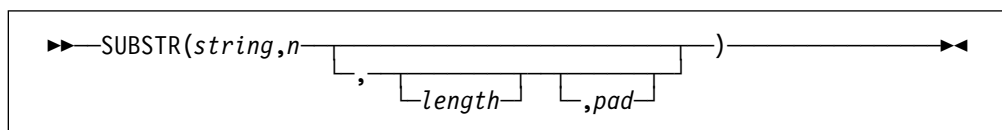
returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. The *n* must be a positive whole number. If *n* is greater than `LENGTH(string)`, then only pad characters are returned.

If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

Here are some examples:

```
SUBSTR('abc',2)          ->    'bc'
SUBSTR('abc',2,4)        ->    'bc  '
SUBSTR('abc',2,6,'.')    ->    'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string. See also the LEFT and RIGHT functions.

14.3.50 SUBWORD

```
►►──SUBWORD(string,n──────────)──────────────────────►◄
                    └─,length─┘
```
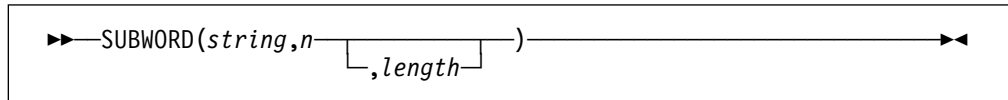
returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

Here are some examples:

```
SUBWORD('Now is the  time',2,2)   ->    'is the'
SUBWORD('Now is the  time',3)     ->    'the  time'
SUBWORD('Now is the  time',5)     ->    ''
```

14.3.51 SYMBOL

```
►►──SYMBOL(name)────────────────────────────────────►◄
```
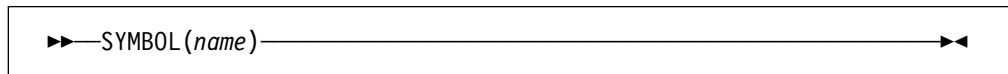
returns the state of the symbol named by *name*. Returns BAD if *name* is not a valid REXX symbol. Returns VAR if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

As with symbols in REXX expressions, lowercase characters in *name* are translated to uppercase and substitution in a compound name occurs if possible.

Note: You should specify *name* as a literal string (or it should be derived from an expression) to prevent substitution before it is passed to the function.

Here are some examples:

```
/* following: Drop A.3;  J=3 */
SYMBOL('J')       ->    'VAR'
SYMBOL(J)         ->    'LIT' /* has tested "3"     */
SYMBOL('a.j')     ->    'LIT' /* has tested A.3     */
SYMBOL(2)         ->    'LIT' /* a constant symbol  */
SYMBOL('*')       ->    'BAD' /* not a valid symbol */
```

14.3.52 TIME

```
►►──TIME(────────────)──────────────────────────────►◄
        └─option─┘
```
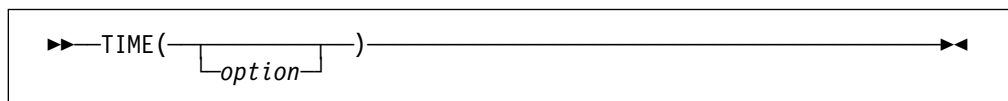
returns the local time in the 24-hour clock format: hh:mm:ss (hours, minutes, and seconds) by default, for example, 04:41:37.

You can use the following *option*s to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Civil   returns the time in Civil format: hh:mmxx. The hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters am or pm. This distinguishes times in the morning (12 midnight through 11:59 a.m.—appearing as 12:00am through 11:59am) from noon and afternoon (12 noon through 11:59 p.m.—appearing as 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

Elapsed   returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Hours   returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of 0).

Long   returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds). The first eight characters of the result follow the same rules as for the Normal form, and the fractional part is always six digits.

Minutes   returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of 0).

Normal   returns the time in the default format hh:mm:ss, as described previously. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59. All these are always two digits. Any fractions of seconds are ignored (times are never rounded up). **This is the default**.

Reset   returns sssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

Seconds   returns up to five characters giving the number of seconds since midnight in the format: sssss (no leading zeros or blanks, except for a result of 0).

Here are some examples, assuming that the time is 4:54 p.m.:

```
TIME()        ->   '16:54:22'
TIME('C')     ->   '4:54pm'
TIME('H')     ->   '16'
TIME('L')     ->   '16:54:22.123456'   /* Perhaps */
TIME('M')     ->   '1014'              /* 54 + 60*16 */
TIME('N')     ->   '16:54:22'
TIME('S')     ->   '60862'  /* 22 + 60*(54+60*16) */
```

The elapsed-time clock:

You can use the TIME function to measure real (elapsed) time intervals. On the first call in a program to TIME('E') or TIME('R'), the elapsed-time clock is started, and either call returns 0. From then on, calls to TIME('E') and to TIME('R') return the elapsed time since that first call or since the last call to TIME('R').

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

```
time('E')    ->    0         /* The first call */
/* pause of one second here */
time('E')    ->    1.002345  /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.004690  /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.002345  /* or thereabouts */
```

Note: See the note under DATE about consistency of times within a single clause. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single clause always return the same result. For the same reason, the interval between two usual TIME/DATE results may be calculated exactly using the elapsed-time clock.

Implementation maximum: If the number of seconds in the elapsed time exceeds nine digits (equivalent to over 31.6 years), an error results.

14.3.53 TRACE

```
►►──TRACE(──┬────────┬──)──────────────────────────────►◄
            └─option─┘
```
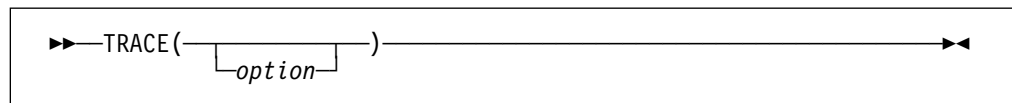
returns trace actions currently in effect and, optionally, alters the setting.

If you specify *option*, it selects the trace setting. It must be one of the valid prefixes ? or ! or one of the alphabetic character options associated with the TRACE instruction (that is, starting with A, C, E, F, I, L, N, O, or S) or both. (See the TRACE instruction on page 195 for full details.)

Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active. Also unlike the TRACE instruction, *option* cannot be a number.

Here are some examples:

```
TRACE()      ->    '?R' /* maybe */
TRACE('O')   ->    '?R' /* also sets tracing off    */
TRACE('?I')  ->    'O'  /* now in interactive debug */
```

14.3.54 TRANSLATE

```
►►──TRANSLATE(string──────────────────────────────────────)──►◄
                    │                  │                  │
                    └─┬────────┬─┘ └─┬────────┬─┘ └──,pad──┘
                      └─tableo─┘   , └─tablei─┘
```
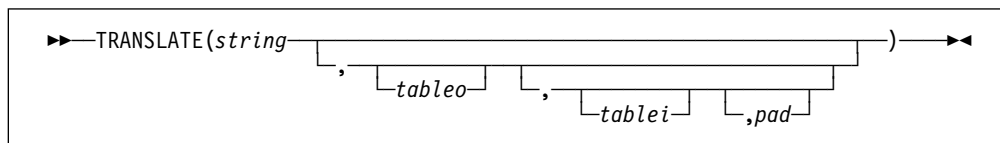
returns *string* with each character translated to another character or unchanged. You can also use this function to reorder the characters in *string*.

The output table is *tableo* and the input translation table is *tablei*. TRANSLATE searches *tablei* for each character in *string*. If the character is found, then the corresponding character in *tableo* is used in the result string; if there are duplicates in *tablei*, the first (leftmost) occurrence is used. If the character is not found, the original character in *string* is used. The result string is always the same length as *string*.

The tables can be of any length. If you specify neither translation table and omit *pad*, *string* is simply translated to uppercase (that is, lowercase `a–z` to uppercase `A–Z`), but, if you include *pad*, the language processor translates the entire string to *pad* characters. *tablei* defaults to `XRANGE('00'x,'FF'x)`, and *tableo* defaults to the null string and is padded with *pad* or truncated as necessary. The default *pad* is a blank.

Here are some examples:

```
TRANSLATE('abcdef')                     ->     'ABCDEF'
TRANSLATE('abbc','&','b')               ->     'a&&c'
TRANSLATE('abcdef','12','ec')           ->     'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')     ->     '12..ef'
TRANSLATE('APQRV',,'PR')                ->     'A Q V'
TRANSLATE('APQRV',XRANGE('00'X,'Q'))    ->     'APQ  '
TRANSLATE('4123','abcd','1234')         ->     'dabc'
```

Note: The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

14.3.55 TRUNC (Truncate)

```
►►──TRUNC(number──┬──────┬──)────────────────────────────►◄
                  └─,n─┘
```
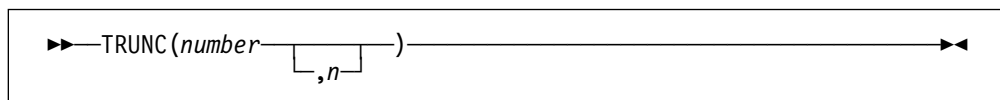
returns the integer part of *number* and *n* decimal places. The default *n* is `0` and returns an integer with no decimal point. If you specify *n*, it must be a positive whole number or zero. The *number* is first rounded according to standard REXX rules, just as though the operation `number+0` had been carried out. The number is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

Here are some examples:

```
TRUNC(12.3)          ->    12
TRUNC(127.09782,3)   ->    127.097
TRUNC(127.1,3)       ->    127.100
TRUNC(127,2)         ->    127.00
```

Note: The *number* is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

14.3.56 USERID

```
►►──USERID()───────────────────────────────────────────►◄
```

returns the CICS signon user ID if the user is signed onto CICS or the CICS region default user ID (if one was specified by the CICS systems programmer). User IDs are padded on the right with blanks so that the returned value is always eight bytes long.

Here is an example:

```
USERID()    ->    'ARTHUR' /* Maybe */
```

14.3.57 VALUE

```
►►──VALUE(name─┬──────────────────────────────┬──)──────►◄
               │  ┌────────────────────────┐   │
               └──┴──,─┬─────────┬──┬───────────┬──┘
                       └─newvalue─┘  └─,selector─┘
```

returns the value of the symbol that *name* (often constructed dynamically) represents and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment, however, if you want to specify *selector* the value must be RLS. If the selector of RLS is specified, then the variable operated on is a REXX List System (RLS) variable, rather than a REXX variable. If you use the function to refer to REXX variables, then *name* must be a valid REXX symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in *name* are translated to uppercase. Substitution in a compound name (see section 12.4.3, "Compound Symbols" on page 144) occurs if possible.

If you specify *newvalue*, then the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of *name* as it was before the new assignment.

Here are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)     ->  'A3' /* looks up A3     */
VALUE('a'k||k)  ->  '7'  /* looks up A33    */
VALUE('fred')   ->  'K'  /* looks up FRED   */
VALUE(fred)     ->  '3'  /* looks up K      */
VALUE(fred,5)   ->  '3'  /* looks up K and  */
                         /* then sets K=5   */
VALUE(fred)     ->  '5'  /* looks up K      */
VALUE('LIST.'k) ->  'Hi' /* looks up LIST.5 */
```

The following example returns the VALUE of the REXX variable FRED that has been stored in an RLS variable.

```
/* REXX EXEC - ASSIGN FIND VALUE OF FRED  */
FRED = 7
'RLS VARPUT FRED \USERS\userid\'
X = VALUE(FRED,,RLS)
SAY X
/* X now = 7                              */
```

Notes:

1. If the VALUE function refers to an uninitialized REXX variable then the default value of the variable is always returned; the NOVALUE condition is not raised. A reference to RLS variables never raises NOVALUE.

2. If you specify the *name* as a single literal string and omit *newvalue* and *selector*, the symbol is a constant and so the string between the quotation marks can usually replace the whole function call. (For example, fred=VALUE('k'); is identical with the assignment fred=k;, unless the NOVALUE condition is being trapped. See Chapter 17, "Conditions and Condition Traps" on page 265.)

14.3.58 VERIFY

```
►►──VERIFY(string,reference──────────────────────)──────►◄
                          └─,──┬────────┬──┬────────┬─┘
                               └─option─┘  └─,start─┘
```
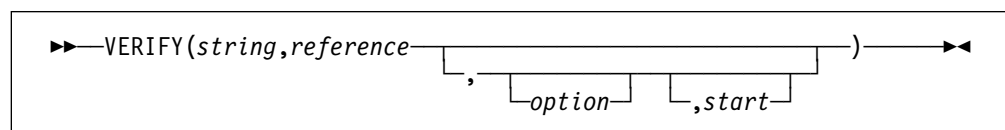
returns a number that, by default, indicates whether *string* is composed only of characters from *reference*; returns 0 if all characters in *string* are in *reference*, or returns the position of the first character in *string* **not** in *reference*.

The *option* can be either Nomatch (the default) or Match. (Only the capitalized and highlighted letter is needed. All characters following it are ignored, and it can be in upper- or lowercase, as usual.) If you specify Match, the function returns the position of the first character in *string* that **is** in *reference*, or returns 0 if none of the characters are found.

The default for *start* is 1; thus, the search starts at the first character of *string*. You can override this by specifying a different *start* point, which must be a positive whole number.

If *string* is null, the function returns 0, regardless of the value of the third argument. Similarly, if *start* is greater than LENGTH(string), the function returns 0. If *reference* is null, the function returns 0 if you specify Match; otherwise the function returns the *start* value.

Here are some examples:

```
VERIFY('123','1234567890')              ->    0
VERIFY('1Z3','1234567890')              ->    2
VERIFY('AB4T','1234567890')             ->    1
VERIFY('AB4T','1234567890','M')         ->    3
VERIFY('AB4T','1234567890','N')         ->    1
VERIFY('1P3Q4','1234567890',,3)         ->    4
VERIFY('123','',N,2)                    ->    2
VERIFY('ABCDE','',,3)                   ->    3
VERIFY('AB3CD5','1234567890','M',4)     ->    6
```

14.3.59 WORD

```
▶▶──WORD(string,n)──────────────────────────────────────▶◀
```

returns the *n*th blank-delimited word in *string* or returns the null string if fewer than
n words are in *string*. The *n* must be a positive whole number. This function is
exactly equivalent to SUBWORD(*string*,*n*,1).

Here are some examples:

```
WORD('Now is the time',3)    ->    'the'
WORD('Now is the time',5)    ->    ''
```

14.3.60 WORDINDEX

```
▶▶──WORDINDEX(string,n)──────────────────────────────────▶◀
```

returns the position of the first character in the *n*th blank-delimited word in *string* or
returns 0 if fewer than *n* words are in *string*. The *n* must be a positive whole
number.

Here are some examples:

```
WORDINDEX('Now is the time',3)    ->    8
WORDINDEX('Now is the time',6)    ->    0
```

14.3.61 WORDLENGTH

```
▶▶──WORDLENGTH(string,n)─────────────────────────────────▶◀
```

returns the length of the *n*th blank-delimited word in *string* or returns 0 if fewer than
n words are in *string*. The *n* must be a positive whole number.

Here are some examples:

```
WORDLENGTH('Now is the time',2)      ->    2
WORDLENGTH('Now comes the time',2)   ->    5
WORDLENGTH('Now is the time',6)      ->    0
```

14.3.62 WORDPOS (Word Position)

```
▶▶──WORDPOS(phrase,string──────────)────────────────────▶◀
                        └─,start─┘
```

returns the word number of the first word of *phrase* found in *string* or returns 0 if
phrase contains no words or if *phrase* is not found. Multiple blanks between words
in either *phrase* or *string* are treated as a single blank for the comparison, but
otherwise the words must match exactly.

By default the search starts at the first word in *string*. You can override this by
specifying *start* (which must be positive), the word at which to start the search.

Here are some examples:

```
WORDPOS('the','now is the time')           ->  3
WORDPOS('The','now is the time')           ->  0
WORDPOS('is the','now is the time')        ->  2
WORDPOS('is   the','now is the time')      ->  2
WORDPOS('is   time ','now is   the time')  ->  0
WORDPOS('be','To be or not to be')         ->  2
WORDPOS('be','To be or not to be',3)       ->  6
```

14.3.63 WORDS

```
►►──WORDS(string)────────────────────────────────────►◄
```

returns the number of blank-delimited words in *string*.

Here are some examples:

```
WORDS('Now is the time')   ->    4
WORDS(' ')                 ->    0
```

14.3.64 XRANGE (Hexadecimal Range)

```
►►──XRANGE(──┬───────┬──┬──────┬──)──────────────────►◄
             └start─┘  └,end─┘
```

returns a string of all valid 1-byte encodings (in ascending order) between and
including the values *start* and *end*. The default value for *start* is `'00'`x, and the
default value for *end* is `'FF'`x. If *start* is greater than *end*, the values wrap from
`'FF'`x to `'00'`x. If specified, *start* and *end* must be single characters.

Here are some examples:

```
XRANGE('a','f')       ->    'abcdef'
XRANGE('03'x,'07'x)   ->    '0304050607'x
XRANGE(,'04'x)        ->    '0001020304'x
XRANGE('i','j')       ->    '898A8B8C8D8E8F9091'x  /* EBCDIC */
XRANGE('FE'x,'02'x)   ->    'FEFF000102'x
```

14.3.65 X2B (Hexadecimal to Binary)

```
►►──X2B(hexstring)───────────────────────────────────►◄
```

returns a string, in character format, that represents *hexstring* converted to binary.
The *hexstring* is a string of hexadecimal characters. It can be of any length. Each
hexadecimal character is converted to a string of four binary digits. You can
optionally include blanks in *hexstring* (at byte boundaries only, not leading or
trailing) to aid readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any
blanks.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2B('C3')          ->   '11000011'
X2B('7')           ->   '0111'
X2B('1 C1')        ->   '000111000001'
```

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form.

Here are some examples:

```
X2B(C2X('C3'x))    ->   '11000011'
X2B(D2X('129'))    ->   '10000001'
X2B(D2X('12'))     ->   '1100'
```

14.3.66 X2C (Hexadecimal to Character)

```
►►──X2C(hexstring)──────────────────────────────────►◄
```
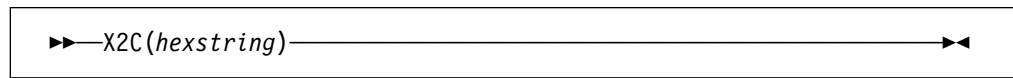
returns a string, in character format, that represents *hexstring* converted to character. The returned string is half as many bytes as the original *hexstring*. *hexstring* can be of any length. If necessary, it is padded with a leading 0 to make an even number of hexadecimal digits.

You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, the function returns a null string.

Here are some examples:

```
X2C('F7F2 A2')     ->   '72s'      /*  EBCDIC                     */
X2C('F7f2a2')      ->   '72s'      /*  EBCDIC                     */
X2C('F')           ->   ' '        /*  '0F' is unprintable EBCDIC */
```

14.3.67 X2D (Hexadecimal to Decimal)

```
►►──X2D(hexstring──┬────┬──)──────────────────────────►◄
                   └─,n─┘
```
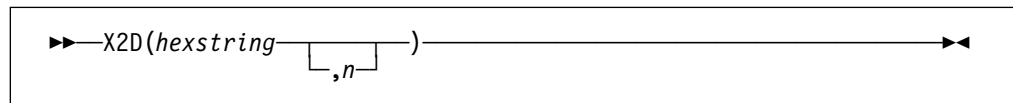
returns the decimal representation of *hexstring*. The *hexstring* is a string of hexadecimal characters. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, the function returns 0.

If you do not specify *n*, *hexstring* is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')        ->     14
X2D('81')        ->     129
X2D('F81')       ->     3969
X2D('FF81')      ->     65409
X2D('c6 f0'X)    ->     240          /*  EBCDIC  */
```

If you specify *n*, the string is taken as a signed number expressed in *n* hexadecimal digits. If the leftmost bit is off, then the number is positive; otherwise, it is a negative number in two's complement notation. In both cases it is converted to a whole number, which may, therefore, be negative. If *n* is 0, the function returns 0.

If necessary, *hexstring* is padded on the left with 0 characters (note, not "sign-extended"), or truncated on the left to *n* characters.

Here are some examples:

```
X2D('81',2)      ->     -127
X2D('81',4)      ->     129
X2D('F081',4)    ->     -3967
X2D('F081',3)    ->     129
X2D('F081',2)    ->     -127
X2D('F081',1)    ->     1
X2D('0031',0)    ->     0
```

Implementation maximum: The input string may not have more than 500 hexadecimal characters that will be significant in forming the final result. Leading sign characters (0 and F) do not count towards this total.

14.4 External Functions Provided in REXX/CICS

Additional external functions provided in the REXX/CICS environment are discussed in this section.

14.4.1 STORAGE

Note: This is an **authorized** function.

```
►►──STORAGE(address──┬──────────────────────┬──)──────►◄
                     └─,─┬─────────┬─┬───────┬
                         └─length─┘ └─,data─┘
```
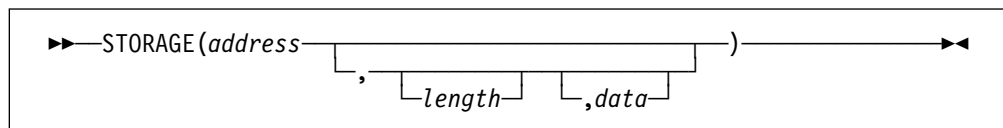
returns *length* bytes from the user's memory starting at *address*. The *length* is in decimal; the default is 1 byte. The *address* is a hexadecimal number. The high-order bit of *address* is ignored.

If you specify *data*, after the *old* value has been retrieved storage starting at *address* is overwritten with *data* (the *length* argument has no effect on this).

Note: The STORAGE function can operate on storage above the 16MB line.
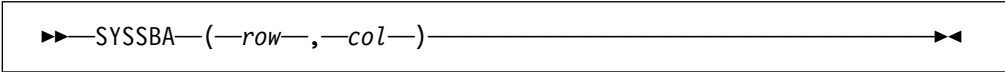
Warning: The STORAGE function, which allows a REXX user to display and/or modify the virtual storage of the CICS region, can only be successfully invoked from an authorized exec or by an authorized user.

Examples:

```
/* Note that the following results vary from system to system.  */
STORAGE(200000,32)
/* This returns 32 bytes of storage at hex address 200000 as a result. */
```

14.4.2 SYSSBA

```
►►──SYSSBA──(──row──,──col──)──────────────────────────────►◄
```

SYSSBA converts screen row,col to a set buffer address (SBA).

Operands:

row   specifies the row number counting from the top of the screen.

col   specifies the column number (counting from the left of the screen).

Example:

```
x = SYSSBA(10,20)
```

This example returns a three byte set buffer address for screen row 10, column 20 into REXX variable x.

Note: The SYSSBA function queries the terminal model on each invocation and uses this to adjust the SBA calculation to terminal type.

Chapter 15. Parsing

15.1 General Description

The parsing instructions are ARG, PARSE, and PULL (see section 13.2, "ARG" on page 158, section 13.14, "PARSE" on page 180, and section 13.16, "PULL" on page 186).

The data to parse is a *source string*. Parsing splits up the data in a source string and assigns pieces of it into the variables named in a template. A *template* is a model specifying how to split the source string. The simplest kind of template consists of only a list of variable names. Here is an example:

```
variable1 variable2 variable3
```

This kind of template parses the source string into blank-delimited words. More complicated templates contain patterns in addition to variable names.

String patterns   Match characters in the source string to specify where to split it. (See section 15.1.2, "Templates Containing String Patterns" on page 241 for details.)

Positional patterns Indicate the character positions at which to split the source string. (See section 15.1.3, "Templates Containing Positional (Numeric) Patterns" on page 242 for details.)

Parsing is essentially a two-step process.

1. Parse the source string into appropriate substrings using patterns.
2. Parse each substring into words.

15.1.1 Simple Templates for Parsing into Words

Here is a parsing instruction:

```
parse value 'time and tide' with var1 var2 var3
```

The template in this instruction is: `var1 var2 var3`. The data to parse is between the keywords `PARSE VALUE` and the keyword `WITH`, the source string `time and tide`. Parsing divides the source string into blank-delimited words and assigns them to the variables named in the template as follows:

```
var1='time'
var2='and'
var3='tide'
```

In this example, the source string to parse is a literal string, `time and tide`. In the next example, the source string is a variable.

```
/* PARSE VALUE using a variable as the source string to parse   */
string='time and tide'
parse value string with var1 var2 var3          /* same results */
```

(PARSE VALUE does not convert lowercase `a–z` in the source string to uppercase `A–Z`. If you want to convert characters to uppercase, use PARSE UPPER VALUE. See 15.1.5, "Using UPPER" on page 246 for a summary of the effect of parsing instructions on case.)

All of the parsing instructions assign the parts of a source string into the variables named in a template. There are various parsing instructions because of differences in the nature or origin of source strings. (A summary of all the parsing instructions is on page 247.)

The PARSE VAR instruction is similar to PARSE VALUE except that the source string to parse is always a variable. In PARSE VAR, the name of the variable containing the source string follows the keywords PARSE VAR. In the next example, the variable stars contains the source string. The template is star1 star2 star3.

```
/* PARSE VAR example                                      */
stars='Sirius Polaris Rigil'
parse var stars star1 star2 star3        /* star1='Sirius'  */
                                         /* star2='Polaris' */
                                         /* star3='Rigil'   */
```

All variables in a template receive new values. If there are *more variables in the template than words in the source string*, the leftover variables receive null (empty) values. This is true for all parsing: for parsing into words with simple templates and for parsing with templates containing patterns. Here is an example using parsing into words.

```
/* More variables in template than (words in) the source string  */
satellite='moon'
parse var satellite Earth Mercury        /* Earth='moon'  */
                                         /* Mercury=''     */
```

If there are *more words in the source string than variables in the template*, the last variable in the template receives all leftover data. Here is an example:

```
/* More (words in the) source string than variables in template  */
satellites='moon Io Europa Callisto...'
parse var satellites Earth Jupiter       /* Earth='moon'  */
                              /* Jupiter='Io Europa Callisto...'*/
```

Parsing into words removes leading and trailing blanks from each word before it is assigned to a variable. The exception to this is the word or group of words assigned to the last variable. The last variable in a template receives leftover data, *preserving extra leading and trailing blanks*. Here is an example:

```
/* Preserving extra blanks                                */
solar5='Mercury Venus  Earth   Mars     Jupiter '
parse var solar5 var1 var2 var3 var4
/* var1  ='Mercury'                                       */
/* var2  ='Venus'                                         */
/* var3  ='Earth'                                         */
/* var4 ='  Mars     Jupiter '                            */
```

In the source string, Earth has two leading blanks. Parsing removes both of them (the word-separator blank and the extra blank) before assigning var3='Earth'. Mars has three leading blanks. Parsing removes one word-separator blank and keeps the other two leading blanks. It also keeps all five blanks between Mars and Jupiter and both trailing blanks after Jupiter.

Parsing removes *no* blanks if the template contains only one variable. For example:

```
parse value '  Pluto   ' with var1       /* var1='  Pluto   '*/
```

15.1.1.1 The Period as a Placeholder

A period in a template is a placeholder. It is used instead of a variable name, but it receives no data. It is useful:

- As a "dummy variable" in a list of variables
- Or to collect unwanted information at the end of a string.

The period in the first example is a placeholder. Be sure to separate adjacent periods with spaces; otherwise, an error results.

```
/* Period as a placeholder                              */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars . . brightest .          /* brightest='Sirius' */


/* Alternative to period as placeholder                 */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars drop junk brightest rest   /* brightest='Sirius' */
```

A placeholder saves the overhead of unneeded variables.

15.1.2 Templates Containing String Patterns

A *string pattern* matches characters in the source string to indicate where to split it. A string pattern can be a:

Literal string pattern One or more characters within quotation marks.

Variable string pattern A variable within parentheses with no plus (+) or minus (-) or equal sign (=) before the left parenthesis. (See page 246 for details.)

Here are two templates: a simple template and a template containing a literal string pattern:

```
var1 var2          /* simple template                         */
var1 ', ' var2     /* template with literal string pattern     */
```

The literal string pattern is: ', '. This template:

- Puts characters from the start of the source string up to (but not including) the first character of the match (the comma) into `var1`

- Puts characters starting with the character after the last character of the match (the character after the blank that follows the comma) and ending with the end of the string into `var2`.

A template with a string pattern can omit some of the data in a source string when assigning data into variables. The next two examples contrast simple templates with templates containing literal string patterns.

```
/* Simple template                                      */
name='Smith, John'
parse var name ln fn                /* Assigns: ln='Smith,' */
                                    /*          fn='John'   */
```

Notice that the comma remains (the variable `ln` contains `'Smith,'`). In the next example the template is `ln ', ' fn`. This removes the comma.

```
/* Template with literal string pattern                 */
name='Smith, John'
parse var name ln ', ' fn           /* Assigns: ln='Smith' */
                                    /*          fn='John'  */
```

First, the language processor scans the source string for ', '. It splits the source string at that point. The variable `ln` receives data starting with the first character of the source string and ending with the last character before the match. The variable `fn` receives data starting with the first character *after* the match and ending with the end of string.

A template with a string pattern omits data in the source string that matches the pattern. (There is a special case (on page 249) in which a template with a string pattern does *not* omit matching data in the source string.) We used the pattern '**,** ' (with a blank) instead of '**,**' (no blank) because, without the blank in the pattern, the variable `fn` receives ' John' (including a blank).

If the source string *does not contain a match for a string pattern*, then any variables preceding the unmatched string pattern get all the data in question. Any variables after that pattern receive the null string.

A null string is never found. It always matches the end of the source string.

15.1.3 Templates Containing Positional (Numeric) Patterns

A *positional pattern* is a number that identifies the character position at which to split data in the source string. The number must be a whole number.

An *absolute positional pattern* is

- A number with no plus (+) or minus (-) sign preceding it or with an equal sign (=) preceding it
- A variable in parentheses with an equal sign before the left parenthesis. (See page 246 for details on *variable positional patterns*.)

The number specifies the absolute character position at which to split the source string.

Here is a template with absolute positional patterns:

```
variable1 11 variable2 21 variable3
```

The numbers `11` and `21` are absolute positional patterns. The number `11` refers to the 11th position in the input string, `21` to the 21st position. This template:

- Puts characters 1 through 10 of the source string into `variable1`
- Puts characters 11 through 20 into `variable2`
- Puts characters 21 to the end into `variable3`.

Positional patterns are probably most useful for working with a file of records, such as:

```
        character positions:
        1        11         21                40
                                                ⌐end of
 FIELDS: │LASTNAME │FIRST    │PSEUDONYM        ││record
         └─────────┴─────────┴─────────────────┘
```

The following example uses this record structure.

```
/* Parsing with absolute positional patterns in template      */
record.1='Clemens    Samuel    Mark Twain          '
record.2='Evans      Mary Ann  George Eliot        '
record.3='Munro      H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                       /* Says 'By George!' after record 2  */
```

The source string is first split at character position 11 and at position 21. The language processor assigns characters 1 to 10 into `lastname`, characters 11 to 20 into `firstname`, and characters 21 to 40 into `pseudonym`.

The template could have been:

```
1 lastname 11 firstname 21 pseudonym
```

instead of

```
  lastname 11 firstname 21 pseudonym
```

Specifying the 1 is optional.

Optionally, you can put an equal sign before a number in a template. An equal sign is the same as no sign before a number in a template. The number refers to a particular character position in the source string. These two templates work the same:

```
lastname  11 first  21 pseudonym
```

```
lastname =11 first =21 pseudonym
```

A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. (It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see section 15.1.4, "Parsing with Variable Patterns" on page 246.)

The number specifies the relative character position at which to split the source string. The plus or minus indicates movement right or left, respectively, from the start of the string (for the first pattern) or from the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for absolute positional patterns done with relative positional patterns:

```
/* Parsing with relative positional patterns in template       */
record.1='Clemens    Samuel    Mark Twain          '
record.2='Evans      Mary Ann  George Eliot        '
record.3='Munro      H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                                         /* same results */
```

Blanks between the sign and the number are insignificant. Therefore, +10 and + 10 have the same meaning. Note that +0 is a valid relative positional pattern.

Absolute and relative positional patterns are interchangeable (except in the special case (on page 249) when a string pattern precedes a variable name and a positional pattern follows the variable name). The templates from the examples of absolute and relative positional patterns give the same results.

```
|       |  |lastname   11|  |firstname 21|  |pseudonym |
|       |  |lastname  +10|  |firstname + 10|  |pseudonym |
|       |
  |          |               |               |
(Implied    Put characters   Put characters   Put characters
starting    1 through 10     11 through 20    21 through
point is    in lastname.     in firstname.    end of string
position    (Non-inclusive   (Non-inclusive   in pseudonym.
1.)         stopping point   stopping point
            is 11 (1+10).)   is 21 (11+10).)
```
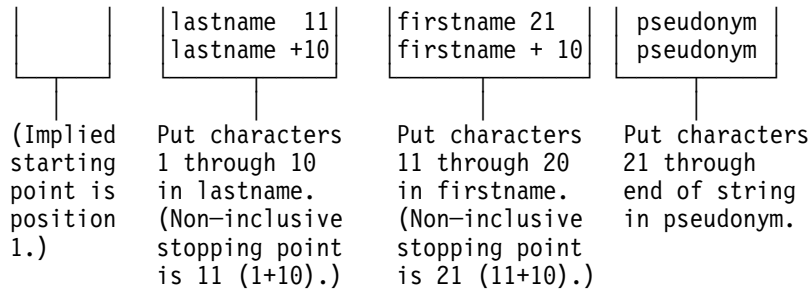
Only with positional patterns can a matching operation *back up* to an earlier position in the source string. Here is an example using absolute positional patterns:

```
/* Backing up to an earlier position (with absolute positional)  */
string='astronomers'
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string 'study' var1||var2||var3||var4
/* Displays: "astronomers study stars"                           */
```

The absolute positional pattern 1 backs up to the first character in the source string.

With relative positional patterns, a number preceded by a minus sign backs up to an earlier position. Here is the same example using relative positional patterns:

```
/* Backing up to an earlier position (with relative positional)  */
string='astronomers'
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string 'study' var1||var2||var3||var4      /* same results   */
```

In the previous example, the relative positional pattern -3 backs up to the first character in the source string.

The templates in the last two examples are equivalent.

```
|  2  |   |var1  4 |   |  1  |   |var2  2 |   |4 var3  5|   |11 var4 |
|  2  |   |var1 +2 |   | -3  |   |var2 +1 |   |+2 var3 +1|   |+6 var4 |
   |          |            |          |            |             |
Start      Non-        Go to 1.  Non-         Go to 4      Go to 11
at 2.      inclusive   (4-3=1)   inclusive    (2+2=4).     (5+6=11).
           stopping              stopping     Non-inclusive
           point is 4            point is     stopping point
           (2+2=4).              2 (1+1=2).   is 5 (4+1=5).
```
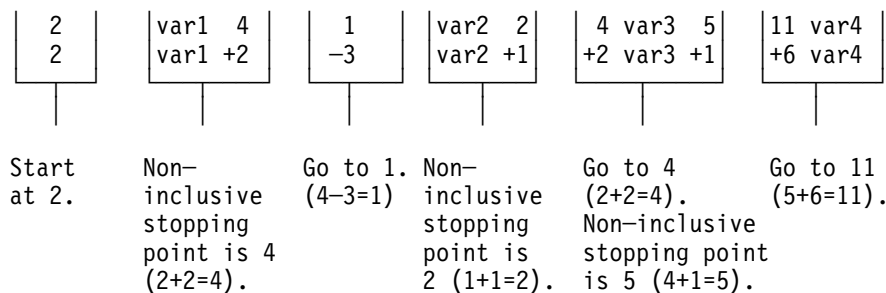
You can use templates with positional patterns to make multiple assignments:

```
/* Making multiple assignments                                    */
books='Silas Marner, Felix Holt, Daniel Deronda, Middlemarch'
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans.    */
```

15.1.3.1 Combining Patterns and Parsing Into Words

What happens when a template contains patterns that divide the source string into sections containing multiple words? String and positional patterns divide the source string into substrings. The language processor then applies a section of the template to each substring, following the rules for parsing into words.

```
/* Combining string pattern and parsing into words        */
name='    John      Q.    Public'
parse var name fn init '.' ln        /* Assigns: fn='John'      */
                                     /*          init='    Q'   */
                                     /*          ln='   Public' */
```

The pattern divides the template into two sections:

- fn init
- ln

The matching pattern splits the source string into two substrings:

- '   John   Q'
- '   Public'

The language processor parses these substrings into words based on the appropriate template section.

John had three leading blanks. All are removed because parsing into words removes leading and trailing blanks except from the last variable.

Q has six leading blanks. Parsing removes one word-separator blank and keeps the rest because init is the last variable in that section of the template.

For the substring '   Public', parsing assigns the entire string into ln without removing any blanks. This is because ln is the only variable in this section of the template. (For details about treatment of blanks, see page 240.)

```
/* Combining positional patterns with parsing into words       */
string='R E X X'
parse var string var1 var2 4 var3 6 var4   /* Assigns: var1='R'  */
                                           /*          var2='E'  */
                                           /*          var3=' X' */
                                           /*          var4=' X' */
```

The pattern divides the template into three sections:

- var1 var2
- var3
- var4

The matching patterns split the source string into three substrings that are individually parsed into words:

- 'R E'
- ' X'
- ' X'

The variable var1 receives 'R'; var2 receives 'E'. Both var3 and var4 receive ' X' (with a blank before the X) because each is the only variable in its section of the template. (For details on treatment of blanks, see page 240.)

15.1.4 Parsing with Variable Patterns

You may want to specify a pattern by using the value of a variable instead of a fixed string or number. You do this by placing the name of the variable in parentheses. This is a *variable reference*. Blanks are not necessary inside or outside the parentheses, but you can add them if you wish.

The template in the next parsing instruction contains the following literal string pattern '. '.

```
parse var name fn  init '. ' ln
```

Here is how to specify that pattern as a variable string pattern:

```
strngptrn='. '
parse var name fn init (strngptrn) ln
```

If no equal, plus, or minus sign precedes the parenthesis that is before the variable name, the value of the variable is then treated as a string pattern. The variable can be one that has been set earlier in the same template.

Example:

```
/* Using a variable as a string pattern                      */
/*  The variable (delim) is set in the same template         */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/90 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
            /* Sets: month='11'; delim='/'; day='15'; year='90'  */
```

If an equal, a plus, or a minus sign precedes the left parenthesis, then the value of the variable is treated as an absolute or relative positional pattern. The value of the variable must be a positive whole number or zero.

The variable can be one that has been set earlier in the same template. In the following example, the first two fields specify the starting character positions of the last two fields.

Example:

```
/* Using a variable as a positional pattern                   */
dataline = '12 26 .....Samuel ClemensMark Twain'
parse var dataline pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname='Samuel Clemens'; pseudonym='Mark Twain'    */
```

Why is the positional pattern 6 needed in the template? Remember that word parsing occurs *after* the language processor divides the source string into substrings using patterns. Therefore, the positional pattern =(pos1) cannot be correctly interpreted as =12 until *after* the language processor has split the string at column 6 and assigned the blank-delimited words 12 and 26 to pos1 and pos2, respectively.

15.1.5 Using UPPER

Specifying UPPER on any of the PARSE instructions converts characters to uppercase (lowercase a–z to uppercase A–Z) before parsing. The following table summarizes the effect of the parsing instructions on case.

Converts alphabetic characters to uppercase before parsing	Maintains alphabetic characters in case entered
ARG PARSE UPPER ARG	PARSE ARG
PARSE UPPER EXTERNAL	PARSE EXTERNAL
PARSE UPPER NUMERIC	PARSE NUMERIC
PULL PARSE UPPER PULL	PARSE PULL
PARSE UPPER SOURCE	PARSE SOURCE
PARSE UPPER VALUE	PARSE VALUE
PARSE UPPER VAR	PARSE VAR
PARSE UPPER VERSION	PARSE VERSION

The ARG instruction is simply a short form of PARSE UPPER ARG. The PULL instruction is simply a short form of PARSE UPPER PULL. If you do not desire uppercase translation, use PARSE ARG (instead of ARG or PARSE UPPER ARG) and use PARSE PULL (instead of PULL or PARSE UPPER PULL).

15.1.6 Parsing Instructions Summary

Remember: *All* parsing instructions assign parts of the source string into the variables named in the template. The following table summarizes where the source string comes from.

Instruction	Where the source string comes from
ARG PARSE ARG	Arguments you list when you call the program or arguments in the call to a subroutine or function.
PARSE EXTERNAL	Next line from terminal input buffer
PARSE NUMERIC	Numeric control information (from NUMERIC instruction).
PULL PARSE PULL	The string at the head of the external data queue. (If queue empty, uses default input, typically the terminal.)
PARSE SOURCE	System-supplied string giving information about the executing program.
PARSE VALUE	Expression between the keyword VALUE and the keyword WITH in the instruction.
PARSE VAR *name*	Parses the value of *name*.
PARSE VERSION	System-supplied string specifying the language, language level, and (three-word) date.

15.1.7 Parsing Instructions Examples

All examples in this section parse source strings into words.

ARG

```
/* ARG with source string named in REXX program invocation    */
/*  Program name is PALETTE.  Specify 2 primary colors (yellow, */
/*   red, blue) on call.   Assume call is: palette red blue    */
arg var1 var2              /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1                     /* entered duplicates */
END
Say new; exit                           /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in section 15.2.1, "Parsing Multiple Strings" on page 249.

PARSE ARG works the same as ARG except that PARSE ARG does not convert alphabetic characters to uppercase before parsing.

PARSE EXTERNAL

```
Say "Enter Yes or No =====> "
parse upper external answer 2 .
If answer='Y'
  then say "You said 'Yes'!"
  else say "You said 'No'!"
```

PARSE NUMERIC

```
parse numeric digits fuzz form
say digits fuzz form       /* Displays: '9 0 SCIENTIFIC'    */
                           /* (if defaults are in effect)   */
```

PARSE PULL

```
PUSH '80 7'               /* Puts data on queue            */
parse pull fourscore seven /* Assigns: fourscore='80'; seven='7' */
SAY fourscore+seven       /* Displays: "87"                */
```

PARSE SOURCE

```
parse source sysname .
Say sysname                            /* Displays:    "CICS" */
```

PARSE VALUE example is on page 239.

PARSE VAR examples are throughout the chapter, starting on page 240.

PARSE VERSION

```
parse version . level .
say level                              /* Displays: "3.48" */
```

PULL works the same as PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.

15.2 Advanced Topics in Parsing

This section includes parsing multiple strings and flow charts depicting a conceptual view of parsing.

15.2.1 Parsing Multiple Strings

Only ARG and PARSE ARG can have more than one source string. To parse *multiple strings*, you can specify multiple comma-separated templates. Here is an example:

```
parse arg template1, template2, template3
```

This instruction consists of the keywords PARSE ARG and three comma-separated templates. (For an ARG instruction, the source strings to parse come from arguments you specify when you call a program or CALL a subroutine or function.) Each comma is an instruction to the parser to move on to the next string.

Example:

```
/* Parsing multiple strings in a subroutine                    */
num='3'
musketeers="Porthos Athos Aramis D'Artagnon"
CALL Sub num,musketeers   /* Passes num and musketeers to sub    */
SAY total; say fourth /* Displays: "4" and " D'Artagnon"        */
EXIT

Sub:
parse arg subtotal, . . . fourth
total=subtotal+1
RETURN
```

Note that when a REXX program is started as a command, only one argument string is recognized. You can pass multiple argument strings for parsing:

- When one REXX program calls another REXX program with the CALL instruction or a function call.

- When programs written in other languages start a REXX program.

If there are more templates than source strings, each variable in a leftover template receives a null string. If there are more source strings than templates, the language processor ignores leftover source strings. If a template is empty (two commas in a row) or contains no variable names, parsing proceeds to the next template and source string.

15.2.2 Combining String and Positional Patterns: A Special Case

There is a special case in which absolute and relative positional patterns do not work identically. We have shown how parsing with a template containing a string pattern skips over the data in the source string that matches the pattern (see page 242). But a template containing the sequence:

- string pattern
- variable name
- *relative* positional pattern

does *not* skip over the matching data. A relative positional pattern moves relative to the first character matching a string pattern. As a result, assignment includes the data in the source string that matches the string pattern.

```
/* Template containing string pattern, then variable name, then  */
/*  relative positional pattern does not skip over any data.     */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:

```
|var1  3|    |junk 'X'|    |var2 +1|    |junk  'X'|    |var3 +1|    |junk |
     |            |            |            |             |            |

Put          Starting     Starting     Starting      Starting     Starting
characters   at 3, put    with first   with char-    with         with char-
1 through    characters   'X' put 1    acter after   second 'X'   acter
2 in var1.   up to (not   (+1)         first 'X'     put 1 (+1)   after sec-
(Stopping    including)   character    put up to     character    ond 'X'
point is     first 'X'    in var2.     second 'X'    in var3.     put rest
3.)          in junk.                  in junk.                   in junk.

var1='RE'    junk=        var2='X'     junk=         var3='X'     junk=
             'structured               'tended e'                 'ecutor'
             e'
```

15.2.3 Parsing with DBCS Characters

Parsing with DBCS characters generally follows the same rules as parsing with SBCS characters. Literal strings and symbols can contain DBCS characters, but numbers must be in SBCS characters. See C.1.4.1, "PARSE" on page 452 for examples of DBCS parsing.

15.2.4 Details of Steps in Parsing

The three figures that follow are to help you understand the concept of parsing. Please note that the figures do not include error cases.

The figures include terms whose definitions are as follows:

string start   is the beginning of the source string (or substring).

string end   is the end of the source string (or substring).

length   is the length of the source string.

match start   is in the source string and is the first character of the match.

match end   is in the source string. For a string pattern, it is the first character after the end of the match. For a positional pattern, it is the same as match start.

match position is in the source string. For a string pattern, it is the first matching character. For a positional pattern, it is the position of the matching character.

token   is a distinct syntactic element in a template, such as a variable, a period, a pattern, or a comma.

value   is the numeric value of a positional pattern. This can be either a constant or the resolved value of a variable.

```
┌─────────────────────────────────────────────────────────────────┐
│                          ┌──────────────────────────────────┐    │
│                          ▼                                   │    │
│          ┌──────────────────────────────────────┐           │    │
│          │ START                                 │           │    │
│          │ Token is first one in template.       │           │    │
│          │ Length=length(source string)          │           │    │
│          │ Match start=1. Match end=1.           │           │    │
│          └──────────────────────────────────────┘           │    │
│                          │                                   │    │
│          ┌───────────────▼                                   │    │
│          │               ▼             yes                   │    │
│          │  ┌────────────────────┐    ┌──────────────────┐   │    │
│          │  │ End of template?   │───▶│ Parsing complete.│   │    │
│          │  └────────────────────┘    └──────────────────┘   │    │
│          │               │ no                                │    │
│          │  ┌────────────▼───────┐                           │    │
│          │  │ CALL Find Next     │                           │    │
│          │  │  Pattern.          │                           │    │
│          │  └────────────────────┘                           │    │
│          │               │                                   │    │
│          │  ┌────────────▼───────┐                           │    │
│          │  │ CALL Word Parsing. │                           │    │
│          │  └────────────────────┘                           │    │
│          │               │                                   │    │
│          │  ┌────────────▼───────┐                           │    │
│          │  │ Step to next token.│                           │    │
│          │  └────────────────────┘                           │    │
│          │               │             yes                   │    │
│          │  ┌────────────▼───────┐    ┌──────────────────┐   │    │
│          │  │ Token a comma?     │───▶│ Set next source  │───┘    │
│          │  └────────────────────┘    │ string and       │        │
│          └───────────│ no             │ template.        │        │
│                                       └──────────────────┘        │
└─────────────────────────────────────────────────────────────────┘
```
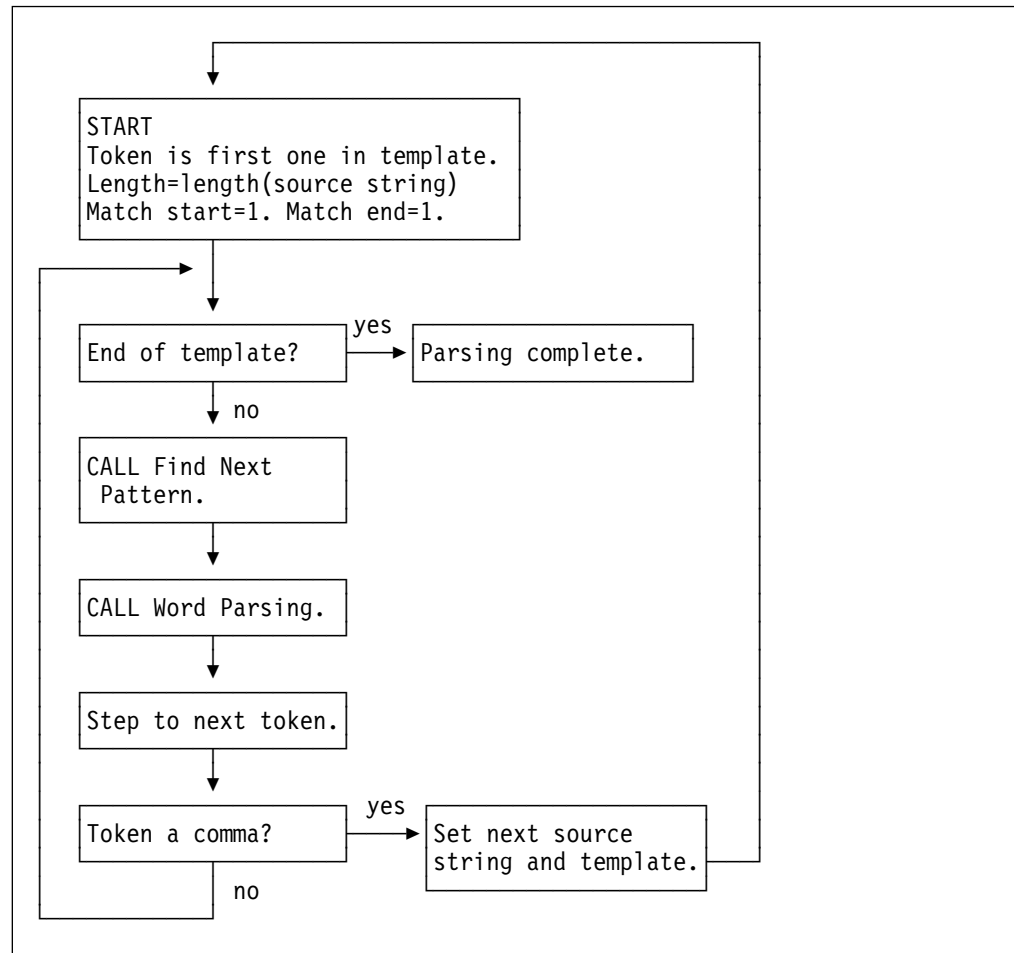
Figure 50. Conceptual Overview of Parsing
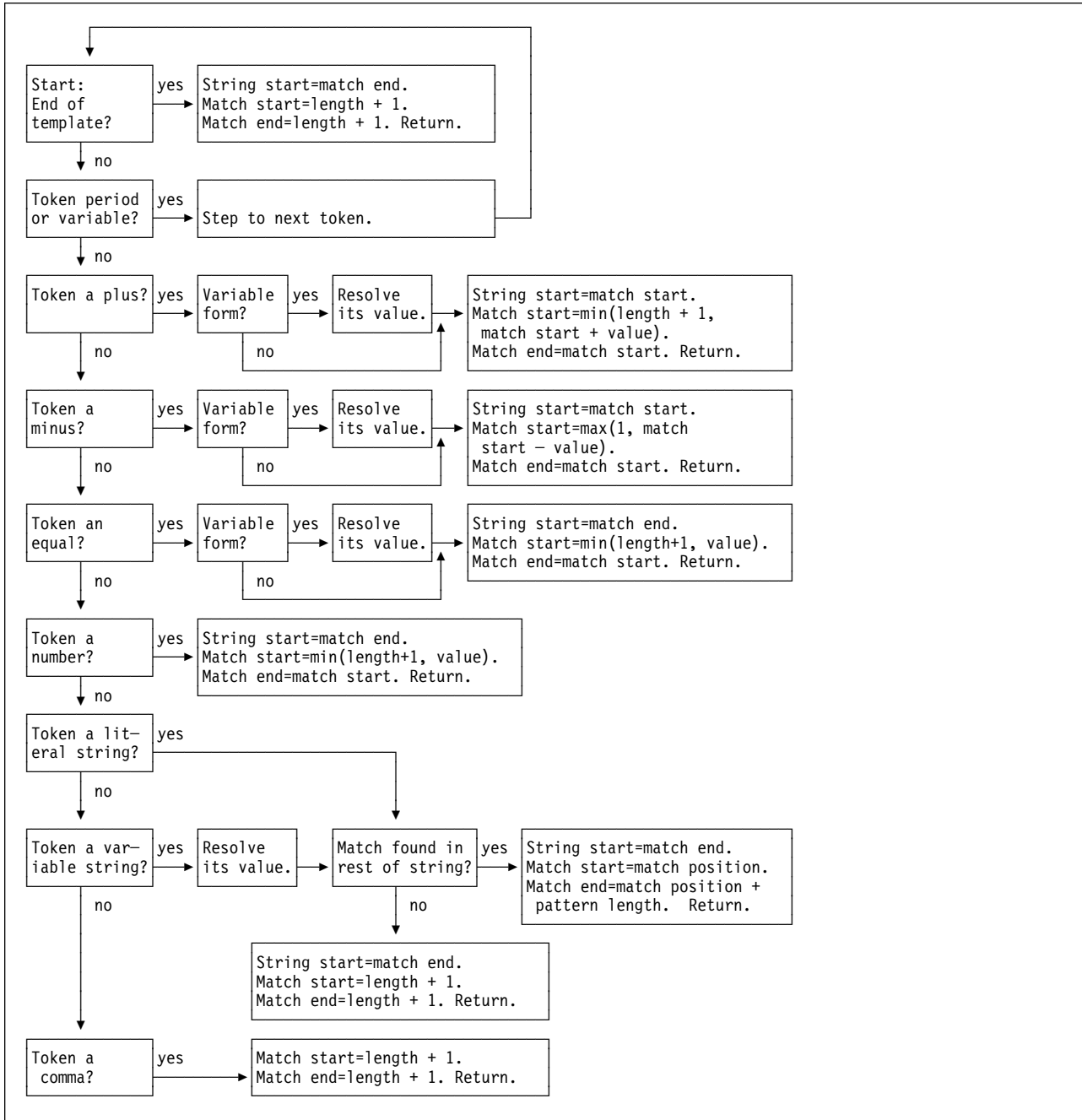
Figure 51. Conceptual View of Finding Next Pattern
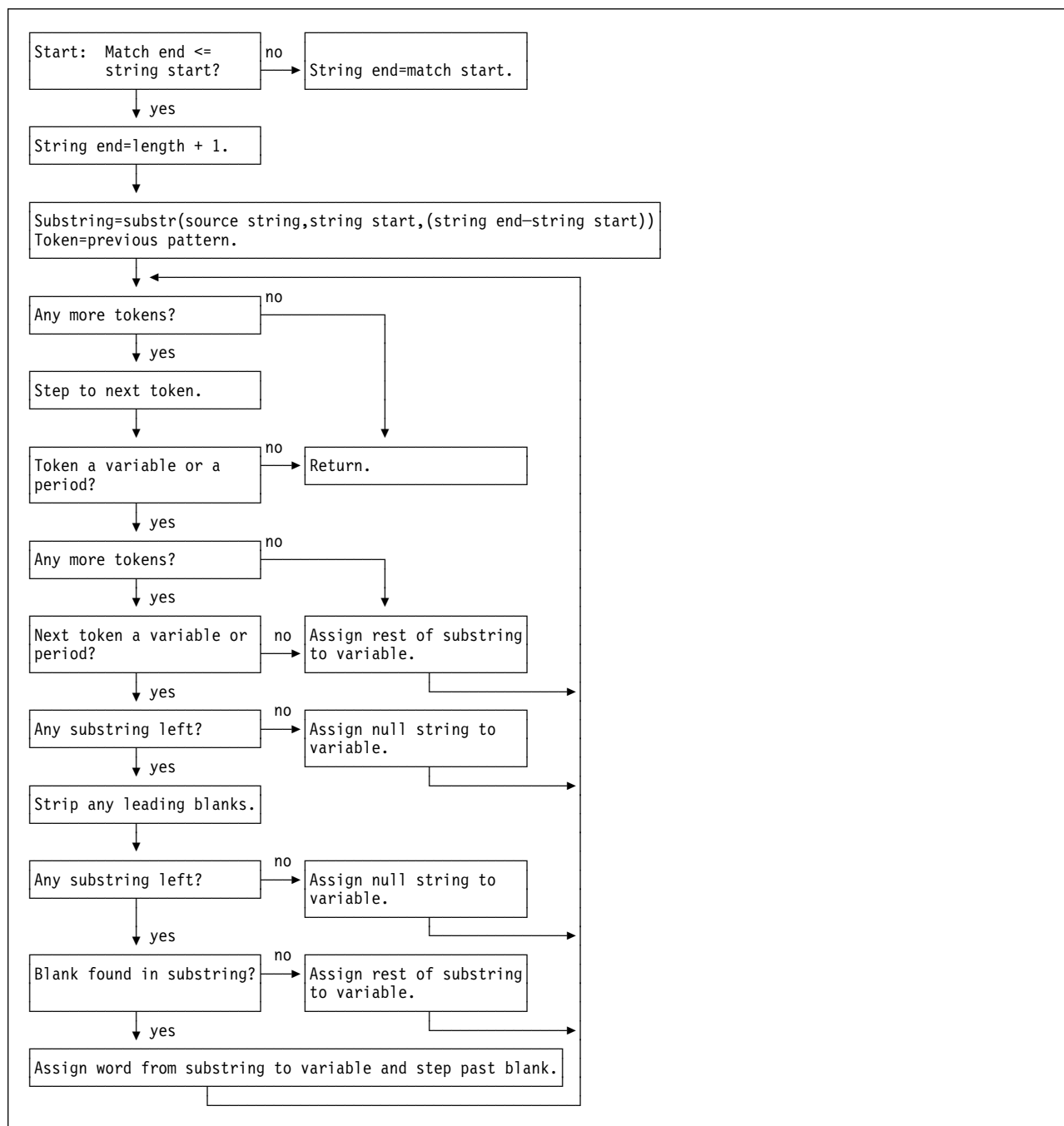
Figure 52. Conceptual View of Word Parsing

Parsing

Chapter 16. Numbers and Arithmetic

REXX defines the usual arithmetic operations (addition, subtraction, multiplication, and division) in as natural a way as possible. What this really means is that the rules followed are those that are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The arithmetic described here is, therefore, a compromise that (although not the simplest) should provide acceptable results in most applications.

16.1 Introduction

Numbers (that is, character strings used as input to REXX arithmetic operations and built-in functions) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

```
   12         /* a whole number                    */
  '-76'       /* a signed whole number             */
   12.76      /* decimal places                    */
' +  0.003 '  /* blanks around the sign and so forth */
   17.        /* same as "17"                      */
    .5        /* same as "0.5"                     */
   4E9        /* exponential notation              */
   0.73e-7    /* exponential notation              */
```

In exponential notation, a number includes an exponent, a power of ten by which the number is multiplied before use. The exponent indicates how the decimal point is shifted. Thus, in the preceding examples, `4E9` is simply a short way of writing `4000000000`, and `0.73e-7` is short for `0.000000073`.

The **arithmetic operators** include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, there are two further division operators: integer divide (%) divides and returns the integer part; remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see the "Definition" section for full details):

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus, if a result requires more than 9 digits, it would usually be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).

- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros in the decimal part of results). So, for example:

```
2.40 + 2    ->    4.40
2.40 - 2    ->    0.40
2.40 * 2    ->    4.80
2.40 / 2    ->    1.2
```

This behavior is desirable for most calculations (especially financial calculations).

If necessary, you can remove trailing zeros with the STRIP function (see page 227), or by division by 1.

- A zero result is always expressed as the single digit 0.

- Exponential form is used for a result depending on its value and the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number is expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12           /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11  /* not 0.0000000000333333333 */
```

16.2 Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

16.2.1 Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. (See section 16.2.7, "Exponential Notation" on page 261 for an extension of this definition.) The decimal point may be embedded in the number, or may be a prefix or suffix. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:
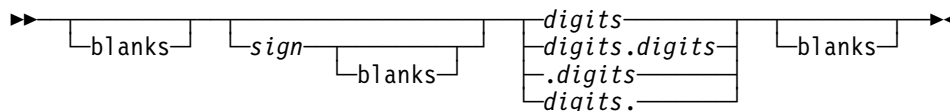
blanks
     are one or more spaces

sign
     is either + or –

digits
     are one or more of the decimal digits 0–9.

Note that a single period alone is not a valid number.

16.2.2 Precision

Precision is the maximum number of significant digits that can result from an operation. This is controlled by the instruction:

The *expression* is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify *expression* in this instruction, or if no NUMERIC DIGITS instruction has been processed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. However, use small values with care—the loss of precision and rounding thus requested affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

16.2.3 Arithmetic Operators

REXX arithmetic is performed by the operators +, -, *, /, %, //, and ** (add, subtract, multiply, divide, integer divide, remainder, and power), which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving only one zero if all the digits in the number are zeros). They are then truncated (if necessary) to DIGITS + 1 significant digits before being used in the computation. (The extra digit is a "guard" digit. It improves accuracy because it is inspected at the end of an operation, when a number is rounded to the required precision.) The operation is then carried out under up to double that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Rounding is done in the traditional manner. The digit to the right of the least significant digit in the result (the "guard digit") is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is, therefore, not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, insignificant trailing zeros are removed after rounding.

The FORMAT built-in function (see page 219) allows a number to be represented in a particular format if the standard result provided does not meet your requirements.

16.2.4 Arithmetic Operation Rules—Basic Operators

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows.

16.2.4.1 Addition and Subtraction

If either number is `0`, the other number, rounded to NUMERIC DIGITS digits, if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary, up to a total maximum of DIGITS + 1 digits (the number with the smaller absolute value may, therefore, lose some or all of its digits on the right) and are then added or subtracted as appropriate.

Example:

```
        xxx.xxx  +  yy.yyyyy
```

becomes:

```
        xxx.xxx00
    +  0yy.yyyyy
       _____
        zzz.zzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS if necessary (taking into account any extra "carry digit" on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted). Finally, any insignificant leading zeros are removed.

The prefix operators are evaluated using the same rules; the operations `+number` and `-number` are calculated as `0+number` and `0-number`, respectively.

16.2.4.2 Multiplication

The numbers are multiplied together ("long multiplication") resulting in a number that may be as long as the sum of the lengths of the two operands.

Example:

```
        xxx.xxx * yy.yyyyy
```

becomes:

```
        zzzzz.zzzzzzzz
```

The result is then rounded, counting from the first significant digit of the result, to the current setting of NUMERIC DIGITS.

16.2.4.3 Division

For the division:

```
yyy / xxxxx
```

the following steps are taken: First the number `yyy` is extended with zeros on the right until it is larger than the number `xxxxx` (with note being taken of the change in the power of ten that this implies). Thus, in this example, `yyy` might become `yyy00`. Traditional long division then takes place. This might be written:

```
                zzzz
              _____
    xxxxx  |   yyy00
```

The length of the result (`zzzz`) is such that the rightmost `z` is at least as far right as the rightmost digit of the (extended) `y` number in the example. During the division, the `y` number is extended further as necessary. The `z` number may increase up to NUMERIC DIGITS+1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

16.2.4.4 Basic Operator Examples

Following are some examples that illustrate the main implications of the rules just described.

```
/* With:  Numeric digits 5 */
12+7.00      ->    19.00
1.3-1.07     ->     0.23
1.3-2.07     ->    -0.77
1.20*3       ->     3.60
7*3          ->    21
0.9*0.8      ->     0.72
1/3          ->     0.33333
2/3          ->     0.66667
5/2          ->     2.5
1/10         ->     0.1
12/12        ->     1
8.0/2        ->     4
```

Note: With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterward. Therefore, the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

16.2.5 Arithmetic Operation Rules—Additional Operators

The operation rules for the power (**), integer divide (%), and remainder (//) operators follow.

16.2.5.1 Power

The **** (power) operator** raises a number to a power, which may be positive, negative, or 0. The power must be a whole number. (The second term in the operation must be a whole number and is rounded to DIGITS digits, if necessary, as described under 16.2.10, "Numbers Used Directly by REXX" on page 263.) If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the power, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by 1).

In practice (see Note 1 on page 260 for the reasons), the power is calculated by the process of left-to-right binary reduction. For a**n: n is converted to binary, and a temporary accumulator is set to 1. If n = 0 the initial calculation is complete. (Thus, a**0 = 1 for all a, including 0**0.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by a. If all bits have now been inspected, the initial calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the arithmetic operation rules, using a precision of DIGITS + L + 1 digits. L is the length in digits of the integer part of the whole number n (that is, excluding any decimal part, as though the built-in function TRUNC(n) had been used). Finally, the result is rounded to NUMERIC DIGITS digits, if necessary, and insignificant trailing zeros are removed.

16.2.5.2 Integer Division

The **% (integer divide) operator** divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result from regular division.

The result returned has no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed as a whole number, the operation is in error and will fail—that is, the result must not have more digits than the current setting of NUMERIC DIGITS. For example, `10000000000%3` requires 10 digits for the result (3333333333) and would, therefore, fail if `NUMERIC DIGITS 9` were in effect. Note that this operator may not give the same result as truncating regular division (which could be affected by rounding).

16.2.5.3 Remainder

The **// (remainder) operator** returns the remainder from integer division and is defined as being the residue of the dividend after the operation of calculating integer division as previously described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation fails under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

16.2.5.4 Additional Operator Examples

Following are some examples using the power, integer divide, and remainder operators:

```
/* Again with:  Numeric digits 5 */
2**3        ->      8
2**-3       ->      0.125
1.7**8      ->     69.758
2%3         ->      0
2.1//3      ->      2.1
10%3        ->      3
10//3       ->      1
-10//3      ->     -1
10.2//1     ->      0.2
10//0.3     ->      0.1
3.6//1.3    ->      1.0
```

Notes:

1. A particular algorithm for calculating powers is used, because it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It, therefore, gives better performance than the simpler definition of repeated multiplication. Because results may differ from those of repeated multiplication, the algorithm is defined here.

2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

16.2.6 Numeric Comparisons

The comparison operators are listed in section 12.2.2.3, "Comparison" on page 138. You can use any of these for comparing numeric strings. However, you should not use ==, \==, ¬==, >>, \>>, ¬>>, <<, \<<, and ¬<< for comparing numbers because leading and trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

```
A ? Z
```

where ? is any numeric comparison operator, is identical with:

```
(A - Z) ? '0'
```

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

A quantity called **fuzz** affects the comparison of two numbers. This controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The FUZZ value is set by the instruction:

```
►►──NUMERIC FUZZ─────────────────;──►◄
              └─expression─┘
```

Here *expression* must result in a positive whole number or zero. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison. That is, the numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison. Clearly the FUZZ setting must be less than DIGITS.

Thus if DIGITS = 9 and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though `NUMERIC DIGITS 8` had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5     /* Displays "0"   */
say  4.9999 < 5     /* Displays "1"   */
Numeric fuzz 1
say  4.9999 = 5     /* Displays "1"   */
say  4.9999 < 5     /* Displays "0"   */
```

16.2.7 Exponential Notation

The preceding description of numbers describes "pure" numbers, in the sense that the character strings that describe numbers can be very long. For example:

```
10000000000 * 10000000000
```

would give

```
100000000000000000000
```

and

```
.00000000001 * .00000000001
```

would give

```
0.000000000000000000000001
```

For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the "simple" form would give misleading information.

For example:

```
numeric digits 5
say 54321*54321
```

would display `2950800000` in long form. This is clearly misleading, and so the result is expressed as `2.9508E+9` instead.

The definition of numbers is, therefore, extended as:
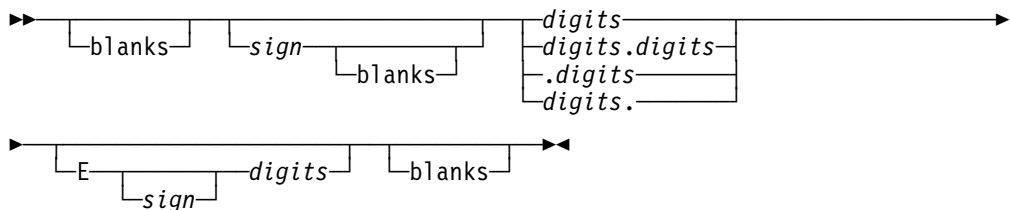
The integer following the `E` represents a power of ten that is to be applied to the number. The `E` can be in uppercase or lowercase.

Certain character strings are numbers even though they do not appear to be numeric to the user. Specifically, because of the format of numbers in exponential notation, strings, such as `0E123` (0 raised to the 123 power) and `1E342` (1 raised to the 342 power), are numeric. In addition, a comparison such as `0E123=0E567` gives a true result of `1` (0 is equal to 0). To prevent problems when comparing nonnumeric strings, use the strict comparison operators.

Here are some examples:

```
12E7   =    120000000       /* Displays "1" */
12E-5  =    0.00012         /* Displays "1" */
-12e4  =    -120000         /* Displays "1" */
0e123  =    0e456           /* Displays "1" */
0e123  ==   0e456           /* Displays "0" */
```

The preceding numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form, depending on the setting of NUMERIC DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form REXX generates always has a sign following the `E` to improve readability. If the exponent is `0`, then the exponential part is omitted—that is, an exponential part of `E+0` is never generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in long form, by using the FORMAT built-in function (see page 219).

Scientific notation is a form of exponential notation that adjusts the power of ten so a single nonzero digit appears to the left of the decimal point. **Engineering notation** is a form of exponential notation in which from one to three digits (but not

simply 0) appear before the decimal point, and the power of ten is always expressed as a multiple of three. The integer part may, therefore, range from 1 through 999. You can control whether Scientific or Engineering notation is used with the instruction:

```
>>──NUMERIC FORM──┬──SCIENTIFIC──┬──────────;──><
                  ├──ENGINEERING──┤
                  │        expression
                  └──VALUE──┘
```
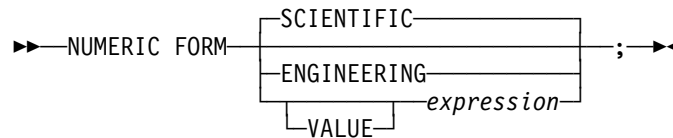
Scientific notation is the default.

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11     ->     1.2345E+13

/* after the instruction */
Numeric form engineering

123.45  * 1e11    ->     12.345E+12
```

16.2.8 Numeric Information

To determine the current settings of the NUMERIC options, use the built-in functions DIGITS, FORM, and FUZZ. These functions return the current settings of NUMERIC DIGITS, NUMERIC FORM, and NUMERIC FUZZ, respectively.

16.2.9 Whole Numbers

Within the set of numbers REXX understands, it is useful to distinguish the subset defined as **whole numbers**. A whole number in REXX is a number that has a decimal part that is all zeros (or that has no decimal part). In addition, it must be possible to express its integer part simply as digits within the precision set by the NUMERIC DIGITS instruction. REXX would express larger numbers in exponential notation, after rounding, and, therefore, these could no longer be safely described or used as whole numbers.

16.2.10 Numbers Used Directly by REXX

As discussed, the result of any arithmetic operation is rounded (if necessary) according to the setting of NUMERIC DIGITS. Similarly, when REXX directly uses a number (which has not necessarily been involved in an arithmetic operation), the same rounding is also applied. It is just as though the number had been added to 0.

In the following cases, the number used must be a whole number, and the largest number you can use is 999999999.

- The positional patterns in parsing templates (including variable positional patterns)
- The power value (right hand operand) of the power operator
- The values of *exprr* and *exprf* in the DO instruction
- The values given for DIGITS or FUZZ in the NUMERIC instruction
- Any number used in the numeric option in the TRACE instruction.

16.2.11 Errors

Two types of errors may occur during arithmetic:

- Overflow or Underflow

  This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is 9, VM supports exponents in the range -999999999 through 999999999.

  Because this allows for (very) large exponents, overflow or underflow is treated as a syntax error.

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail because of lack of storage. This is considered a terminating error as usual, rather than an arithmetic error.

Chapter 17. Conditions and Condition Traps

A **condition** is a specified event or state that CALL ON or SIGNAL ON can trap. A condition trap can modify the flow of execution in a REXX program. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see section 13.3, "CALL" on page 160 and section 13.22, "SIGNAL" on page 192).

```
►►──┬─CALL───┬──┬─OFF──condition──────────────────────┬──;─►◄
    └─SIGNAL─┘  └─ON──condition──┬───────────────────┬─┘
                                 └─NAME──trapname─────┘
```
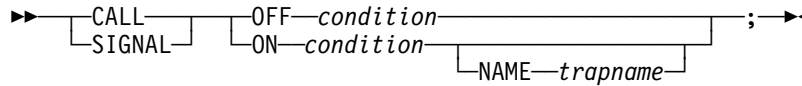
condition and *trapname* are single symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified *condition* occurs, control passes to the routine or label *trapname* if you have specified *trapname*. Otherwise, control passes to the routine or label *condition*. CALL or SIGNAL is used, depending on whether the most recent trap for the condition was set using CALL ON or SIGNAL ON, respectively.

Note: If you use CALL, the *trapname* can be an internal label, a built-in function, or an external routine. If you use SIGNAL, the *trapname* can be only an internal label.

The conditions and their corresponding events that can be trapped are:

ERROR

raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE is active. The condition is raised at the end of the clause that called the command but is ignored if the ERROR condition trap is already in the delayed state. The **delayed state** is the state of a condition trap when the condition has been raised but the trap has not yet been reset to the enabled (ON) or disabled (OFF) state. See note 3 on page 268.

CALL ON ERROR and SIGNAL ON ERROR trap all positive return codes, and negative return codes only if CALL ON FAILURE and SIGNAL ON FAILURE are not set.

FAILURE

raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that called the command but is ignored if the FAILURE condition trap is already in the delayed state.

CALL ON FAILURE and SIGNAL ON FAILURE trap all negative return codes from commands.

HALT

raised if an external attempt is made to interrupt and end execution of the program. The condition is usually raised at the end of the clause that was being processed when the external interruption occurred.

NOVALUE

raised if an uninitialized variable is used:

- As a term in an expression
- As the *name* following the VAR subkeyword of a PARSE instruction
- As a variable reference in a parsing template, a PROCEDURE instruction, or a DROP instruction.

Note: SIGNAL ON NOVALUE can trap any uninitialized variables except tails in compound variables.

```
/* The following does not raise NOVALUE. */
signal on novalue
a.=0
say a.z
say 'NOVALUE is not raised.'
exit

novalue:
say 'NOVALUE is raised.'
```

You can specify this condition only for SIGNAL ON.

SYNTAX

raised if any language processing error is detected while the program is running. This includes all kinds of processing errors, including true syntax errors and "run-time" errors, such as attempting an arithmetic operation on nonnumeric terms. You can specify this condition only for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or DELAY, and any *trapname*) of that condition trap. Thus, a CALL ON HALT replaces any current SIGNAL ON HALT (and a SIGNAL ON HALT replaces any current CALL ON HALT), a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, any OFF reference disables the trap for CALL or SIGNAL, and so on.

17.1 Action Taken When a Condition Is Not Trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

- For HALT and SYNTAX, the processing of the program ends, and a message (see Appendix A, "Error Numbers and Messages" on page 429) describing the nature of the event that occurred usually indicates the condition.

- For all other conditions, the condition is ignored and its state remains OFF.

17.2 Action Taken When a Condition Is Trapped

When a condition trap is currently enabled (ON) and the specified condition occurs, instead of the usual flow of control, a CALL *trapname* or SIGNAL *trapname* instruction is processed automatically. You can specify the *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction. If you do not specify a *trapname*, the name of the condition itself (ERROR, FAILURE, HALT, NOTREADY, NOVALUE, or SYNTAX) is used.

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

The sequence of events, after a condition has been trapped, varies depending on whether a SIGNAL or CALL is processed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see page 192).

  If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it when the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then, if the SIGNAL ON SYNTAX label name is not found, a usual syntax error termination occurs.

- If the action taken is a CALL (which can occur only at a clause boundary), the CALL is made in the usual way (see page 160) except that the call does not affect the special variable RESULT. If the routine should RETURN any data, then the returned character string is ignored.

  Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

  As the condition is raised, and before the CALL is made, the condition trap is put into a delayed state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is raised again, it is either ignored (for ERROR, FAILURE, or NOTREADY) or (for the other conditions) any action (including the updating of the condition information) is delayed until one of the following events occurs:

  1. A CALL ON or SIGNAL ON, for the delayed condition, is processed. In this case a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been processed.

  2. A CALL OFF or SIGNAL OFF, for the delayed condition, is processed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.

  3. A RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

  On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

  Notes:

  1. You must be extra careful when you write a syntax trap routine. Where possible, put the routine near the beginning of the program. This is necessary because the trap routine label might not be found if there are certain scanning errors, such as a missing ending comment. Also, the trap routine should not contain any statements that might cause more of the program in error to be scanned. Examples of this are calls to built-in

functions with no quotation marks around the name. If the built-in function name is in uppercase and is enclosed in quotation marks, REXX goes directly to the function, rather than searching for an internal label.

2. In all cases, the condition is raised immediately upon detection. If SIGNAL ON traps the condition, the current instruction is ended, if necessary. Therefore, the instruction during which an event occurs may be only partly processed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that the CALL for ERROR, FAILURE, HALT, and NOTREADY traps can occur only at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET may be interrupted and later resumed. Similarly, other instructions, for example, DO or SELECT, may be temporarily interrupted by a CALL at a clause boundary.

3. The state (ON, OFF, or DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction (page 160) for details of other information that is saved during a subroutine call.

4. The state of condition traps is not affected when an external routine is called by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.

5. While user input is processed during interactive tracing, all condition traps are temporarily set OFF. This prevents any unexpected transfer of control—for example, should the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program but is trapped specially and then ignored after a message is given.

6. The system interface detects certain execution errors either before execution of the program starts or after the program has ended. SIGNAL ON SYNTAX cannot trap these errors.

Note that a **label** is a clause consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

17.3 Condition Information

When any condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see page 210).

The condition information includes:

- The name of the current trapped condition
- The name of the instruction processed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition
- Any descriptive string associated with that condition.

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is saved and restored across subroutine or function calls, including one because of a CALL ON trap. Therefore, a routine called by a CALL ON can access the appropriate condition information. Any previous condition information is still available after the routine returns.

17.3.1 Descriptive Strings

The descriptive string varies, depending on the condition trapped.

ERROR   The string that was processed and resulted in the error condition.

FAILURE   The string that was processed and resulted in the failure condition.

HALT   Any string associated with the halt request. This can be the null string if no string was provided.

NOVALUE   The derived name of the variable whose attempted reference caused the NOVALUE condition. The NOVALUE condition trap can be enabled only using SIGNAL ON.

SYNTAX   Any string the language processor associated with the error. This can be the null string if you did not provide a specific string. Note that the special variables RC and SIGL provide information on the nature and position of the processing error. You can enable the SYNTAX condition trap only by using SIGNAL ON.

17.4 Special Variables

A special variable is one that may be set automatically during processing of a REXX program. There are three special variables: RC, RESULT, and SIGL. None of these has an initial value, but the program may alter them. (For information about RESULT, see page 189.)

17.4.1 The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code, as usual, before control is transferred to the condition label.

For SIGNAL ON SYNTAX, RC is set to the syntax error number.

17.4.2 The Special Variable SIGL

Following any transfer of control because of a CALL or SIGNAL, the program line number of the clause causing the transfer of control is stored in the special variable SIGL. Where the transfer of control is because of a condition trap, the line number assigned to SIGL is that of the last clause processed (at the current subroutine level) before the CALL or SIGNAL took place. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control a text editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then call an editor to edit the source file positioned at the line in error. Note that in this case you may have to run the program again before any changes made in the editor can take effect.

Alternatively, SIGL can be used to help determine the cause of an error (such as the occasional failure of a function call) as in the following example:

```
signal on syntax
a = a + 1        /* This is to create a syntax error */
say 'SYNTAX error not raised'
exit

/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl':' "ERRORTEXT"(rc)
  say "SOURCELINE"(sigl)
  trace ?r; nop
```

This code first displays the error code, line number, and error message. It then displays the line in error, and finally drops into debug mode to let you inspect the values of the variables used at the line in error.

Chapter 18. REXX/CICS Text Editor

REXX/CICS provides a general purpose CICS-based text editor, patterned after VM/CMS XEDIT.

The editor is provided so that execs and data can be created, updated, and viewed from within the CICS environment.

The REXX/CICS editor includes several prefix commands (for example: C, CC, M, MM, B, A, F, P) in common with the XEDIT editor. To use the editor, enter `EDIT`.

While in an editor session, commands entered on the command line can be chained by placing a ";" between each command. Support is also provided for macros written in REXX. This lets you customize editor settings with a profile exec, add new commands to the editor, or use the editor facilities as part of an application.

Note: The REXX/CICS Text Editor does not support binary files.

18.1 Invocation

You can start a REXX/CICS edit session from a CICS terminal (clear screen), a REXX/CICS session, another edit session, or a REXX exec. You can enter the editor by executing the CICS EDIT transaction identifier (or your site-defined transaction identifier for the REXX/CICS editor). From a clear REXX/CICS screen enter `EDIT` followed by a REXX File System (RFS) file identifier and an edit session is issued for this file.

Note: If you specify REXX as a CICS transaction identifier with no exec name, the IBM supplied REXXTRY interactive utility (CICRXTRY exec) is issued. REXXTRY provides an interactive shell for performing REXX statements and commands.

While you are in the editor, you can start another edit session by entering `EDIT` *fileid*. The syntax for this method of invocation is defined in the EDIT command, section 18.5.13, "EDIT" on page 281. When you write an exec, you can start the REXX/CICS editor session by issuing the CICS EDIT transaction identifier as a command. If a file ID is not specified when you invoke the editor, the file ID defaults to file NONAME in your current RFS directory. A fully qualified file ID is: *poolid:\dir1\dir2\fn.ft*

However, you can specify a partial file ID if the target file is in (or is going to be created in) your current directory (as specified by the CD command).

Examples of how you can call the editor:

```
EDIT TEST.EXEC (From terminal using CICS transaction ID EDIT)
EDIT TEST.EXEC (From within an edit session using the EDIT editor command)
'EDIT TEST.EXEC' (From within an exec using the EDIT command)
'EDIT TEST.EXEC' (From terminal in a REXX/CICS REXXTRY session)
```

This example shows four different ways to start an edit session. The first example starts a session from REXX/CICS. The second starts a new session from an existing session. The third starts a session from a REXX exec. The fourth starts a session from a REXX/CICS REXXTRY session.

18.2 Screen Format

When you call the editor without a profile, the default screen definition is displayed as shown in the following figure.

```
EDIT ---- POOL1:\USERS\USER1\TEST ------------------------------ COLUMN 1 73
COMMAND ===>
00000 **************************** TOP OF DATA ************************
00001
00002
00003
00004
00005
00006
00007
00008
00009
00010
00011
00012
00013 *************************** BOTTOM OF DATA *********************




F1=HELP F2=LADD F3=FIL F4=SPLT F5=F F6=JN F7=BA F8=FWD F10=LFT F11=RGT F12=QUI
```

The first line is reserved for the **title line** that contains the following:

- Fully qualified file ID
- Column numbers
- Displayed messages.

The informational lines that let you know where the top and bottom of the file is are the only lines that are displayed when you bring up a new edit session. These lines consist of a **prefix area** and a **data area**. The line where "*** TOP OF DATA ***" is displayed, in this example, is the current line. This is distinguished, from the rest, by being highlighted and it is the line most of the command line commands affect. The **command line** lets you interact with the editor on a high level basis. All of the screen lines, except for the title line, may be moved for convenience.

18.3 Prefix Commands

Several prefix commands are provided by the editor. These commands are entered in the prefix area and give you the ability to copy, move, insert, delete, and replicate lines on either an individual or a consecutive block basis.

18.3.1.1 Individual Line Commands
The following commands work with individual lines and consist of one character:

/	Specify current line in file
I	Insert a line
D	Delete a line
C	Copy a line
M	Move a line

R	Replicate a line
"	Synonym for replicate

When you enter one of the previous commands in the prefix area of a line, the command performs its respective function on that line. If you enter an "I" in the prefix area of line 00000, the editor opens a new line just below it for input. You can also append a number to the end of the prefix command. This acts as a replication factor. If the number "5" is appended to the "I," five lines are opened for input instead of one.

18.3.1.2 Consecutive Block Commands

The following commands work with consecutive blocks of lines and consist of two characters:

DD	Delete a block of lines
CC	Copy a block of lines
MM	Move a block of lines
RR	Replicate a block of lines
""	Synonym for replicate

Block commands are processed in pairs. You place one command in the prefix area of the first line of the block and place the same command in the prefix area of the last line of the block. For example, to delete a block of lines, you place a "DD" in the prefix area of line 00001 and another "DD" in the prefix area of line 00005. This deletes all lines between line 00001 and line 00005, inclusive. The only block prefix command that allows a replication factor is the replicate command. So, when you specify a number with the replicate block command you need to know how many times you want the block (not individual lines) replicated. The replication factor must be specified with the first RR replication command.

18.3.1.3 Destination Commands

The following commands are called destination prefix commands:

A	After
B	Before
F	Following
P	Preceding

These commands give the move and copy prefix commands a destination for a block of text. When you enter these in the prefix area, the text from a copy or a move is placed either before or after that line according to what destination command is specified. The replicate prefix command does not use a destination prefix command. Instead, it places its output immediately after the block that is to be replicated.

18.4 Macros Under the REXX/CICS Editor

The editor supports REXX macros, giving the macros the ability to alter the editor settings and display the editor screens. Macros can process all of the editor command line commands.

Example:

```
/* Macro to alter the setting of the REXX/CICS editor */
ADDRESS EDITSVR
'SET NUMBERS OFF'
'SET CURLINE 10'
'SET MSGLINE 2'
'SET CMDLINE TOP'
'SET CASE MIXED IGNORE'
```

This example addresses the editor command environment and alters the editor settings.

Example:

```
/* Macro to use the REXX/CICS editor as an I/O interface */
ADDRESS EDITSVR
'INPUT Some'
'INPUT More'
'INPUT DATA'
'DISPLAY'
```

This example enters the text `Some`, `More`, and `DATA` on three separate lines and then displays the current edit screen.

18.5 Command Line Commands

The syntax and description for each of the command line commands follows.

18.5.1 ARBCHAR

```
►►──ARBCHAR──arbchar──────────────────────────────────────►◄
```

ARBCHAR sets the arbitrary characters.

18.5.1.1 Operands
arbchar   specifies a printable (typeable) character.

18.5.1.2 Return Codes
0   Normal return
202   Invalid operand

18.5.1.3 Example
```
'ARBCHAR .'
```

This example defines the character "." as being the arbitrary character.

18.5.1.4 Note
The arbitrary character takes the place of text in a string. The default value for the arbchar is a ".". For more information, on how you can do searches with the arbitrary character, see the FIND command, section 18.5.16, "FIND" on page 283.

18.5.2 ARGS

```
           (1)
►►──ARGS──┬─────────────┬──────────────────────────►◄
          └─ arguments ─┘
```

Note:

[1] If *arguments* is not specified, any previously defined arguments are deleted.
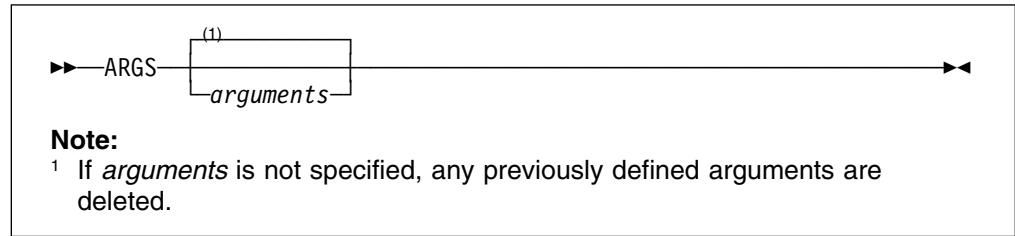
ARGS stores the default parameters to be passed to the program being edited when invoked with the text editor EXEC command.

18.5.2.1 Operands

arguments

specifies the parameter string to be passed. If you do not specify *arguments*, any previously defined arguments are deleted.

18.5.2.2 Return Codes

0   Normal return

18.5.2.3 Example

```
'ARGS A B C'
'EXEC'
'ARGS'
```

The first line of this example defines the arguments to be passed as `A`, `B`, and `C`. The second line executes the last saved copy of the file that is currently being edited, passing it the arguments defined in line one. The last line deletes the arguments.

18.5.3 BACKWARD

```
           ┌─ 1 ─┐
►►──BAckward──┼─ n ─┼──────────────────────────────────►◄
           └─ * ─┘
```
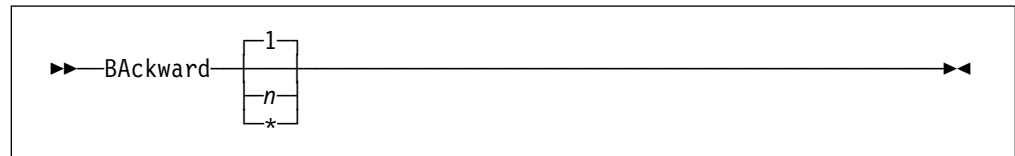
BACKWARD scrolls backward toward the beginning of a file for a specified number of screen displays.

18.5.3.1 Operands

n   specifies the number of screen displays you want to scroll backward. If you specify an asterisk (*), the screen scrolls to the top of the file and the current line is set to the top of the file. If *n* is not specified, the screen scrolls back one display.

18.5.3.2 Return Codes
0   Normal return
202   Invalid operand

18.5.3.3 Example
`'BACKWARD'`

This example scrolls one screen toward the top of the file.

18.5.3.4 Note
The editor, by default, sets PF7 to BACKWARD and PF8 to FORWARD.

18.5.4 BOTTOM

```
►►──BOTTOM──────────────────────────────────────────────►◄
```

BOTTOM scrolls to the bottom of the file.

18.5.4.1 Return Codes
0   Normal return

18.5.4.2 Example
`'BOTTOM'`

This example scrolls to the bottom of the file.

18.5.5 CANCEL

```
►►──CANCEL──────────────────────────────────────────────►◄
```
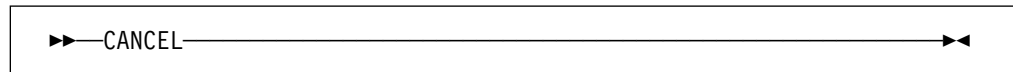
CANCEL ends the current edit session without saving the changes.

18.5.5.1 Return Codes
0   Normal return
210   Request failed

18.5.5.2 Example
`'CANCEL'`

This example quits the current editor session unconditionally, without saving any file changes.

18.5.5.3 Notes
1. CANCEL lets you exit the editor without saving changes, and without any warning messages that changes have been made.

2. CANCEL is a synonym for QQUIT.

18.5.6 CASE

```
►►──CASE─────────┬────(1)────────────────────────────────────►◄
                 ├─MIXED─┬─RESPECT─┤
                 │       └─IGNORE──┤
                 └─UPPER─┬─RESPECT─┤
                         └─IGNORE──┘
```

Note:
[1] The default is set in the user profile.
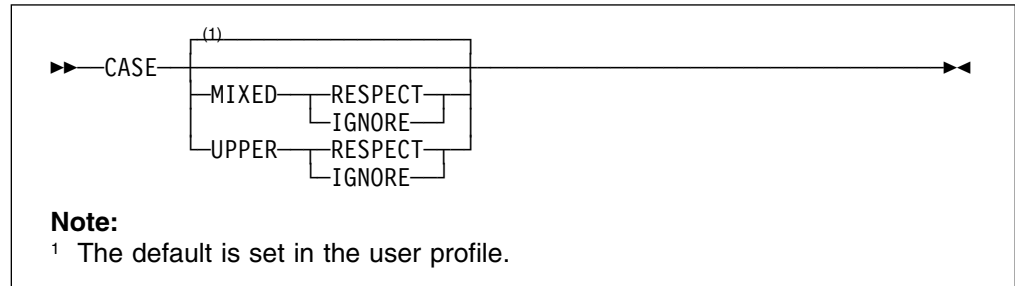
CASE sets the case translation and interpretation preferences.

18.5.6.1 Operands
UPPER translates lowercase characters to uppercase when entered.

MIXED works with each character in its original form.

RESPECT
       respects the case of each character while doing a search.

IGNORE ignores the case of each character while doing a search.

18.5.6.2 Return Codes
0   Normal request
202 Invalid operand

18.5.6.3 Example
```
'CASE MIXED RESPECT'
```

This example sets the case to MIXED and the sensitivity to RESPECT. For more information on sensitivity, see the FIND command, section 18.5.16, "FIND" on page 283.

18.5.7 CHANGE

```
►►──CHANGE──/string1/string2/──┬──────┬──────────────────────►◄
                               └─ALL─┘
```
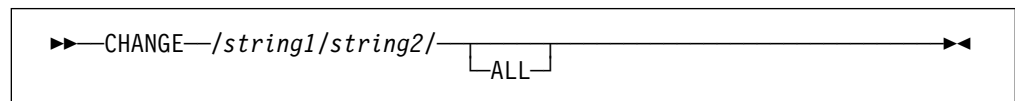
CHANGE changes a string in the file.

18.5.7.1 Operands
string1 specifies the string being replaced.

string2 specifies the string that replaces *string1*.

ALL is a keyword indicating that all occurrences on all lines, from the current line, to the end of the file, are changing.

/ is the delimiting character.

18.5.7.2 Return Codes
0   Normal return
202 Invalid operand
210 Request failed
223 Search argument not found

18.5.7.3 Example
`'CHANGE /noeditor/editor/'`

This example replaces the first occurrence of `noeditor` with `editor`.

18.5.7.4 Note
The CHANGE command respects the sensitivity settings of the editor when locating *string1* and when changing *string2* as defined by the CASE command or the system default.

18.5.8 CMDLINE

```
►►──CMDLINE──┬─TOP────┬──────────────────────────────►◄
             └─BOTTOM─┘
```
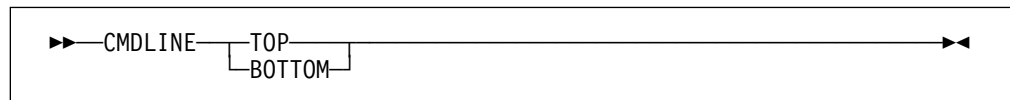
CMDLINE sets the command line display preferences.

18.5.8.1 Operands
TOP   displays the command line on the second line of the screen.

BOTTOM displays the command line on the bottom line of the screen.

18.5.8.2 Return Codes
0   Normal return
202 Invalid operand

18.5.8.3 Example
`'CMDLINE TOP'`

This example places the command line on the second line of the screen.

18.5.9 CTLCHAR

```
►►──CTLCHAR──┬─character─┬──┬─ESCAPE────┬──────────────►◄
             └─OFF───────┘  ├─PROTECT───┤
                            ├─NOPROTECT─┤
                            └─OFF───────┘
```
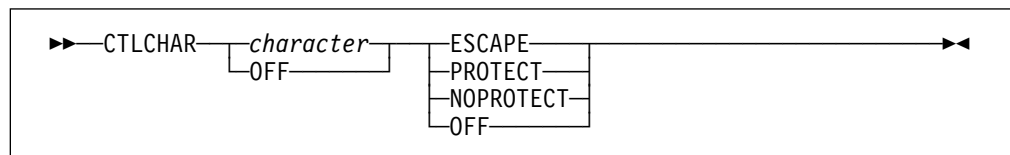
CTLCHAR sets a control character's function.

18.5.9.1 Operands

character specifies a control character to use.

OFF drops all definitions of control characters if OFF is used without specifying a character or drops the specific character if one is specified.

ESCAPE specifies that the following character in the string, passed to the RESERVED command, is a control character.

PROTECT specifies that the string passed to the RESERVED command is protected from user input.

NOPROTECT

specifies that user input, passed to the RESERVED command, is allowed on this string.

18.5.9.2 Return Codes

0 Normal return
202 Invalid operand

18.5.9.3 Example

```
'CTLCHAR ! ESCAPE'
'CTLCHAR % PROTECT'
'RESERVED 20 HIGH !% Important Info'
```

This example defines ! as the escape character and % as the field protection character. After you enter these commands, the screen line 20 will be protected and contain the text that follows the control characters, !%.

18.5.10 CURLINE

```
►►──CURLINE──number────────────────────────────────►◄
```

CURLINE sets the current line display preferences.

18.5.10.1 Operands

number specifies the screen line number.

18.5.10.2 Return Codes

0 Normal return
202 Invalid operand

18.5.10.3 Example

```
'CURLINE 3'
```

This example sets the current display line to screen line 3.

18.5.10.4 Note

The current line is displayed at the screen line number specified in this command. However, the current line cannot be displayed on line 1 because line 1 is reserved for the title line.

18.5.11 DISPLAY

```
►►──DISPLAY──────────────────────────────────────────────►◄
```
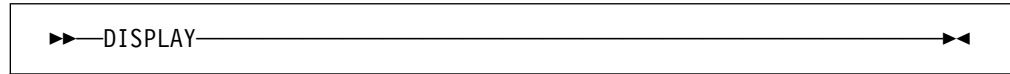
DISPLAY shows the current edit screen.

18.5.11.1 Return Codes

0   Normal return
210 Request failed

18.5.11.2 Example

```
'DISPLAY'
```

This example displays the current edit screen.

18.5.11.3 Note

The DISPLAY command is only useful when it runs from a macro. It displays the current edit session's screen. When it runs from a normal terminal edit session there is no noticeable effect.

18.5.12 DOWN

```
                  ┌─1──────┐
►►──DOWN──┼────────┼──────────────────────────────►◄
                  └─number─┘
```
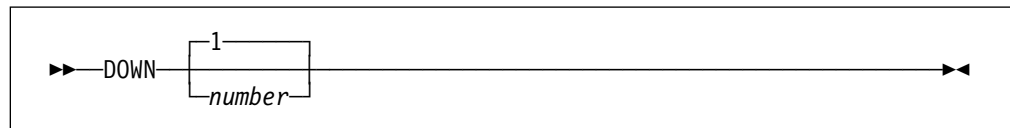
DOWN scrolls forward in the file.

18.5.12.1 Operands

number   specifies the number of lines to scroll. If you do not specify *number*, the screen moves down only one line.

18.5.12.2 Return Codes

0   Normal return
202 Invalid operand

18.5.12.3 Example

```
'DOWN 5'
```

This example scrolls forward through the file five lines.

18.5.13 EDIT

```
►►──EDIT──┬─────NONAME─────┬──────────────────────────────►◄
          ├──fileid────────┤
          └──lib.sublib(mem.type)──LIB──┘
                                              └──(──MACRO──macroname──┘
```
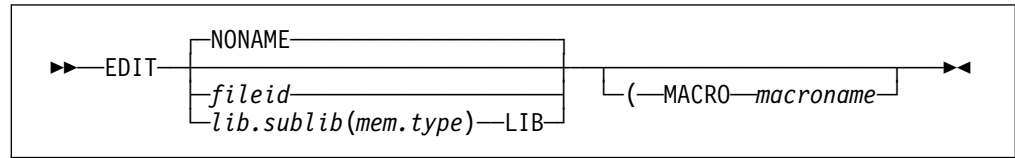
EDIT opens a new edit session.

18.5.13.1 Operands

fileid   specifies the file ID of the file to be created or edited.

lib.sublib(mem.type)
         specifies a VSE Librarian sublibrary and member to be edited.

LIB   is a keyword that follows a VSE Librarian sublibrary member name when a sublibrary member is being edited.

MACRO is a keyword specifying a group of instructions applied to the file being edited.

macroname
         specifies the file name portion of the profile macro file ID (REXX exec name).

18.5.13.2 Return Codes

0   Normal return
203 File not found
204 Not authorized
211 Invalid file ID
226 File is currently being edited
299 Internal error

18.5.13.3 Example

`'EDIT TEST.EXEC'`

This example opens an edit session for the file TEST.EXEC.

18.5.13.4 Notes

1. The directory used when editing a file, is determined as follows:

   • If a fully qualified directory ID is explicitly given, then it is always used.

   • If a partially qualified file ID is specified, then the current directory and path are searched in an attempt to find an existing file that would match the file ID (if it were fully resolved using that directory name).

     – If such a match is successful, then the edit session is for an existing file and the file ID is fully resolved using the directory the file was found in.

     – If the file is not located in the search order, then an edit session for a new file is created, with the fully resolved file ID specifying the current working directory (as specified by the CD command).

2. A VSE Librarian sublibrary member is distinguished by using the keyword LIB after the member name. Members must be enclosed in parentheses, quotes are not allowed.

3. The default user profile macro that the editor tries to call is CICEPROF. The CICEPROF macro creates an ISPF/PDF like environment. A second profile macro, named CICXPROF, is provided. CICXPROF creates a VM/CMS XEDIT like environment.

4. If a file ID or VSE Librarian sublibrary member name is not specified, an RFS file with the special name, NONAME, is created.

18.5.14 EXEC

```
►►──EXEC──────────────(1)──────────────────────────►◄
           └─REXX_program─┬──────────────┬─┘
                          └─arguments────┘
```

Note:
[1] If *REXX_program* is not specified, the last saved copy of the file that is currently being edited is executed.
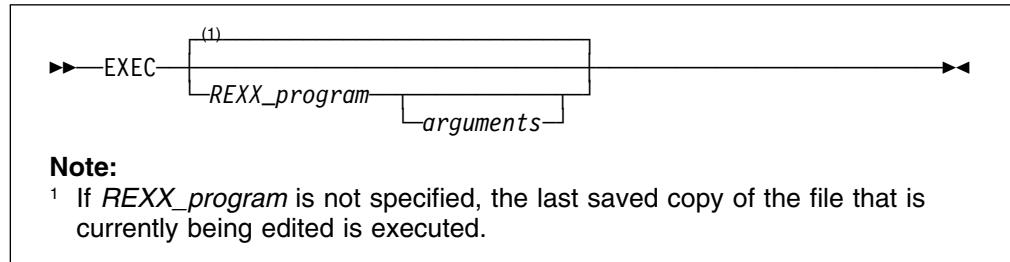
EXEC executes a REXX program within an editor session.

18.5.14.1 Operands
REXX_program
> specifies the REXX program name to execute. If you do not specify a name, the last saved copy of the file that is currently being edited will be executed. If any arguments have been defined using the ARGS command, these will be passed.

arguments
> specifies the arguments to be passed to the program specified in the *REXX_program* operand.

18.5.14.2 Return Codes
n   specifies the return code set by the exit of the called exec
0   Normal return
-3   Exec not found
-10 Exec name not specified
-11 Invalid exec name
-12 GETMAIN error
-99 Internal error

18.5.14.3 Example
```
'EXEC TEST1.EXEC X Y Z'
```

This example executes program TEST1.EXEC and passes X, Y, and Z as arguments.

18.5.15 FILE

```
►►──FILe──────(1)──────────────────────►◄
        └─fileid─┘
```

Note:
[1] If *fileid* is not specified, the file is saved as the default file ID.
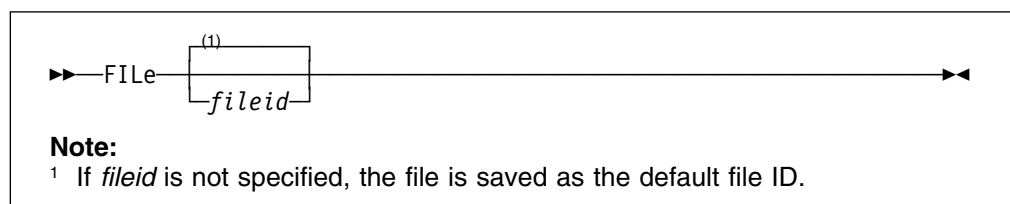
FILE saves the current file being edited.

18.5.15.1 Operands
fileid   specifies the file ID of the file. If you do not specify *fileid*, the file is saved as the default file ID.

18.5.15.2 Return Codes
0   Normal return
202 Invalid operand
204 Not authorized
207 Insufficient space in file pool
210 Request failed

18.5.15.3 Example
```
'FILE'
```

This example saves the current file being edited, using the current file ID specification for the edit session. The current file ID is initially taken from the file ID specified on the edit command, when an edit session is created.

18.5.16 FIND

```
>>--Find--+--previous_searcharg--+-------------------><
           |                      |
           +--searcharg-----------+
```
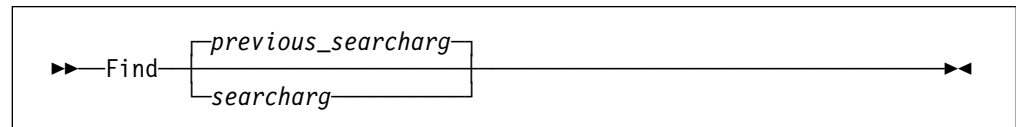
FIND locates a string of text in the file.

18.5.16.1 Operands
searcharg   specifies the text string to be searched on. If you do not specify *searcharg*, a search is performed on the previous search string (*previous_searcharg*).

18.5.16.2 Return Codes
0   Normal return
202 Invalid operand
223 Search argument not found

18.5.16.3 Examples
```
'FIND REDT'
```

This example finds the first occurrence of `REDT`.

```
'FIND Redt'
```

If CASE is set to RESPECT then this example will not find the first occurrence of `REDT`. It will find the first occurrence of `Redt`. For more information, see the CASE command, section 18.5.6, "CASE" on page 277.

The *searcharg* can contain the arbitrary character, in which case the arbitrary character represents any text string which might be imbedded at the arbitrary character's location.

```
'FIND ONE.THREE'
```

This example finds the first occurrence of any string with `ONE` and `THREE` joined by another string.

18.5.16.4 Notes

1. When the RESPECT flag is set with the CASE command, the case of the *searcharg* is respected.

2. The search begins at the current line and continues downward until BOTTOM OF DATA is reached, or a match is made. If BOTTOM OF DATA is reached without a match, then the current line remains where it was before the FIND was processed, rather than making BOTTOM OF DATA the current line.

18.5.17 FORWARD

```
►►──FORward──┬───1───┬─────────────────────────►◄
             ├───n───┤
             └───*───┘
```
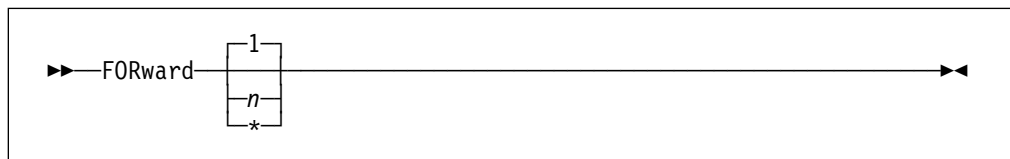
FORWARD scrolls forward toward the end of the file for a specified number of screen displays.

18.5.17.1 Operands

n   specifies the number of screen displays you want to scroll forward. If you specify an asterisk (*), the screen scrolls to the bottom of the file and the current line is set to the last line of data. If *n* is not specified, the screen scrolls forward one display.

18.5.17.2 Return Codes

0   Normal return
202   Invalid operand

18.5.17.3 Example

```
'FORWARD'
```

This example scrolls one screen toward the end of the file.

18.5.17.4 Note

The editor, by default, sets PF7 to BACKWARD and PF8 to FORWARD.

18.5.18 GET

```
►►──GET──┬──────────┬─────────────────────────────►◄
         └─ fileid ─┘
```
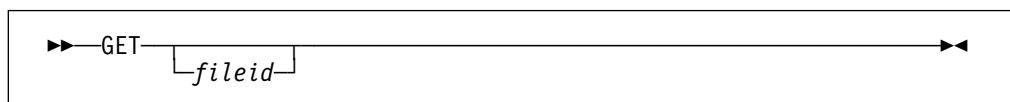
GET imports an RFS file into the current edit session.

18.5.18.1 Operands
fileid   specifies the file ID of the file.

18.5.18.2 Return Codes
0   Normal return
203 File not found
204 Not authorized
210 Request failed

18.5.18.3 Example
```
'GET POOL1:\USERS\USER1\TEST.EXEC'
```

This example pulls the REXX File System file TEST.EXEC in after the current line.

18.5.19 GETLIB

```
►►──GETLIB──lib.sublib(mem.type)─────────────────────────────►◄
```

GETLIB imports a member from a VSE Librarian sublibrary into the current edit session. The file is inserted after the current line.

18.5.19.1 Operands
lib.sublib(mem.type)
            specifies a VSE Librarian sublibrary and member name.

18.5.19.2 Return Codes
0   Normal return
203 File not found
204 Not authorized
210 Request failed

18.5.19.3 Example
```
'GETLIB MYSLIB.PROJ1(MEM1.PROC)'
```

This example gets member MEM1.PROC from sublibrary MYSLIB.PROJ1 and puts it after the current line in an edit session.

Note: GETLIB is the VSE equivalent of the GETPDS function available with CICS/ESA.

18.5.20 INPUT

```
                            (1)
►►──Input──────────────────────────────────────────────────►◄
           └─text─┘
```

Note:
[1] If *text* is not specified, the new line is blank.

INPUT inserts a new line after the current line.

18.5.20.1 Operands
text   specifies the text being inserted on the new line. If you do not specify *text*, the new line is blank.

18.5.20.2 Return Codes
0   Normal return

18.5.20.3 Example
```
'INPUT Test Input Data'
```

This example places the text `Test Input Data` on a newly inserted line after the current line.

18.5.21 JOIN

```
►►──JOIN──────────────────────────────────────────────►◄
```
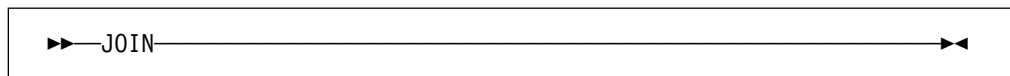
JOIN joins two lines into one.

18.5.21.1 Return Codes
0   Normal return
210 Request failed

18.5.21.2 Example
```
'JOIN'
```

This example joins the line that the cursor is on with the line immediately following it.

18.5.22 LEFT

```
            ┌─1──────┐
►►──LEFT────┤        ├────────────────────────────────►◄
            └─number─┘
```
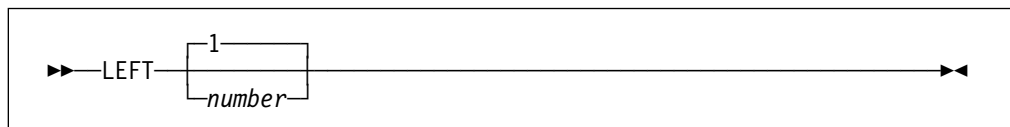
LEFT scrolls left in the file.

18.5.22.1 Operands
number   specifies the number of characters to scroll. If you do not specify *number*, the screen scrolls left one character in the file. If you specify `0` for *number*, the file scrolls to the far left side.

18.5.22.2 Return Codes
0   Normal return
202 Invalid operand
210 Request failed

18.5.22.3 Example

```
'LEFT 20'
```

This example scrolls left 20 characters to the left.

18.5.23 LINEADD

```
►►──LINEADD────────────────────────────────────────────────►◄
```
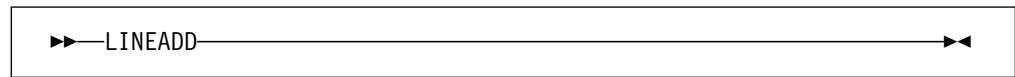
LINEADD adds a blank line after the cursor line.

18.5.23.1 Return Codes

0   Normal return
230 Cursor is not in file area

18.5.23.2 Example

```
'PFKEY 2 LINEADD'
```

This example causes the addition of a blank line after the line where the cursor resides (if it is a file line) whenever PF2 is pressed.

18.5.23.3 Note

LINEADD is mainly useful when assigned to a program function (PF) key. It is by default assigned to PF2.

18.5.24 LPREFIX

```
►►──LPREFIX──*prefix*──────────────────────────────────────►◄
```
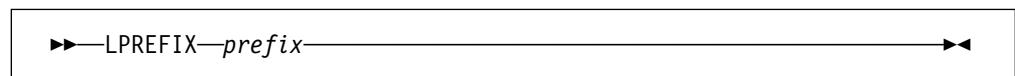
LPREFIX enters a prefix command into the current line prefix area.

18.5.24.1 Operands

prefix   specifies any standard prefix (such a C, CC, M, MM, B, A) that is entered during an edit session.

18.5.24.2 Return Codes

0   Normal return
202 Invalid operand

18.5.24.3 Example

```
'LPREFIX D'
```

This example causes the deletion of the current file line.

18.5.24.4 Note

LPREFIX is provided to let you use the prefix commands from within edit macros.

18.5.25 MACRO

```
►►──MACRO──fileid──────────────────────────────────────────────►◄
```

MACRO calls a macro.

18.5.25.1 Operands
fileid   specifies the file ID of the macro you want to run. If this file ID includes a file type suffix, then an attempt is made to call an exec with that suffix. Otherwise, an attempt is made to call an exec whose suffix is EXEC.

18.5.25.2 Return Codes
n   specifies the return code set by the exit of the called exec
0   Normal return
-3   Exec not found
-10   Exec name not specified
-11   Invalid exec name
-12   GETMAIN error
-99   Internal error

18.5.25.3 Example
```
'MACRO POOL1:\USERS\USER1\TEST'
```

This example calls the macro, POOL1:\USERS\USER1\TEST.EXEC.

18.5.25.4 Note
Macros have the ability to make calls to the REXX/CICS editor server. Any command that you can enter from the command line of the editor can be run from a macro.

18.5.26 MSGLINE

```
►►──MSGLINE──┬─number─┬──────────────────────────────────────►◄
             ├─OFF────┤
             └─INFO───┘
```

MSGLINE sets the message line display preferences.

18.5.26.1 Operands
number   displays the message line on the corresponding screen line.

OFF   does not display the message line.

INFO   displays messages in the header line.

18.5.26.2 Return Codes
0   Normal return
202   Invalid operand

18.5.26.3 Example
```
'MSGLINE 2'
```

This example places the message line on screen line 2.

18.5.27 NULLS

```
►►──NULLS──┬─ON──┬──────────────────────────────────────────►◄
           └─OFF─┘
```
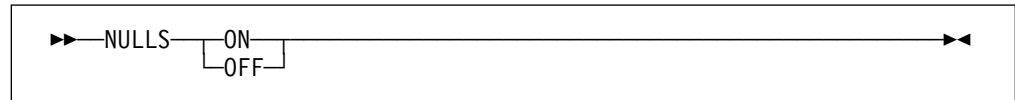
NULLS controls whether the fields on the screen will be written with trailing blanks or trailing nulls.

18.5.27.1 Operands
ON   specifies that fields on the screen are written with trailing nulls.

OFF   specifies that fields on the screen are written with trailing blanks.

18.5.27.2 Return Codes
0   Normal return
202 Invalid operand

18.5.27.3 Example
```
'NULLS ON'
```

This example causes trailing nulls on the fields of the screen.

18.5.28 NUMBERS

```
►►──NUMbers──┬─ON──┬────────────────────────────────────────►◄
             └─OFF─┘
```
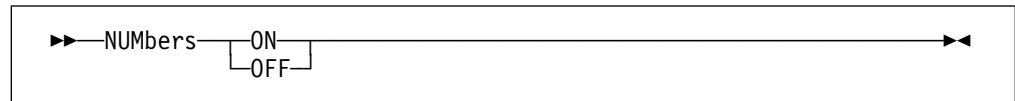
NUMBERS sets the prefix area display preferences.

18.5.28.1 Operands
ON   displays sequential numbers in the prefix area.

OFF   displays equal signs in the prefix area.

18.5.28.2 Return Codes
0   Normal return
202 Invalid operand

18.5.28.3 Example
```
'NUMBERS ON'
```

This example displays sequential numbers in the prefix area.

18.5.28.4 Note
Line number sequencing is not done on the data within the edit session, but are pseudo line numbers associated with the file lines during the edit session only.

18.5.29 PFKEY

```
►►─PFkey─number─┬──────────────(1)──────┬─►◄
                └─text─┘
```
Note:
[1] If *text* is not specified, the PF key is processed.
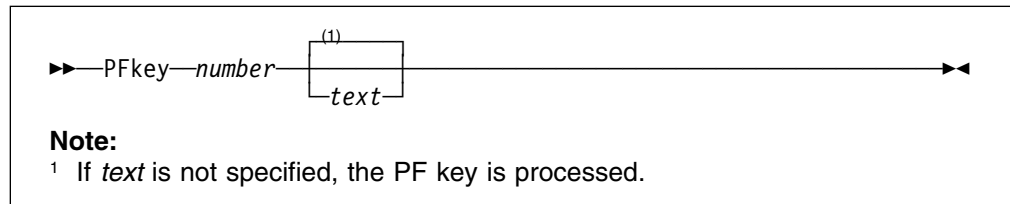
PFKEY sets or processes a program function (PF) key.

18.5.29.1 Operands
number   specifies the PF key that is set or processed.

text   specifies the text that the PF key is set to. If you do not specify *text*, the PF key is processed.

18.5.29.2 Return Codes
0   Normal return
202   Invalid Operand
229   Number out of range

18.5.29.3 Example
```
'PFKEY 3 quit'
'PFKEY 3'
```

This example first sets PFKEY 3 to quit and then processes the PF key.

18.5.30 PFKLINE

```
►►─PFKLINE─┬─TOP────┬─►◄
           ├─BOTTOM─┤
           ├─number─┤
           └─OFF────┘
```
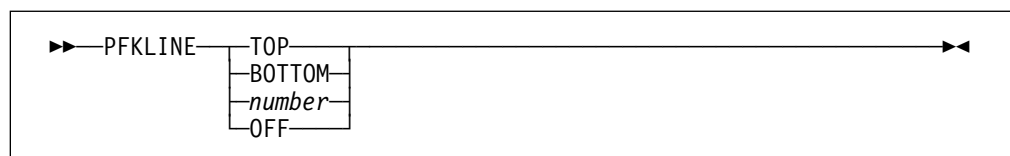
PFKLINE sets the program function (PF) key line display preferences.

18.5.30.1 Operands
TOP   displays the PF key line on the second line of the screen.

BOTTOM displays the PF key line on the bottom line of the screen.

number   specifies the screen line number.

OFF   removes the PF key from the display screen.

18.5.30.2 Return Codes

0   Normal return
202 Invalid operand

18.5.30.3 Example

`'PFKLINE BOTTOM'`

This example places the PF key line on the bottom line of the screen.

18.5.31 QQUIT

```
►►──QQuit──────────────────────────────────────────►◄
```

QQUIT ends the current edit session without saving changes.

18.5.31.1 Return Codes

0   Normal return

18.5.31.2 Example

`'QQUIT'`

This example quits the current editor session unconditionally, without saving any file changes.

18.5.31.3 Notes

1. QQUIT lets you exit the editor without saving changes, and without any warning messages that changes have been made.

2. A synonym for QQUIT is CANCEL.

18.5.32 QUERY

```
►►──QUERY──┬──CHANGES──┬──────────────────────────►◄
           ├──CMDLINE──┤
           ├──COLUMN───┤
           ├──DIR──────┤
           ├──FILEID───┤
           ├──MSGLINE──┤
           ├──NULLS────┤
           ├──NUMBERS──┤
           ├──PFKLINE──┤
           ├──PFKEY.n──┤
           └──RECORDS──┘
```

QUERY displays the current settings of the editor.

18.5.32.1 Operands

CHANGES

   displays the number of modifications to the file since it was last saved.

CMDLINE

   displays the current setting of the command line. For more information see the Text Editor command, section 18.5.8, "CMDLINE" on page 278.

COLUMN displays the starting column in the file that is displayed on the screen.

DIR displays the directory that is associated with the file.

FILEID displays the name of the file being edited.

MSGLINE
displays the current setting of the message line. For more information see the Text Editor command, section 18.5.26, "MSGLINE" on page 288.

NULLS displays the current setting of NULLS. For more information see the Text Editor command, section 18.5.27, "NULLS" on page 289.

NUMBERS
displays the current setting of NUMBERS. For more information see the Text Editor command, section 18.5.28, "NUMBERS" on page 289.

PFKLINE displays the current setting of the PFKLINE. For more information see the Text Editor command, section 18.5.30, "PFKLINE" on page 290.

PFKEY.*n* displays the command that is processed if PFKEY *n* is pressed. The *n* is any number from 1 to 24.

RECORDS
displays the number of lines in the file.

18.5.32.2 Return Codes
0   Normal return
202   Invalid operand
236   Not defined

18.5.32.3 Example
`'QUERY PFKEY.1'`

This example displays the command that is processed when PFKEY 1 is pressed.

18.5.33 QUIT

```
►►──QUIT───────────────────────────────────────────►◄
```

QUIT ends the current edit session.

18.5.33.1 Return Codes
0   Normal return
210   Request failed

18.5.33.2 Example
`'QUIT'`

This example exits the editor.

18.5.33.3 Note

When the current file has been changed, the editor does not let you exit until either a save is done or you enter the QQUIT command.

18.5.34 RESERVED

```
>>--REServed--line---HIGH-------------------------------------->◄
                    |-NOHIGH-|
                    |-OFF----|  |-text-|
```
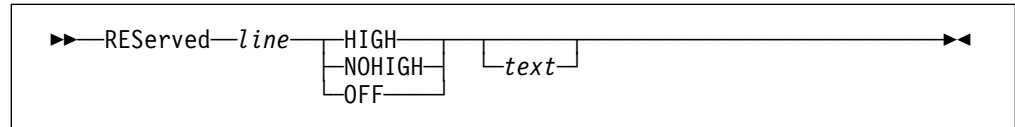
RESERVED reserves a line on the screen for your output.

18.5.34.1 Operands

line   specifies the line that is reserved and the text is displayed.

HIGH   is a keyword specifying that the text is highlighted.

NOHIGH   is a keyword specifying that the text is the usual intensity.

OFF   is a keyword specifying that the line is freed from its reserved state.

text   specifies a string of text, with optional control characters, that is displayed on the reserved line.

18.5.34.2 Return Codes

0   Normal return
202   Invalid operand
210   Request failed
229   Number out of range

18.5.34.3 Example

```
'CTLCHAR ! ESCAPE'
'CTLCHAR % PROTECT'
'RESERVED 20 HIGH !% Important Info'
```

This example displays `Important Info` in a high intensity, protected field on screen line 20.

18.5.35 RESET

```
>>--RESET------------------------------------------------------>◄
```
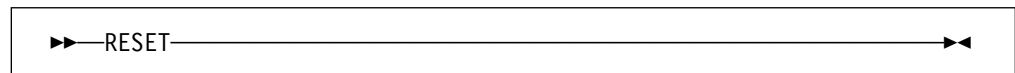
RESET terminates any pending prefix commands.

18.5.35.1 Return Codes

0   Normal return

18.5.35.2 Example
```
'RESET'
```

This example cancels all pending prefix commands.

18.5.36 RIGHT

```
►►──RIght──┬──────1──────┬──────────────────────────────►◄
           └──number──┘
```
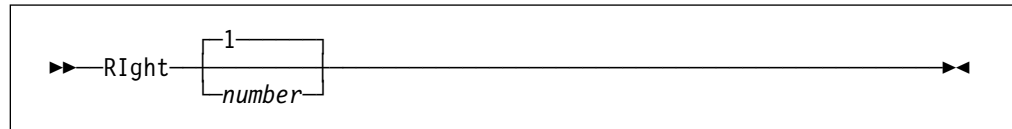
RIGHT scrolls right in the file.

18.5.36.1 Operands
number specifies the number of characters to scroll. If you do not specify
         number, the screen scrolls to the right one character in the file. If you
         specify 0 for *number*, the file scrolls to the far right.

18.5.36.2 Return Codes
0   Normal return
202   Invalid operand

18.5.36.3 Example
```
'RIGHT 20'
```

This example scrolls 20 characters to the right in the file.

18.5.36.4 Note
If the value you specified causes a target outside of the record, scrolling stops at
the right side of the record.

18.5.37 SAVE

```
►►──Save──┬────(1)────┬──────────────────────────────────►◄
          └──fileid──┘
```
Note:
¹ If *fileid* is not specified, the file is saved as the default file ID.
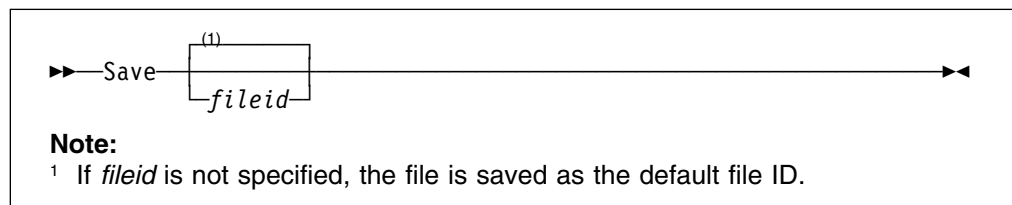
SAVE saves a file to an RFS file or VSE Librarian sublibrary member.

18.5.37.1 Operands
fileid specifies the file ID of the file. If you do not specify *fileid*, the file is
         saved as the default file ID.

18.5.37.2 Return Codes
0   Normal return
202   Invalid operand
207   Insufficient space in file pool
210   Request failed

18.5.37.3 Example
```
'SAVE SYSTEM:\USERS\USER1\TEST.EXEC'
```

This example saves the current file to the RFS and names it,
SYSTEM:\USERS\USER1\TEST.EXEC.

18.5.38 SORT
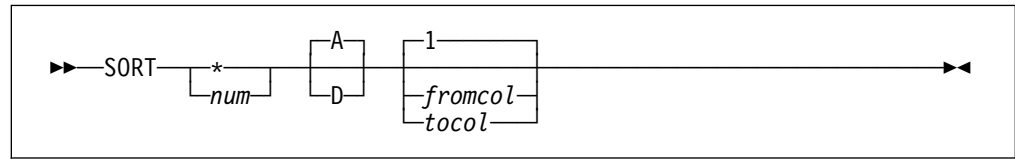
SORT sorts the lines from the current line on down.

18.5.38.1 Operands
*   specifies that all the lines from the current line to the end of the file are
              sorted.

num   specifies that the lines from the current line for the value of *num* are
              sorted.

A   specifies that the lines are sorted in ascending order. (This is the
              default.)

D   specifies that the lines are sorted in descending order.

fromcol   specifies that the lines are sorted on data beginning in this column. If
              you do not specify *fromcol tocol*, sorting begins at the first column.

tocol   specifies that the lines are sorted on data ending in this column.

18.5.38.2 Return Codes
0   Normal return
202   Invalid operand
229   Number out of range

18.5.38.3 Example
```
'SORT * A 5 10'
```

This example sorts all lines in the file from the current line down and is sorted on
columns 5 to 10.

18.5.38.4 Note

If you sort a large number of lines, the sort will work very slowly.

18.5.39 SPLIT

```
►►──SPLIT────────────────────────────────────────────►◄
```

SPLIT splits a line into two lines.

18.5.39.1 Return Codes

0   Normal return
210   Request failed

18.5.39.2 Example

`'SPLIT'`

This example splits the line, that the cursor is on, into two lines. One line contains all the text on that line to the left of the cursor and the line following contains the remaining text (under and to the right of the cursor).

18.5.40 STRIP

```
►►──STRIP─────────────────────────────────────────────►◄
```

STRIP strips the trailing blanks off all file lines.

18.5.40.1 Return Codes

0   Normal return

18.5.40.2 Example

`'STRIP'`

This example strips all trailing blanks of each file line.

18.5.41 SYNONYM

```
►►──SYNONYM──syn──┬───────────┬──►◄
                  └─command─┘
```

SYNONYM assigns a command action to any other valid command.

18.5.41.1 Operands

syn   specifies any valid command that executes the command action for which it is a synonym.

command specifies any valid command.

18.5.41.2 Return Codes
0   Normal return

18.5.41.3 Example
`'SYNONYM GL GETLIB'`

This example makes GL equivalent to the command GETLIB.

18.5.42 TOP

```
►►──TOP──────────────────────────────────────────────────►◄
```

TOP scrolls to the top of the file.

18.5.42.1 Return Codes
0   Normal return

18.5.42.2 Example
`'TOP'`

This example scrolls to the top of the file.

18.5.43 TRUNC

```
►►──TRUNC──column──────────────────────────────────────────►◄
```

TRUNC truncates each line in the file to the given length.

18.5.43.1 Operands
column   specifies the last column you want to keep.

18.5.43.2 Return Codes
0   Normal return
202 Invalid operand

18.5.43.3 Example
`'TRUNC 72'`

This example truncates all lines in the file to a length of 72 characters.

18.5.43.4 Note
This command is useful when you are working with datasets that have sequence
numbers that require removing. The editor does not currently have support for
placing or maintaining sequence numbers in a file.

18.5.44 UP

```
                    ┌─1──────┐
►►──UP──┤        ├────────────────────────►◄
                    └─number─┘
```
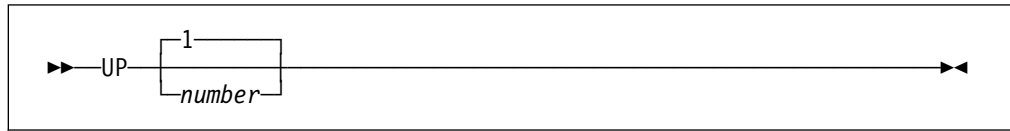
UP scrolls backward in the file (towards the top of the file).

18.5.44.1 Operands

number   specifies the number of lines to scroll. If you do not specify *number*, the
default scroll amount is set to 1.

18.5.44.2 Return Codes

0   Normal return
202   Invalid operand

18.5.44.3 Example

`'UP 20'`

This example scrolls 20 lines backward in the file.

Chapter 19. REXX/CICS File System

The REXX File System (RFS) is provided for the storage of text files and execs created with the REXX/CICS editor, and by execs using RFS commands and data imported from outside of the REXX File System. RFS was modeled after the Advanced Interactive Executive (AIX) and OS/2 file systems. The concept of directories is the main idea taken from these environments. Partitioning each directory into several subdirectories will give a hierarchical organization.

You can access RFS functions by using the RFS command. In addition to providing the ability to perform file I/O, the RFS command provides you the ability to do essential file system maintenance.

A file list utility (FLST) is provided as a full screen interface to the REXX File System. This utility can also be used as a platform where other CICS work can be performed.

19.1 File Pools, Directories, and Files

File pools are sets of VSAM files. The first VSAM file in the pool contains information about the pool and must be formatted before data can be written to the pool. All other VSAM files in the pool are extensions of the first file and only need to be defined in the pool's list of VSAM files after they have been allocated and defined to Access Method Services and CICS. VSAM files can be added to a file pool, at will, to provide additional storage space in that pool. However, defining and adding space to file pools is a task for REXX/CICS administrators or CICS Systems Programmers. File pools are given a unique one to seven character name.

In each file pool, there is a root directory. The root directory for a file pool is named *file_pool_name:\.* This directory can contain files as well as subdirectories. A subdirectory is a directory within another directory. Subdirectories can be created within any other directory. A new directory can be created with the RFS MKDIR command. All directories, except the root, are distinguished by a one to eight character directory file name joined, by a period, with an optional one to eight character directory file type. This is called the directory ID.

A file pool may be defined to be a user or non-user file pool. One directory that should exist in all user file pools is the USERS directory. This directory contains several subdirectories that correspond to the users on the system. When you save a file into the RFS for the first time, a new subdirectory is created in the USERS directory. This new directory is named with your user ID if you are signed onto CICS. Otherwise, the directory name will default to the CICS DFLTUSER value. After this directory is created, you can create any number of subdirectories within that personal directory. You can place files in any of the directories that you create.

Files are either data files or REXX execs. They use the same naming conventions as directories (a file name and an optional file type joined with a period). The default file type for REXX execs is EXEC. Files are created with the REXX/CICS editor or with the RFS DISKW command.

A fully qualified file ID consists of a file pool name followed by a :\, each directory's ID in the path followed by a \, and the file ID, that is comprised of a file name and an optional file type.

The following example shows a fully qualified file ID. POOL1 is the file pool name, USERS and USER1 are directory ID's, and TEST.EXEC is the file ID.

Example:

```
POOL1:\USERS\USER1\TEST.EXEC
```

The following example shows file pools, directories, and files.

Example:

```
POOL1:                                 File Pool
     \                                 Root Directory
       TEST1.EXEC                      File
       USERS\                          Subdirectory
            USER1\                     Subdirectory
                 TEST2.EXEC            File
                 DOCS\                 Subdirectory
                      TEST3.DOCUMENT   File
            USER2\                     Subdirectory
                 LETTER.DOCUMENT       File
       PROJECT1\                       Subdirectory
              PROD1.EXEC               File
              DATA\                    Subdirectory
                  PROD1.DATA           File

WORK:\                                 File Pool and Root Directory
     TEST1.DATA                        File
     CHARTS\                           Subdirectory
            CHART1.DATA                File
            CHART2.DATA                File
```

This example shows two file pools (POOL1 and WORK). File pool POOL1 contains a file (TEST1.EXEC) and two subdirectories (USERS and PROJECT1). Inside the USERS subdirectory are two subdirectories (USER1 and USER2) that correspond to user IDs (USER1 and USER2). User USER1 has a file (TEST2.EXEC) and a subdirectory (DOCS) inside its directory. Inside the DOCS subdirectory there is another file (TEST3.DOCUMENT). User USER2 has a file (LETTER.DOCUMENT) inside its directory. File pool WORK contains a file (TEST1.DATA) and a subdirectory (CHARTS). Inside subdirectory CHARTS are two files (CHART1.DATA and CHART2.DATA).

19.2 Current Directory and Path

The current directory is the current working directory, and is first in the search order when working with REXX File System (RFS). The current directory can be set using the CD command, see section 25.2, "CD" on page 371. The CD command has a similar format to the IBM Personal Computer OS/2 and IBM Personal Computer DOS CD commands. The syntax is `CD` followed by the partially or fully qualified directory name. To change from a subdirectory back to the parent directory, type `CD ...` To change to another subdirectory, CD can be followed by the subdirectory name.

In the following example, the first command sets the current directory to POOL1:\USERS\USER1 and the second command sets the current directory to POOL1:\USERS\USER1\DOCS. The third command changes the current directory back to POOL1:\USERS\USER1.

Example:

```
'CD POOL1:\USERS\USER1'
'CD DOCS'
'CD ..'
```

The PATH command is used to define the search order for REXX execs, after the current directory is searched. See the PATH command, section 25.31, "PATH" on page 413, for more information. The syntax is: PATH, followed by a list, separated by spaces, of fully qualified directory names.

The following example first sets a current directory, then defines the search order.

Example:

```
'CD POOL1:\USERS\USER1\EXECS'
'PATH POOL1:\ POOL1:\USERS\USER1'
'EXEC TEST2.EXEC'
```

The exec name is fully qualified, using the directory ID of each directory in the search before the search of each respective directory is performed. The fully qualified names are as follows:

```
'POOL1:\USERS\USER1\EXECS\TEST2.EXEC'
'POOL1:\TEST2.EXEC'
'POOL1:\USERS\USER1\TEST2.EXEC'
```

When the REXX/CICS command EXEC is invoked, all three directories above are searched resulting in REXX/CICS finding the exec in the POOL1:\USERS\USER1 directory. If TEST2.EXEC existed in the POOL1:\ directory, RFS would have stopped searching when it was found. The first copy found in the search order is accessed.

Note: Whenever a file name is not fully qualified, RFS follows the search order looking for the exec, beginning with the current directory. The first copy found is executed. If none are found, then an error is returned indicating the target file or exec was not found.

19.3 Security

There are two general types of REXX/CICS File System security:

- **File access security** controls access to execs and data. RFS file security can be controlled at two levels; the CICS level, and at the RFS directory level.

    1. At the CICS level, authorization to access file pool VSAM files can be given to specific users. This gives a high level of security.

    2. At the RFS directory level, user directories are private directories and can be accessed only by the owning user (by default).

    However, the owner of a directory can use the RFS AUTH command to define a directory as being publicr, publicw, or secured. **Publicr** means any other REXX/CICS user has read/only access to this directory. **Publicw** means any

other REXX/CICS user has read/write access to this directory. **Secured** means that the RFS security exit will be invoked to determine if access should be allowed. For more information, see the RFS AUTH command, section 19.4.1, "AUTH" on page 302. Non-user directories can be created and their access levels defined by an authorized user.

- **Command execution security** controls the use of certain REXX/CICS command, or command keywords. For more information, on this type of REXX/CICS security, see Appendix H, "Security" on page 481.

19.4 RFS commands

Under the RFS command environment you issue commands to interface with RFS. If you set the command environment to RFS, you should not specify RFS in front of RFS commands.

Example:

```
'RFS DISKR POOL1:\USERS\USER1\TEST.EXEC DATA.'
```

This example reads the contents of the RFS file TEST.EXEC into the REXX compound variable DATA. TEST.EXEC is in the fully qualified directory: POOL1:\USERS\USER1\.

The syntax for the RFS commands follow.

19.4.1 AUTH

```
                               ┌─PRIVATE──┐
►►──RFS──AUTH──dirid───────────┼──────────┼───────────────────────►◄
                               ├─PUBLICR──┤
                               ├─PUBLICW──┤
                               └─SECURED──┘
```
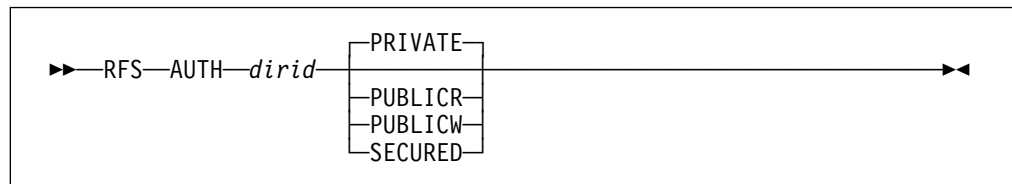
AUTH authorizes access to RFS directories.

19.4.1.1 Operands

dirid   specifies a REXX File System directory identifier. This is partially or fully qualified. See the CD command, 25.2, "CD" on page 371, for more information.

PRIVATE   specifies that only the owner of the directory has read/write access to the files. This is the default.

PUBLICR   specifies that any user has read-only access to the files in the directory.

PUBLICW

specifies that any user has read/write access to the files in the directory.

SECURED

specifies that an external security manager grants access to the files in the directory.

19.4.1.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.1.3 Example

`'RFS AUTH POOL1:\USERS\USER1\DOCS PUBLICR'`

This example makes directory DOCS a public directory. All users have read/only access to the files in directory DOCS.

19.4.2 CKDIR

```
►►──RFS──CKDIR──dirid───────────────────────────────►◄
```

CKDIR checks for an existing RFS directory level.

19.4.2.1 Operands

dirid   specifies a REXX File System directory identifier. This is partially or fully qualified. See the CD command, section 25.2, "CD" on page 371, for more information.

19.4.2.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.2.3 Example

`'RFS CKDIR POOL1:\USERS\USER1\DOCS'`

This example checks for a directory called DOCS in the existing directory POOL1:\USERS\USER1.

19.4.3 CKFILE

```
►►──RFS──CKFILE──fileid──────────────────────────────►◄
```

CKFILE checks to see if the specified, partially or fully qualified, file ID exists.

19.4.3.1 Operands

fileid   specifies the file identifier.

19.4.3.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.3.3 Example

`'CKFILE POOL1:\USERS\USER1\TEST.EXEC'`

This example checks for a file called TEST.EXEC in the existing directory POOL1:\USERS\USER1.

19.4.4 COPY

```
►►──RFS──COPY──fileid1──fileid2──────────────────────────────────►◄
```

COPY copies a file.

19.4.4.1 Operands

fileid1   specifies the source file identifier, it may be a fully or partially qualified directory and file identifier.

fileid2   specifies the target file identifier, it may be a fully or partially qualified directory and file identifier.

> **Note:** If the target file (*fileid2*) already exists, the contents of *fileid1* replaces it.

19.4.4.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.4.3 Example

```
'RFS COPY POOL1:\USERS\USER1\TEST1.EXEC POOL1:\USERS\USER1\TEST2.EXEC'
```

This example copies TEST1.EXEC to TEST2.EXEC within the directory POOL1:\USERS\USER1.

19.4.5 DELETE

```
►►──RFS──DELETE──fileid──────────────────────────────────────────►◄
```

DELETE deletes an RFS file.

19.4.5.1 Operands

fileid   specifies the name of the file to be deleted. This may be partially or fully qualified.

19.4.5.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.5.3 Example

```
'RFS DELETE POOL1:\USERS\USER1\TEST1.EXEC'
```

This example deletes file TEST1.EXEC within directory POOL1:\USERS\USER1.

19.4.6 DISKR

```
                                  ┌─DATA.─┐
►►──RFS──DISKR──fileid──┬───────┬─────────────────────────────────►◄
                        └─stem.─┘
```

DISKR reads records from an RFS file.

19.4.6.1 Operands

fileid   specifies the file identifier.

stem.   specifies the name of a stem. (A stem must end in a period.) See section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA..

19.4.6.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.6.3 Example

```
'RFS DISKR POOL1:\USERS\USER1\TEST.DATA DATA.'
```

This example stores the entire contents of the RFS file POOL1:\USERS\USER1\TEST.DATA in the DATA. REXX compound variable.

19.4.6.4 Note

DATA.0 is set to the number of records read from the file. DATA.*n* contains the *n*th record read from the file.

19.4.7 DISKW

```
►►──RFS──DISKW──fileid──┬──DATA.──┬──────────►◄
                        └──stem.──┘
```
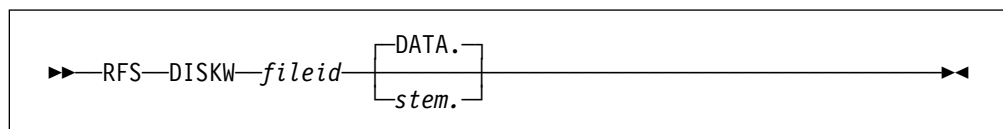
DISKW writes records to an RFS file from a stem. The file is overlaid with the data in the stem.

19.4.7.1 Operands

fileid   specifies the file identifier.

stem.   specifies the name of a stem. (A stem must end in a period.) See section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA..

19.4.7.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.7.3 Example

```
'RFS DISKW POOL1:\USERS\USER1\TEST.EXEC DATA.'
```

This example stores the contents of the DATA. REXX compound variable into the RFS file POOL1:\USERS\USER1\TEST.EXEC.

19.4.7.4 Note

Set DATA.0 to the number of records to be written to the file.

19.4.8 GETDIR

```
►►──RFS──GETDIR──stem.──────────────────────────────────►◄
                        └─dirid─┘
```

GETDIR returns a list of the contents of the current or specified directory into the specified REXX array.

19.4.8.1 Operands

stem.   specifies the name of a stem. (A stem must end in a period.) See section 12.4.4, "Stems" on page 145 for more information.

dirid   specifies a REXX File System directory level identifier. This is partially or fully qualified. See the CD command, section 25.2, "CD" on page 371, for more information.

19.4.8.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.8.3 Example

```
'RFS GETDIR DIRDOC. POOL1:\USERS\USER1\DOCS'
```

This example places the contents of directory DOCS in the DIRDOC. REXX compound variable.

19.4.9 MKDIR

```
►►──RFS──MKDIR──dirid────────────────────────────────────►◄
```

MKDIR creates a new RFS directory level.

19.4.9.1 Operands

dirid   specifies a REXX File System directory identifier. This is partially or fully qualified. See the CD command, section 25.2, "CD" on page 371, for more information.

19.4.9.2 Return Codes

See the RFS command, section 25.33, "RFS" on page 416.

19.4.9.3 Example

```
'RFS MKDIR POOL1:\USERS\USER1\DOCS'
```

This example creates a new directory called DOCS in the existing directory POOL1:\USERS\USER1.

19.4.9.4 Note

Only authorized users can create directories outside of their \USERS*userid* directory structure.

19.4.10 RDIR

```
►►──RFS──RDIR──dirid──────────────────────────────────────────────►◄
```

RDIR removes the specified RFS directory.

19.4.10.1 Operands
dirid   specifies a REXX File System directory identifier. This is partially or fully qualified. Refer to the CD command, section 25.2, "CD" on page 371, for more information.

19.4.10.2 Return Codes
Refer to the RFS command, section 25.33, "RFS" on page 416.

19.4.10.3 Example
```
'RFS RDIR POOL1:\USERS\USER1\DOCS'
```

This example deletes a directory called DOCS in the existing directory POOL1:\USERS\USER1.

19.4.11 RENAME

```
►►──RFS──RENAME──fileid1──fileid2─────────────────────────────────►◄
```

RENAME renames an RFS file to a new name.

19.4.11.1 Operands
fileid1   specifies the source file identifier, it may be a fully or partially qualified directory and file identifier.

fileid2   specifies the source target file identifier, it may be a fully or partially qualified directory and file identifier.

> **Note:** If the target file (*fileid2*) already exists, the contents of *fileid1* replaces it.

19.4.11.2 Return Codes
See the RFS command, section 25.33, "RFS" on page 416.

19.4.11.3 Example
```
'RFS RENAME POOL1:\USERS\USER1\TEST1.EXEC POOL1:\USERS\USER1\TEST2.EXEC'
```

This example renames the file POOL1:\USERS\USER1\TEST1.EXEC to POOL1:\USERS\USER1\TEST2.EXEC.

19.5 File List Utility

The File List Utility (FLST) provides a full screen interface to the REXX File System. When running, FLST manages RFS, calls an exec, or starts a transaction. It is meant to be a high level interface to REXX, RFS, and CICS.

19.5.1 Invocation

When you want to run FLST, go to a cleared CICS screen or a cleared REXX/CICS screen, enter `FLST` and FLST starts running. The FLST screen format follows.

```
USER=USER1 - DIRECTORY=\USERS\USER1
CMD    FILENAME FILETYPE ATTRIBUTES RECORDS SIZE   DATE       TIME
       TEST1    EXEC     FILE       11      1      1994/03/27 10:30:29
       TEST2    EXEC     FILE       5       1      1994/03/27 10:31:04














 COMMAND ===>
F1=HELP F2=REFRESH F3=END F7=UP 18 F8=DOWN 18 F11=EDIT F12=CANCEL
```

Your **user ID** is displayed in the upper left hand corner. The **current directory** is displayed beside your user ID. The rest of the screen looks very similar to a REXX/CICS editor session. FLST uses the editor for all I/O except for the first two lines that are displayed using the RESERVED command in the editor. The advantages of using the REXX/CICS editor for the I/O shell are seen in the ability to search for a file name or file type in a large directory and your ability to save the directory to a file on disk.

19.6 Macros under the REXX/CICS File List Utility

The REXX File List Utility supports REXX macros, giving the macros the ability to alter the FLST settings and display the FLST screens. Macros can process all of the FLST commands.

The following example addresses the FLST environment and alters the FLST settings.

Example:

```
/* Macro to set some FLST settings */
ADDRESS FLSTSVR
'SET PFKEY 11 EDIT'
'SET PFKEY 12 CANCEL'
'SYNONYM DISCARD RFS DELETE'
```

19.7 FLST Commands

This section describes the FLST commands. You can type these commands anywhere on the source FLST command column or from the command line.

Note: If data is entered in multiple places, program function keys take precedence, followed by data entered on the command line, and finally data entered on the command column.

19.7.1 CANCEL

When you type CANCEL from the command line use the following syntax:

```
►►──CANCEL──────────────────────────────────────────────►◄
```

CANCEL terminates without executing any commands in the command column.

19.7.2 CD

When you type CD from the command line use the following syntax:

```
►►──CD─┬──────────┬──────────────────────────────────────►◄
       └─dirid─┘
```

CD changes the current directory.

19.7.2.1 Operands
dirid   specifies a REXX File System directory level identifier. This is partially or fully qualified. See the CD command, section 25.2, "CD" on page 371, for more information.

19.7.2.2 Example
`'CD TEMP'`

This example changes the current directory to TEMP and updates the FLST display.

19.7.3 COPY

When you type COPY on the FLST command column use the following syntax:

```
►►──COPY──/──fileid──────────────────────────────────────►◄
```

COPY copies a file.

19.7.3.1 Operands

fileid　　　specifies the file ID of the file where the results are placed.

>　　　**Note:**　If *fileid* already exists, it is replaced.

19.7.3.2 Example

```
'COPY / TEST3.EXEC'
```

This example, executed from the command column next to TEST1.EXEC, creates a new file, TEST3.EXEC, that is identical to TEST1.EXEC.

When you type COPY from the command line use the following syntax:

```
►►──COPY──fileid1──fileid2─────────────────────────►◄
```

19.7.3.3 Operands

fileid1　　　specifies the file ID of the file the command acts on.

fileid2　　　specifies the file ID of the file where the results are placed.

>　　　**Note:**　If *fileid2* already exists, the contents of *fileid1* replaces it.

19.7.3.4 Example

```
'COPY TEST1.EXEC TEST3.EXEC'
```

This example, executed from the command line, creates a new file (TEST3.EXEC) that is identical to TEST1.EXEC.

19.7.4 DELETE

When you type DELETE on the FLST command column use the following syntax:

```
►►──DELETE────────────────────────────────────────►◄
```

DELETE deletes a file.

When you type DELETE from the command line use the following syntax:

```
►►──DELETE──fileid─────────────────────────────────►◄
```

19.7.4.1 Operands

fileid　　　specifies the file ID of the file the command acts on.

19.7.4.2 Example

`'DELETE TEST1.EXEC'`

This example, executed from the command line, deletes file TEST1.EXEC.

19.7.5 DOWN

When you type DOWN from the command line use the following syntax:

```
►►──DOWN──────────────────────────────────────►◄
          └─n─┘
```
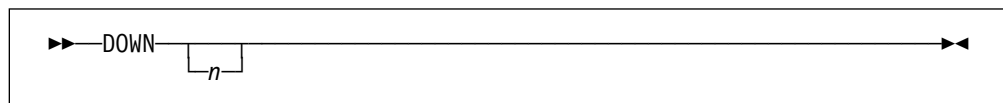
DOWN scrolls down one or more lines.

19.7.5.1 Operands

n   specifies the number of lines to be scrolled down.

19.7.5.2 Example

`'DOWN 5'`

This example scrolls forward through the list five lines.

19.7.6 END

When you type END from the command line use the following syntax:

```
►►──END───────────────────────────────────────►◄
```
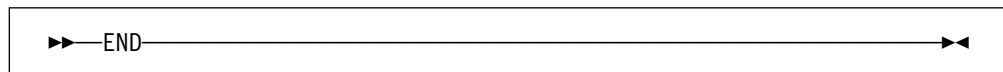
END executes all commands you typed, then terminates when END is typed on the command line or used as a PF key.

19.7.7 EXEC

When you type EXEC on the FLST command column use the following syntax:

```
►►──EXEC──────────────────────────────────────►◄
         └─/──parameter─┘
```
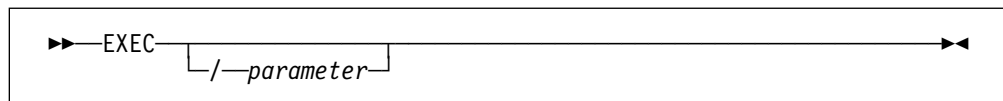
EXEC executes the exec, then terminates.

19.7.7.1 Operands

parameter
          specifies the parameters passed to the exec as arguments.

19.7.7.2 Example
```
'EXEC / PARMS'
```

This example, executed on the command column next to TEST3.EXEC, executes exec TEST3.EXEC and passes `PARMS` as the argument.

When you type EXEC from the command line use the following syntax:

```
►►──EXEC──fileid────────────────────────────────────────►◄
                 └─parameter─┘
```
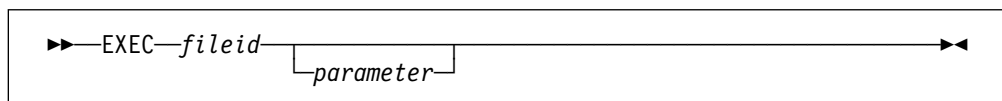
19.7.7.3 Operands
fileid   specifies the file ID of the file the command acts on.

parameter
        specifies the parameters passed to the exec as arguments.

19.7.7.4 Return Codes
n   specifies the return code set by the exit of the called exec
0   Normal return
-3   Exec not found
-10   Exec name not specified
-11   Invalid exec name
-12   GETMAIN error
-99   Internal error

19.7.7.5 Example
```
'EXEC TEST3 PARMS'
```

This example executes exec TEST3.EXEC and passes `PARMS` as the argument.

19.7.8 FLST

When you type FLST from the command line use the following syntax:

```
►►──FLST──────────────────────────────────────────────►◄
         └─dirid─┘
```
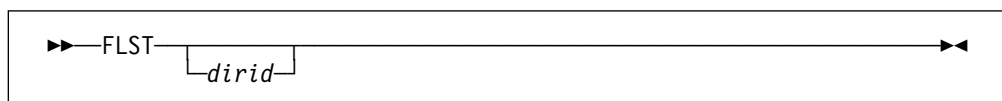
FLST calls the file list utility.

19.7.8.1 Operands
dirid   specifies the ID of a full or partial directory. If you do not specify *dirid*, FLST defaults to the current working directory.

19.7.8.2 Example
```
'FLST'
```

This example displays the file list for the member of the current working directory.

19.7.8.3 Note

See Chapter 19, "REXX/CICS File System" on page 299 for more information about the REXX File System.

19.7.9 MACRO

When you type MACRO from the command line use the following syntax:

```
►►──MACRO──fileid──────────────────────────────────────────────►◄
```

MACRO calls a macro.

19.7.9.1 Operands

fileid   specifies the file ID of the macro you want to run. If this file ID includes a file type suffix, then an attempt is made to call an exec with that suffix. Otherwise, an attempt is made to call an exec whose suffix is EXEC.

19.7.9.2 Return Codes

n   specifies the return code set by the exit of the called exec
0   Normal return
-3   Exec not found
-10   Exec name not specified
-11   Invalid exec name
-12   GETMAIN error
-99   Internal error

19.7.9.3 Example

```
'MACRO POOL1:\USERS\USER1\TEST'
```

This example calls the macro, POOL1:\USERS\USER1\TEST.EXEC.

19.7.9.4 Note

Macros have the ability to make calls to the REXX/CICS FLST server. Any command that can be entered from the command line of the FLST can be run from a macro.

19.7.10 PFKEY

When you type PFKEY from the command line use the following syntax:

```
►►──PFkey──number─────────────────────────────────────────────►◄
                  └─text─┘
```

PFKEY sets or processes a program function (PF) key.

19.7.10.1 Operands

number   specifies the PF key that is set or processed.

text   specifies the text that the PF key is set to.

19.7.10.2 Example
```
'PFKEY 3 quit'
'PFKEY 3'
```

This example first sets PFKEY 3 to `quit` and then processes the PF key.

19.7.10.3 Note
If you specify *text*, the PF key is set with the text. If you do not specify *text*, the PF key is processed.

19.7.11 REFRESH

When you type REFRESH on the FLST command column use the following syntax:

```
►►──REFRESH──────────────────────────────────────────►◄
```

REFRESH refreshes the file list.

19.7.11.1 Example
```
'REFRESH'
```

This example refreshes the file list for the member of the current working directory.

19.7.12 RENAME

When you type RENAME on the FLST command column use the following syntax:

```
►►──RENAME──/──fileid────────────────────────────────►◄
```

RENAME renames a file.

19.7.12.1 Operands
fileid   specifies the new file ID.

> **Note:** If *fileid* already exists, it is replaced.

19.7.12.2 Example
```
'RENAME / TEST4.EXEC'
```

This example, executed from the command column next to TEST3.EXEC, renames TEST3.EXEC to TEST4.EXEC.

When you type RENAME from the command line use the following syntax:

```
►►──RENAME──fileid1──fileid2─────────────────────────►◄
```

19.7.12.3 Operands

fileid1   specifies the file ID of the file the command acts on.

fileid2   specifies the new file ID.

> **Note:**   If *fileid2* already exists, the contents of *fileid1* replaces it.

19.7.12.4 Example

`'RENAME TEST3.EXEC TEST4.EXEC'`

This example, executed from the command line, renames file TEST3.EXEC to TEST4.EXEC.

19.7.13 SORT

When you type SORT from the command line use the following syntax:

```
►►──SORT──┬─DT─┬──────────────────────────────►◄
          ├─FN─┤
          ├─FT─┤
          ├─AT─┤
          ├─RC─┤
          └─SZ─┘
```
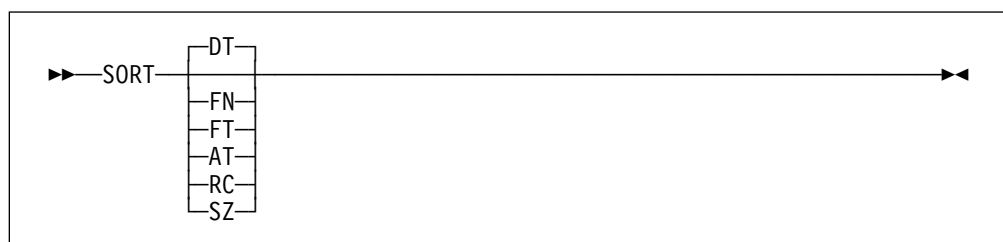
SORT sorts the file list.

19.7.13.1 Operands

DT   specifies sorting the files by date/time. (This is the default.)

FN   specifies sorting the files by file name.

FT   specifies sorting the files by file type.

AT   specifies sorting the files by attribute.

RC   specifies sorting the file by number of records.

SZ   specifies sorting the files by size.

19.7.13.2 Example

`'SORT FN'`

This example sorts the file list by file name.

19.7.14 SYNONYM

When you type SYNONYM from the command line use the following syntax:

```
►►──SYNONYM──syn──┬──────────┬──────────────────►◄
                  └─command──┘
```
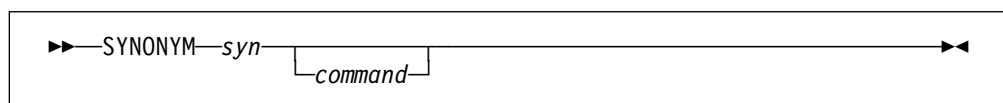
SYNONYM assigns a command action to any other valid command.

19.7.14.1 Operands

syn   specifies any valid command that executes the command action for which it is a synonym.

command specifies any valid command.

19.7.14.2 Example

```
'SYNONYM DISCARD RFS DELETE'
```

This example makes DISCARD equivalent to the RFS command DELETE.

19.7.15 UP

When you type UP from the command line use the following syntax:

```
►►──UP────────────────────────────────────────────►◄
       └─n─┘
```
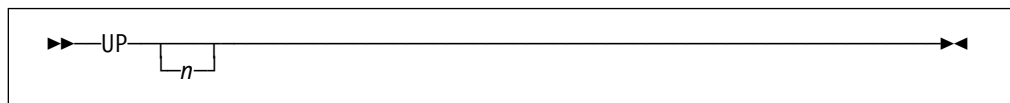
UP scrolls up one or more lines.

19.7.15.1 Operands

n   specifies the number of lines to be scrolled up.

19.7.15.2 Example

```
'UP 5'
```

This example scrolls backward through the list five lines.

19.8 FLST Return Codes

FLST uses the standard return codes for the REXX/CICS File System, except where it is specified differently. See the RFS command, section 25.33, "RFS" on page 416 for more information.

19.9 Running Execs and Transactions from FLST

Most REXX/CICS execs can be run from FLST by simply entering the exec name. The exec name with its arguments is entered on the command line, or the exec name is typed on a FLST command column. The latter case uses the file ID as the argument for the transaction. REXX execs are run by issuing an EXECUTE command either on the command line, in which case a file ID is needed, or on a FLST command column.

Chapter 20. REXX/CICS List System

REXX/CICS provides a facility for maintaining tables or lists of data in virtual storage. This facility is called the REXX List System (RLS). This system provides management of lists of temporary system and user information. The externals for accessing the RLS are the RLS and CLD commands, instead of RFS and CD commands used with the REXX File System. Also, the RLS is for data only, not for execs.

20.1 Directories and Lists

RLS has one root directory. You access the RLS system by reading its anchor address from a CICS temporary storage queue named *CICREX*. This queue contains one item after the first access of RLS. This item contains the address of an area of 6 fullwords. It contains RLS control information and a pointer to the root directory. The root directory RLS is named \. This directory can contain lists, saved variables, or special lists called queues as well as subdirectories. A subdirectory is a directory within another directory. Subdirectories can be created within any other directory. A new directory can be created with the RLS MKDIR command. All directories, except the root, are distinguished by a one to 250 character directory file name. This is called the directory ID.

One RLS directory that always exists is the USERS directory. This directory contains several subdirectories which correspond to the users on the system. When you save a list into the RLS for the first time, a new subdirectory may be created in the USERS directory. This new directory is named with your user ID if you are signed onto CICS. Otherwise, the directory name defaults to the value in CICS DFLTUSER. After this directory is created, you can create any number of subdirectories within that personal directory. You can place lists in any of the directories that you create.

Lists are always data files. They use the same naming conventions as directories.

A fully qualified list ID consists of a \, each directory's ID in the path followed by a \, and the list ID.

The following example shows a fully qualified list ID. USERS and USER1 are directory ID's, and TEST.DATA is the list ID.

Example:

```
\USERS\USER1\TEST.DATA
```

The following example shows RLS directories and lists.

Example:

```
\                                      Root Directory
  TEST1.DATA                           File
  USERS\                               Subdirectory
        USER1\                         Subdirectory
              TEST2.DATA               File
              DOCS\                    Subdirectory
                    TEST3.DOCUMENT     File
        USER2\                         Subdirectory
              LETTER.DOCUMENT          File
  PROJECT1\                            Subdirectory
          PROD1.INFO                   File
          DATA\                        Subdirectory
                PROD1.DATA             File

 \                                     Root Directory
  TEST1.DATA                           File
  CHARTS\                              Subdirectory
         CHART1.DATA                   File
         CHART2.DATA                   File
```

This example shows a list directory structure. The root directory contains a file (TEST1.DATA) and two subdirectories (USERS and PROJECT1). Inside the USERS subdirectory are two subdirectories (USER1 and USER2) that correspond to user IDs (USER1 and USER2). User USER1 has a list (TEST2.DATA) and a subdirectory (DOCS) inside its directory. Inside the DOCS subdirectory there is another list (TEST3.DOCUMENT). User USER2 has a file (LETTER.DOCUMENT) inside its directory. The root directory contains a file (TEST1.DATA) and a subdirectory (CHARTS). Inside subdirectory CHARTS are two files (CHART1.DATA and CHART2.DATA).

20.2 Current Directory and Path

The current list directory is the current working directory, and is first in the search order when working with REXX List System (RLS). The current list directory can be set using the CLD command, see section 25.5, "CLD" on page 375. The syntax is CLD followed by the fully or partially qualified directory name. To change from a subdirectory back to the parent directory, type CLD .. To change to another subdirectory, CLD can be followed by the subdirectory name.

In the following example, the first command sets the current directory to \USERS\USER1 and the second command sets the current directory to \USERS\USER1\DOCS. The third command changes the current directory back to \USERS\USER1.

Example:

```
CLD \USERS\USER1
CLD DOCS
CLD ..
```

Note: If CLD is never specified the default directory is \SYSTEM*nnnnnnnn*\. where *nnnnnnnn* is an internal REXX task number.

20.3 Security

The RLS commands are authorized REXX/CICS commands. This means they may only be executed by an authorized user or from within an exec loaded from an authorized library.

20.4 RLS commands

Under the RLS command environment you issue commands to interface with RLS. If you set the command environment to RLS, you should not specify RLS in front of RLS commands.

Example:

```
'RLS READ \USERS\USER1\TEST.DATA DATA.'
```

This example reads the contents of the RLS list \USERS\USER1\TEST.DATA into the DATA. REXX compound variable.

The syntax for the RLS commands follow.

20.4.1 CKDIR

```
►►──RLS──CKDIR──dirid────────────────────────────►◄
```

CKDIR checks for an existing RLS directory level.

20.4.1.1 Operands

dirid   specifies a REXX List System directory level identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.1.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.1.3 Example

```
'RLS CKDIR \USERS\USER1\DOCS'
```

This example checks for a directory called DOCS in the existing directory \USERS\USER1.

20.4.2 DELETE

```
►►──RLS──DELETE──listname────────────────────────►◄
```

DELETE deletes an RLS list.

20.4.2.1 Operands

listname specifies a REXX List System list identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.2.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.2.3 Example

```
'RLS DELETE \USERS\USER1\TEST.DATA'
```

This example deletes RLS list TEST.DATA.

20.4.3 LPULL

```
►►──RLS──LPULL──varname──┬──*QUEUE*──┬──────────────────────►◄
                         └──queid────┘
```
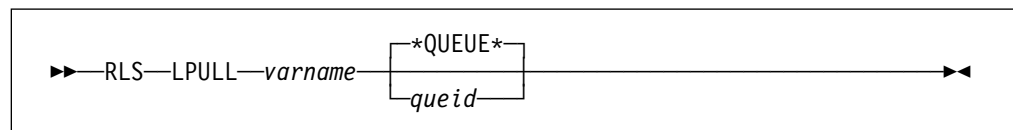
LPULL pulls a record from the top of the RLS queue.

20.4.3.1 Operands

varname specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

QUEUE is a keyword specifying the special default name.

queid specifies the identifier for a special type of RLS list accessed by LPULL, LPUSH, or LQUEUE.

20.4.3.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.3.3 Example

```
'RLS LPULL VARA QUEUE1'
```

This example pulls a record from the top of the RLS queue QUEUE1.

20.4.4 LPUSH

```
►►──RLS──LPUSH──varname──┬──*QUEUE*──┬──────────────────────►◄
                         └──queid────┘
```
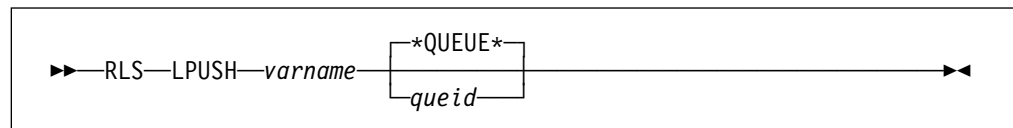
LPUSH pushes a record onto the top of the RLS queue (LIFO).

20.4.4.1 Operands

varname   specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

QUEUE   is a keyword specifying the special default name.

queid   specifies the identifier for a special type of RLS list accessed by LPULL, LPUSH, or LQUEUE.

20.4.4.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.4.3 Example

```
'RLS LPUSH VARA QUEUE1'
```

This example pushes a record (the contents of VARA) onto the top of the RLS queue QUEUE1.

20.4.5 LQUEUE

```
►►──RLS──LQUEUE──varname──┬──*QUEUE*──┬──────────────►◄
                          └──queid────┘
```
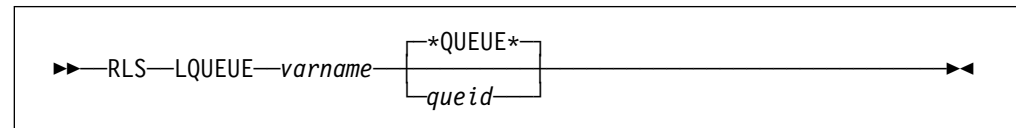
LQUEUE adds a record to the end of the RLS queue (FIFO).

20.4.5.1 Operands

varname   specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

QUEUE   is a keyword specifying the special default name.

queid   specifies the identifier for a special type of RLS list accessed by LPULL, LPUSH, or LQUEUE.

20.4.5.2 Return Codes

See to the RLS command, section 25.34, "RLS" on page 418.

20.4.5.3 Example

```
'RLS LQUEUE VARA QUEUE1'
```

This example adds a record (the contents of VARA) to the end of the RLS queue QUEUE1.

20.4.6 MKDIR

```
►►──RLS──MKDIR──dirid──────────────────────────────►◄
```
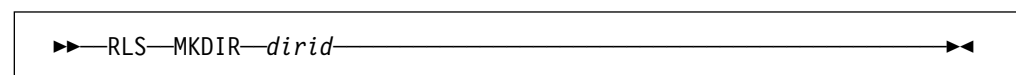
MKDIR creates a new RLS directory level.

20.4.6.1 Operands

dirid   specifies a REXX List System directory level identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.6.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.6.3 Example

```
'RLS MKDIR \USERS\USER1\DOCS'
```

This example creates a new directory called DOCS in the existing directory \USERS\USER1.

20.4.7 READ

```
►►──RLS──READ──listname──┬──┬─DATA.─┬──┬──┬──────────┬──►◄
                         │  └─stem.─┘  │  └─(──UPD─┘
```
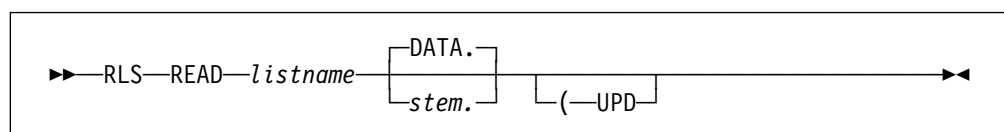
READ reads records from an RLS list.

20.4.7.1 Operands

listname   specifies the list identifier.

stem.   specifies the name of a stem. (A stem must end in a period.) See section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA..

UPD   is a keyword that enqueues on a file for update.

20.4.7.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.7.3 Example

```
'RLS READ \USERS\USER1\TEST.DATA DATA.'
```

This example stores the entire contents of the RLS list \USERS\USER1\TEST.DATA in the DATA. REXX compound variable.

20.4.7.4 Note

DATA.0 is set to the number of records read from the list. DATA.*n* contains the *n*th record read from the list.

20.4.8 VARDROP

```
►►──RLS──VARDROP──varname──dirid──────────────────────────►◄
```
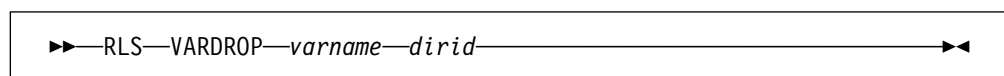
VARDROP deletes an RLS saved variable.

20.4.8.1 Operands

varname specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

dirid specifies a REXX List System directory level identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.8.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.8.3 Example

```
'RLS VARDROP VAR1'
```

This example deletes variable VAR1 from the current directory.

20.4.9 VARGET

```
►►──RLS──VARGET──varname──dirid──────────────────────────►◄
```

VARGET takes an RLS saved variable and copies it into a REXX variable of the same name.

20.4.9.1 Operands

varname specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

dirid specifies a REXX List System directory level identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.9.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.9.3 Example

```
'RLS VARGET VAR1'
```

This example copies the value of variable VAR1 from the current directory into a REXX variable named VAR1.

20.4.10 VARPUT

```
►►──RLS──VARPUT──varname──dirid──────────────────────────►◄
```

VARPUT takes a REXX variable and copies it into an RLS saved variable of the same name.

20.4.10.1 Operands

varname   specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

dirid   specifies a REXX List System directory level identifier. This is partially or fully qualified. See the CLD command, section 25.5, "CLD" on page 375, for more information.

20.4.10.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.10.3 Example

```
'RLS VARPUT VAR1'
```

This example takes the value of REXX variable VAR1 and copies it into variable VAR1 in the current directory.

20.4.11 WRITE

```
►►──RLS──WRITE──listname──┬──DATA.──┬──────────────────────►◄
                          └──stem.──┘
```
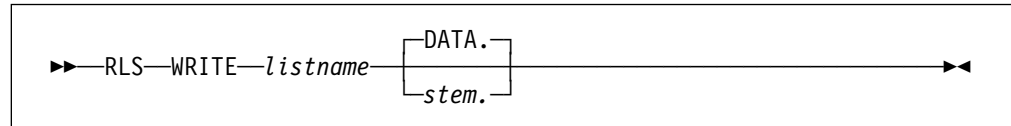
WRITE writes records to an RLS list.

20.4.11.1 Operands

listname   specifies the list identifier.

stem.   specifies the name of a stem. (A stem must end in a period.) See section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA..

20.4.11.2 Return Codes

See the RLS command, section 25.34, "RLS" on page 418.

20.4.11.3 Example

```
'RLS WRITE \USERS\USER1\TEST.DATA DATA.'
```

This example stores the entire contents of the REXX compound variable DATA. into the RLS list \USERS\USER1\TEST.DATA.

20.4.11.4 Note

Set DATA.0 to the number of records to be written to the list.

Chapter 21. REXX/CICS Command Definition

The REXX/CICS Command Definition Facility provides a means of easily defining (or redefining) REXX commands and environments.

21.1 Background

One of the greatest strengths of REXX is its extendibility. You can write your own external functions, subroutines, or commands to extend the capabilities of the REXX language. Because of this, one of the natural uses for REXX is as an Application Integration platform. REXX/CICS provides you the ability to seamlessly integrate your Line Of Business (LOB) application facilities, REXX language facilities, CICS system facilities, and various software products into an integrated software platform.

The two main methods used to extend the REXX language, in order to add function, or to provide an interface to an external facility or product, are:

- REXX External Functions (or subroutines)
- REXX Commands (sometimes called subcommands).

As an example to distinguish between the two, let's say we want to add the capability to REXX to sort a REXX array. This could be accomplished, for example, by adding a sort function called ARRYSORT:

```
x = ARRYSORT(STEM1.)
```

This could likewise be accomplished by adding a REXX command called ARRYSORT:

```
'ARRYSORT STEM1.'
```

Note: In the above command example, it is not necessary to place quotes around the command. It is however good coding practice to place single or double quotes around portions of command strings that do not involve REXX variable substitution.

REXX external functions are traditionally used to extend the capabilities of the REXX language, whereas REXX commands are traditionally used to interface to outside applications, products or system facilities. However, either method is capable of being used either way.

Some major differences between REXX commands and external functions are:

- Functions always return a result, commands do not.
- Functions raise an error condition, commands always set a return code.

21.2 Highlights

The Command Definition Facility provides the capability to:

- Define new commands and environments easily, from a REXX exec.
- Share a common command environment with multiple independent developers.
- Write new REXX commands in the REXX language.
- Change command's implementation language transparently.
- Define authorized commands selectively.

325

> • Redefine existing command names (without a code change).

21.3 Accomplishing Command Definition

You should use the DEFCMD command to perform basic command definitions that only affect your user ID. A systems administrator or system programmer should use the DEFSCMD to perform system-wide REXX/CICS command definitions. The DEFCMD and DEFSCMD commands are used from within a REXX exec to define or change REXX command definitions. You can add or change your own command definitions, using the DEFCMD command, without any special authorization. You must be a REXX/CICS authorized user to use DEFSCMD to change command definitions that affect other REXX/CICS users. See section 25.10, "DEFCMD" on page 383 for more information on the DEFCMD command and section 25.11, "DEFSCMD" on page 386 for more information on the DEFSCMD command.

21.4 Command Arguments Passed to REXX Programs

When a REXX/CICS command is written in REXX and that command is used, the REXX program (defined by DEFCMD or DEFSCMD) is either invoked or awakened (from a WAITREQ induced "sleep"). If it is invoked, then the command string is passed as an argument to the exec. Also, if it is invoked, the very first WAITREQ command issued (if any) falls through immediately, with the command string being placed in the REXX variable REQUEST. If the REXX exec was already started earlier and waiting for a request (due to an earlier WAITREQ command) then the command string is only placed in the REXX variable REQUEST.

Note: Command programs written in REXX can easily get and set the contents of REXX variables in the REXX exec that caused them to be invoked, by using the C2S and S2C commands. See section 25.9, "C2S" on page 382 for more information on the C2S command and section 25.38, "S2C" on page 425 for more information on the S2C command.

21.5 Command Arguments Passed to Assembler Programs

In addition to the REXX language, REXX/CICS command programs may be written in assembler language. Assembler language routines must exist in a CICS program properly defined (for example, by using the CEDA DEFINE PROGRAM command). These programs are invoked by an EXEC CICS LINK if the CICSLINK option was specified on the DEFCMD or DEFSCMD commands. If the DEFCMD or DEFSCMD, on the other hand, specifies the CICSLOAD option, then the program is EXEC CICS LOADed by the first command that causes it to be invoked for the current CICS task, and its load address is remembered. Any subsequent commands in the same CICS task that use this program performs a direct branch entry (by an assembler BASSM instruction) into the program. It is recommended that these assembler programs return control by an assembler BSM instruction so that the correct mode switching (if any) occurs.

The following describes the contents of the registers when an assembler language command program gets control, and it describes the parameters upon entry to these programs.

Entry Specifications when DEFCMD CICSLOAD is specified:

When the code for the command program gets control by a direct branch, the contents of the registers are:

Register 0   Unpredictable

Register 1   Address of the CICPARMS control block

Registers 2-12 Unpredictable

Register 13   Address of 18 fullword register save area

Register 14   Return address

Register 15   Entry point address

Before the program returns to the caller, it should place the return code it wants reflected into the CICPARMS RETCODE field.

Entry Specifications when DEFCMD CICSLINK is specified:

When the code for the command program gets control by an EXEC CICS LINK, the CICS Commarea contains the CICPARMS control block.

Before the program returns to the caller, it should place the return code it wants reflected into the CICPARMS RETCODE field.

21.6 CICPARMS Control Block

The following table shows the CICPARMS control block for mapping passed parameters to assembler routines.

Table 3 (Page 1 of 2). CICPARMS Control Block			
Offset (Decimal)	**Number of Bytes**	**Field Name**	**Description**
0	12		Reserved for IBM use.
12	4	RXWBADDR	REXX work block address which is required to be placed into register 10 before calls to the CICGETV stub routine (for REXX variable access)
16	8	ENVNAME	Internal environment name taken from the DEFCMD or DEFSCMD command definition
24	16	CICCMD	Internal command name taken from the command definition, or in the case where an asterisk was specified, the actual command name from the command string
40	4	ARGSTR	Address of the command argument string beginning with the first non-blank character after the command name in the command string
44	4	ARGLEN	Length of above argument string, in characters
48	4	PLIST	Address of a standard parsed parameter list of the command line parsed into 8 character tokens, followed by a end of list fence of hex high values (X'FFFFFFFFFFFFFFFF')

Table 3 (Page 2 of 2). CICPARMS Control Block			
Offset (Decimal)	**Number of Bytes**	**Field Name**	**Description**
52	4	EPLIST	Address of an extended parameter list which matches up with the standard PLIST above, but that is in a different format. The extended PLIST has an 8 byte entry for each token described above. The first 4 bytes is a fullword address of the start of the string that comprises a token. The second word contains the length of a token, in bytes.
56	4	RETCODE	Return code to be reflected in the exec immediately after the execution of the command. This return code is automatically placed into the special REXX variable RC.
60	4		Reserved for IBM use
64	4	USERWORD	For user use so information can be passed across multiple command routine calls
68	4		Reserved for IBM use
72	4		Reserved for IBM use
76	1	TYPEFLAG	One character code that identifies the call type of the DEFCMD or DEFSCMD definition. The code for REXX is R, for CICSLINK is C, and for CICSLOAD is L
77	1	ITRACE	Internal trace flag. This is a one character code, which has a value of 0 through 9 to indicate if internal tracing is active and what level of tracing is active. The value of zero indicates the normal situation of no tracing. Values from 1 to 9 indicate that increasingly progressively detailed tracing has been requested.

21.7 Non-REXX Language Interfaces

REXX/CICS makes it possible to transparently convert a REXX process to a non-REXX process. To do this requires that non-REXX command routines should be able to access REXX variables in the REXX exec that issued the command to be processed. The routine used to accomplish this is called CICGETV and must be linkedited with your command routine, and called as is described below.

21.8 CICGETV - Call to Get, Set, or Drop a REXX Variable

```
►►──CALL──CICGETV──,(─┤ operands ├─)────────────────────────────────────►◄
```

operands:
```
├──varname_addr──,varname_len──,data_addr──,data_len──,function_name──────────────┤
```

Calls this linkedit stub subroutine from an assembler REXX/CICS command routine, so as to retrieve from, or set or drop REXX variables in the REXX program that issued the command.

21.8.1.1 Operands

varname_addr

> specifies the address of a character string containing the name of the REXX variable. The variable name (that this address points to) must be in uppercase.

varname_len

> specifies the length of the variable name.

data_addr

> specifies the fullword address where the address of variable contents are.

data_len   specifies the length of area pointed to by *data_addr* (fullword).

function_name

> specifies the particular CICGETV function to be performed. There are three choices:

> **GET** retrieves the address and length of a REXX variable.
> **PUT** creates or replaces a REXX variable.
> **DEL** deletes a REXX variable.

21.8.1.2 Notes

1. Immediately before any call to the CICGETV linkedited routine stub, register 10 must be loaded with the value of the RXWBADDR field in the passed parms (CICPARMS).

2. If a get request is issued for a variable that does not exist, the value returned is the same as the variable name.

3. CICGETV uses a standard save area convention with R13 pointing to an 18 fullword length area. R1 points to a standard parameter list. R14 contains the return address. R15, upon entry to CICGETV, returns CICGETV's entry point. Upon return, R15 contains the return code.

4. The CICGETV module is located in the REXX/CICS distribution library.

5. A return code of zero is reflected to indicate the success of the operation, or a decimal 99 indicating an internal error (such as a storage limit exceeded situation).

Command Definition

Chapter 22. REXX/CICS DATABASE 2 Interface

The REXX/CICS DATABASE 2 (DB2) Interface provides a means of executing SQL from a REXX exec. The SQL are prepared and executed dynamically. The REXX/CICS DB2 provides the results of the SQL in REXX predefined variables.

The REXX/CICS DB2 supports DB2 versions 7.1 and above. This chapter explains how to use the interface to DB2 from REXX/CICS. If you need more information about SQL, refer to the *DB2 SQL Reference*.

The following information is provided in this chapter:

- Programming considerations
- Embedding SQL

22.1 Programming Considerations

To embed SQL within a REXX exec, the host command environment must be changed. The ADDRESS instruction, followed by the name of the environment, is used to change the host command environment. The ADDRESS instruction has two forms; one affects all commands issued after the instruction, and one affects only a single command. For more information about host command environments, see section 8.2.3, "Changing the Host Command Environment" on page 97 and for more information about the ADDRESS instruction, see section 13.1, "ADDRESS" on page 156.

The REXX/CICS command environment that supports the REXX/CICS DB2 is:

EXECSQL   the command environment that supports SQL.

Note: EXECSQL is an authorized commands. You must be a REXX/CICS authorized user to use the EXECSQL command environment.

REXX/CICS provides an exec called CICRXTRY that can be used to interactively process REXX statements and commands. CICRXTRY can be pseudo-conversational. The PSEUDO and SETSYS PSEUDO commands are used to turn pseudo-conversational mode on or off. If the environment is set to pseudo-conversational, SQL statements issued from CICRXTRY will be committed. If the environment is set to conversational, any SQL statements issued from the CICRXTRY exec will not be committed and any resources that are locked will remain locked until you exit the CICRXTRY exec or issue a CICS SYNCPOINT command. Similar considerations should be made if embedding SQL statements in lengthy REXX execs.

22.2 Embedding SQL Statements

You can use the EXECSQL command environment to process the SQL. Each SQL statement is prepared and executed dynamically using the CICS/DB2 attachment facility.

You can make each request by writing a valid SQL statement as a REXX command directed to the EXECSQL environment. The SQL statement is made up of the following elements:

- SQL keywords
- Pre-declared identifiers
- Literal values.

Use the following syntax:

```
"EXECSQL statement"

      or

ADDRESS EXECSQL
"statement"
"statement"
      .
      .
      .
```

SQL can exist on more than one line. Each part of the statement is enclosed in quotes and a comma delimits additional statement text as follows:

```
ADDRESS EXECSQL
"SQL text",
"additional text",
      .
      .
      .
"final text"
```

The following rules apply to embedded SQL:

- You can pass the following SQL directly to the EXECSQL command environment:

  > ALTER
  > CREATE
  > COMMENT ON
  > DELETE
  > DROP
  > EXPLAIN
  > GRANT
  > INSERT
  > LABEL ON
  > LOCK
  > REVOKE
  > SELECT
  > SET CURRENT SQLID
  > UPDATE.

- You cannot use the following SQL:

  > BEGIN DECLARE SECTION
  > CLOSE
  > COMMIT
  > CONNECT
  > DECLARE CURSOR
  > DECLARE STATEMENT
  > DECLARE TABLE
  > DESCRIBE

> END DECLARE SECTION
> EXECUTE
> EXECUTE IMMEDIATE
> FETCH
> INCLUDE
> OPEN
> PREPARE
> ROLLBACK
> SET CURRENT PACKAGESET
> SET HOST VARIABLE
> WHENEVER.

- Host variables are not allowed within the SQL. Instead, you can use REXX variables to pass input data to the EXECSQL environment. The REXX variables are not embedded within quotes. The output from the EXECSQL environment is provided in REXX predefined variables (see section 22.2.1, "Receiving the Results").

- When you code a SQL SELECT statement, you cannot use the INTO clause. Instead, the REXX/CICS DB2 returns the requested items in compound variables with stem names equal to the DB2 column names.

- The default number of rows returned for a SELECT statement is 250. If you need more or less rows, you can set the REXX variable SQL_SELECT_MAX before issuing the SELECT statement.

22.2.1 Receiving the Results

The EXECSQL command environment returns results in predefined REXX variables. These variables are:

RC   Each operation sets this return code. Possible values are:

    n   Specifies the SQLCODE if the SQL statement resulted in an error or warning.

    0   The SQL statement was processed by the EXECSQL environment. The REXX variables for the SQLCA contain the completion status of the SQL statement.

    30   There was not enough memory to build the SQLDSECT variable.

    31   There was not enough memory to build the SQL statement area.

    32   There was not enough memory to build the SQLDA variable.

    33   There was not enough memory to build the results area for the SELECT statement.

SQLCODE etc   A set of SQLCA variables are updated after SQL statements are processed. The entries of the SQLCA are described in section 22.2.2, "Using the SQL Communications Area" on page 334.

SQL_COLNAME.*n* Contains the name of each DB2 column whose data was returned by a SELECT statement. SQL_COLUMNS should be used as the maximum value for *n*.

SQL_COLTYPE._n_   Contains the type of each DB2 column whose data was returned by a SELECT statement. SQL_COLUMNS should be used as the maximum value for _n_.

> **Note:** Although all data types are supported, not all are displayable. REXX functions can be used to convert the data into the format desired.

> For information about the meaning of specific SQLTYPE codes found in SQL_COLTYPE, see the _DB2 Server for VSE & VM SQL Reference_, SC09-2671.

SQL_COLLEN._n_   Contains the length of each DB2 column whose data was returned by a SELECT statement. If the data type is DECIMAL, the scale is placed after the length of the column (after one blank space). SQL_COLUMNS should be used as the maximum value for _n_.

SQL_COLUMNS   Contains the count of the number of columns returned.

column.n   The results of a SQL SELECT statement are stored in these REXX compound variables. The _column_ is the name of the DB2 column. Each item contains data for one row from DB2. The count of the number of SQL rows returned is contained in _column_.0. The count should be used as the maximum value for _n_.

SQLCOL_n_**.1**   Some SELECT functions such as CURRENT SQLID, MAX, and AVG are not associated with a particular DB2 column. To view the results you must reference column name SQLCOL_n_.1.

> The _n_ begins with, and is incremented by one, for each function included in the SELECT statement. All columns represented by SQLCOL_n_ appear in the SQL_COLNAME compound variable.

22.2.2 Using the SQL Communications Area

The fields that make up the SQL Communications Area (SQLCA) are automatically included by the REXX/CICS DB2 when you issue SQL. The SQLCODE and SQLSTATE fields of the SQLCA contain SQL return codes. These values are set by the REXX/CICS DB2 after each SQL statement is executed.

The SQLCA fields are maintained in separate variables rather than in a contiguous data area. The variables that are maintained are defined as follows:

SQLCODE   The primary SQL return code.

SQLERRM   Error and warning message tokens. Adjacent tokens are separated by a byte containing X'FF'.

SQLERRP   Product code and, if there is an error, the name of the module that returned the error.

SQLERRD._n_   Six variables containing diagnostic information. (The variable _n_ is a number between 1 and 6.)

> **Note:** The count of the number of SQL rows affected by the DELETE, INSERT, and UPDATE command is contained in SQLERRD.3.

> **SQLWARN.***n*   Eleven variables containing warning flags. (The variable *n* is a number between 0 and 10.)
>
> **SQLSTATE**   The alternate SQL return code.

22.2.3 Example Using SQL Statements

In the following example, the REXX/CICS exec prompts for the name of a department, obtains the names and phone numbers of all members of that department from the EMPLOYEE table, and presents that information on the screen.

```
/****************************************************/
/* Exec to list names and phone numbers by department */
/****************************************************/

/*----------------------------------------------------------------*/
/* Get the department number to be used in the select statement */
/*----------------------------------------------------------------*/
   Say 'Enter a department number'
   Pull dept

/*----------------------------------------------------------------*/
/* Retrieve all rows from the EMPLOYEE table for the department */
/*----------------------------------------------------------------*/
   "EXECSQL SELECT LASTNAME, PHONENO FROM EMPLOYEE ",
                 "WHERE WORKDEPT = '"dept"'"
   If rc <> 0 then
     do
       Say ' '
       Say 'Error accessing EMPLOYEE table'
       Say 'RC      =' rc
       Say 'SQLCODE =' SQLCODE
       Exit rc
     end

/*-------------------------------------*/
/* Display the members of the department */
/*-------------------------------------*/
   Say 'Here are the members of Department' dept
   Do n = 1 to lastname.0
     Say lastname.n phoneno.n
   End

   Exit
```

Chapter 23. REXX/CICS High-level Client/Server Support

Client/Server computing has become very popular in the Information Processing industry. Some of the advantages of client/server computing are:

- The ability to effectively integrate the strengths of mainframes, mini-computers, and new cost-effective workstations, in a transparent fashion.

- The ability to incrementally scale the size of computer's systems, up or down (right-sizing).

- The simplification of complex application systems by breaking them down into manageable sets of clients and servers.

- Applications and data can be distributed throughout a network for better performance, integrity or security.

- The ability for enterprise-wide access to data and applications, wherever they reside in the network, from a variety of unlike computer systems or workstations.

23.1 Overview

REXX/CICS introduces REXX language Client/Server support. REXX/CICS provides a high-level client/server capability. This capability includes:

- High-level, natural, transparent REXX client interface
- Support for REXX-based application clients and servers.

23.1.1 High-level, Natural, Transparent REXX Client Interface

REXX/CICS supports a high-level easy to use interface from client REXX execs to application servers through the ADDRESS keyword instruction in REXX. The ADDRESS instruction includes an external environment name that is used to determine the name of the external procedure that is called to process subsequent REXX command strings.

REXX/CICS provides the optional ability for the environment name (specified in the ADDRESS instruction) to be the name of an application server. This capability is provided by the REXX/CICS DEFCMD and DEFSCMD commands.

The DEFCMD command provides the ability to define (or redefine) REXX commands and environments, and it provides the ability to specify whether an environment-command combination is to be handled by a traditional CALLed routine or by an REXX application server.

23.1.2 Support for REXX-based Application Clients and Servers

In addition to the above REXX client interface, several facilities provide support for the use of application servers written in REXX. One of these facilities is the WAITREQ command, which is used by servers to wait for requests from clients. Another facility, the C2S and S2C commands, provide the ability for servers to fetch or set the contents of client variables. Another capability, Automatic Server Initiation (ASI) provides for servers to be started automatically when a request arrives from a client.

337

23.2 Value of REXX in Client/Server Computing

Some advantages of using REXX for implementing client/server solutions are:

- The availability of REXX interpreter support under REXX/CICS with its quick development cycle and excellent source-based interactive debugging allows the rapid prototyping and development of complex systems.

- The high-level client/server interfaces in REXX/CICS can improve development productivity and lower maintenance costs

- Because REXX/CICS allows REXX clients and servers to be recoded in non-REXX languages, performance intensive parts of an application system can be selectively rewritten, if needed.

The FLST and EDIT commands that REXX/CICS provides are examples of client/server environments.

23.3 REXX/CICS Client Exec Example

```
/* EXAMPLE REXX/CICS EXEC */

TRACE 'O' /* turn off source tracing */

ARG parm1 parm2 parm3

"CICS READQ TS QUEUE(MYQ) INTO(DATA) ITEM(5) NUMITEMS(1)"
if rc ¬= 0 then EXIT 100

SAY 'TSQ Data=' data
"CICS SEND TEXT FROM(DATA) ERASE"

/* Define the SERVER EXEC as a REXX/CICS command */
'DEFCMD REXXCICS SERVER = = SERVER1 (REXX'


/* example of directing a subcommand to a server */
/* named SERVER1, which is written in REXX also  */
DATA = 1
'SERVER COMMAND1 DATA'
say data  /*  ==> 2  */
if rc ¬= 0 then SAY 'Request to SERVER1 failed, RC=' rc
EXIT
```

23.4 REXX/CICS Server Exec Example

```
/* EXAMPLE REXX/CICS SERVER1 EXEC */

TRACE 'O' /* turn off source tracing */

/*----------------------------------------*/
/* Loop waiting on requests from clients  */
/*----------------------------------------*/
Do Forever
  'WAITREQ'
  parse var request cmd varname
  Select
    When request = 'COMMAND1' then CALL command1
    When request = 'COMMAND2' then CALL command2
    When request = 'STOP'     then CALL stop_server
    Otherwise
  End /* Select */
End /* Do Forever */
exit

/* subroutine to process command1 */
Command1:
'C2S' varname 'WORK'
WORK = WORK + 1
'S2C WORK' varname
 return

/* subroutine to process command2 */
Command2:
 return

/* routine to shut down this server */
stop_server:
 say 'The Server is stopping'
 exit
```

Chapter 24. REXX/CICS Panel Facility

24.1 Facility

The REXX panel facility provides the REXX programmer with simple tools and commands for panel definition and for panel input/output to 3270 type terminals. The panel facility allows easy definition of panels using any editor. The requirement is that the panel source definition file should be in the REXX File System (RFS) before it is further processed. The panel input/output command provides the ability within a REXX program to dynamically change many of the field attributes statically defined by the panel definition facility. The following example helps you see the capability and function of the panel facility and also helps you understand and visualize the general concepts described in this chapter. An overview of the example follows:

- Defines field control characters that set the characteristics of panel fields.

- Uses the field control characters to define a panel layout. The control character definition and the panel definition (the two parts together are called **panel source**) is saved in the REXX File System.

- Uses the panel source to generate a **panel object** that is used to send or receive panels in a REXX program.

24.1.1.1 Example of Panel Definition

```
** SAMPLE PANEL DEFINITION.

define the field control characters to be used in the panel layout.
.DEFINE < blue protect
.DEFINE @ blue skip
.DEFINE ! red protect
.DEFINE > green unprotect underline
.DEFINE # green unprotect numeric right underline

define a panel named applican, which
queries an applicant's name and address.
(this line and the above lines are treated as comments).

.PANEL applican


< Please type the requested information below @

                    !Applicant's name
@Last name ...:>&lname                 @
@First name ..:>&fname                 @
@MI...........:>1&mi

                    !Applicant's mailing address

@Street.......:>&mail_street                          @
@City.........:>&mail_city                 @
@State........:>2&mail_state
@Zip...:#5&mail_zip

.PANEL

**  END OF SAMPLE PANEL DEFINITION.
```

```
** START OF REXX PROGRAM USING THE PREVIOUS PANEL.

/* program to query applicant's name and address */
lname = '';  /* null out all name parts */
fname = '';
mi = '';
mail_street = '';
mail_city = 'DALLAS';  /* prefill the most likely response for city/state */
mail_state = 'TX';
mail_zip = '';
do forever;
  'panel send applican cursor(lname)';
  if rc > 0 then
    call error_routine;

  'panel receive applican'; /* pseudo-conversational this would be separate */
  if pan.aid = 'PF3' | pan.aid = 'PF12' then
    leave;
  if pan.aid = 'ENTER' & pan.rea = 124 then
    iterate;
  if pan.aid = 'CLEAR' | substr(pan.aid,1,2) = 'PA' then
    iterate;  /* go to beginning of loop */
  if rc > 0 then
    call error_routine;

  /* process the name and address */

end;
'panel end';
exit

error_routine:
  Say 'An error has occurred'
return;

** END OF REXX PROGRAM and end of sample.
```

24.2 Defining Panels

When you define a panel it requires two steps:

1. You must first use an editor to create the panel source file in the REXX File System. The panel source should contain field control characters and the panel layout. The panel layout may contain field control characters along with regular displayable text characters, and possibly, imbedded variable names.

2. Then you convert the panel source into an intermediate form (panel object) that can be used by the panel input/output commands. The panel facility automatically generates the intermediate file when the input/output command that is referencing that panel is first invoked or an explicit command within the REXX environment (or a REXX exec containing the command) can be invoked to generate the intermediate file.

   Note: This automatic generation is executed whenever the panel object cannot be found or when the panel source has a date or time change that is later than the panel object. Any change to the panel source causes a new panel object to be created. Therefore, be cautious when you change a panel

source after a program using that panel is out of the testing phase. It is to your advantage to move the panel source out of the RFS or into an RFS directory not accessible by the program after the project goes into production.

24.3 Defining the Field Control Characters with the '.DEFINE' Verb

The field control characters define the attributes of fields on the panel. These control characters are definable by using the .DEFINE verb and must precede the panel layout. The .DEFINE verb is terminated at 'End of line' unless the continuation character (a comma) is the last character on the line . The continuation character cannot immediately follow the .DEFINE verb because of the ambiguity of whether the comma is the control character being defined or a continuation character. Spaces delimit each keyword and order is unimportant except the control character that is defined must immediately follow the verb. If any text does not start with .DEFINE in column one it is ignored and treated as a comment unless the line is a continuation. A total of 32 control characters (including default control characters) can be actively defined at one time. If all characters are deleted by using the DROP keyword then the five default control characters are re-activated. Certain keyword combinations are incompatible and are not allowed while others which may seem meaningless are allowed. For example, INVISIBLE and color. This may be useful when the field attribute is changed dynamically within a REXX program (the invisible field can be made visible which makes color meaningful).

The characteristics of the .DEFINE verb follow.

- It must start on the first column, followed by a space, and capitalized.

- It terminates at 'End of line' unless a continuation character (a comma) is used.

- All keywords have a minimum of a two-character abbreviation.

- The maximum number of control characters that you can define at one time is 32. (The default control characters are included in this count.)

- It must be placed before the .PANEL verb.

You can also define variable identifier control characters that let you associate REXX variables with Panel Facility variables. Then, in your panel definition you can imbed the variable identifier control character instead of the REXX variable name. The same REXX variable can be assigned to different variable identifier control characters.

- A stem name must end in a period.

- A variable list can have more variables listed than are used, but less causes an error.

- Multiple variable identifier control characters can be defined and each one is independent of one another.

24.4 .DEFINE

The format of the .DEFINE verb follows. Also, the default control characters are specified if you do not want to define your own.

```
►►──── .DEFINE──char──────┬────────────────────────┬──────────────────►◄
                          ├─┤ options ├────────────┤
                          ├─VAriable───────────────┤
                          │         ┌─stem.─────┐   │
                          │         │  ┌──────┐ │   │
                          │         └──┴variable┴──┘   │
                          └─DROP───────────────────┘
```

options:

```
   ┌─UNProtect─┐  ┌─NORmal────┐        ┌─DEfcolor──┐  ┌─BLInk─────┐
├──┼───────────┼──┼───────────┼────────┼───────────┼──┼───────────┼──►
   ├─PRotect───┤  ├─BRight────┤        ├─GReen─────┤  ├─REVerse───┤
   └─SKip──────┘  └─INVisible─┘        ├─RED───────┤  └─UNDerline─┘
                                       ├─BLUe──────┤
                                       ├─TUrquoise─┤
                                       ├─WHite─────┤
                                       ├─YEllow────┤
                                       └─PInk──────┘

   ┌─NOJustify─┐          ┌─NULls─┐
►──┼───────────┼──PAd──(──┼───────┼──)──NUMeric──CUrsor──MDT──────┤
   ├─LEft──────┤          ├─BLAnks┤
   └─RIght─────┘          └─char──┘
```
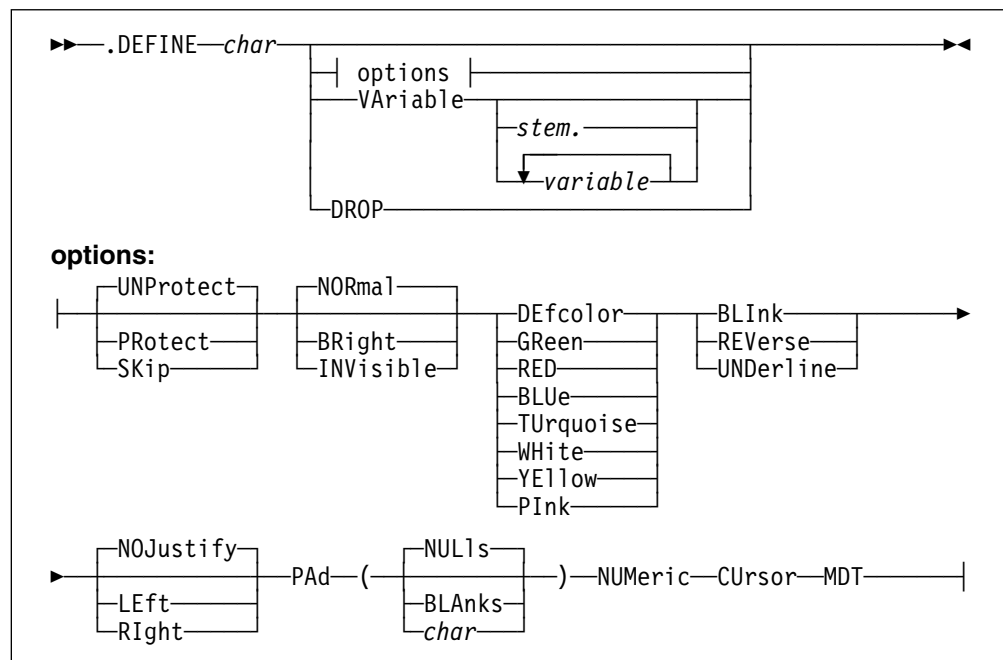
24.4.1.1 Default field control characters

Defcolor skip normal
+   Defcolor protect bright
%   Defcolor unprotect normal
!   Defcolor unprotect bright
&   Variable identifier

24.4.1.2 Operands

char   specifies the control character being defined.

VAriable   defines a REXX variable identifier control character. Variable identifier control characters are used to associate Panel Facility control characters with REXX variable names. More than one variable control character can be defined at one time. Following the VARIABLE keyword may be a list of variable names (*variable*) or a single stem name (*stem.*). The variable list can contain one to 32,767 variable names. Only one stem name can be specified and the stem name must end in a period. This period identifies the variable as a stem and leaving off the period causes the name to be interpreted as a simple variable.

Using a variable list and a stem name cannot be mixed. When the panel generator encounters a variable control character a substitution is done. Simple variable lists are substituted in the same order as listed. For example, the third variable control character is replaced with the third variable listed for that control character. The stem variable is replaced by appending a three-character number (tail) to the stem name. The number starts at 1 and is incremented as that stem control character is encountered. Therefore, the tenth stem control character for

a particular stem would have a 10 as the tail (STEM.10). Since these variables are REXX variables, they must follow the REXX variable naming rules.

DROP   drops *char* as a field control character.

24.4.1.3 Options
UNProtect

specifies that the field is not protected from operator input. (This is the default.)

PRotect   specifies that the field is protected from operator input.

SKip   specifies a protected field with the auto-skip feature. Operator entering a character in the last position of the previous unprotected field causes the cursor to skip over this field.

NORmal   specifies that the field is not highlighted. (This is the default.)

BRight   specifies that the field is highlighted.

INVisible specifies that the field is invisible.

GReen
RED
BLUe
TUrquoise
WHite
YEllow
PInk
DEfcolor are the choices for the color.

> **Notes:**
>
> 1. When you do not specify a default color, the color is based on the field type and intensity values: protect/normal displays blue, protect/bright displays white, unprotect/normal displays green, and unprotect/bright displays red.
>
> 2. If any field on a panel has explicitly specified a color (including DEFCOLOR), all bright fields with DEFCOLOR or no color specified are displayed white and all normal fields with DEFCOLOR or no color specified are displayed green. This is a 3270 hardware limitation and not the panel facility.

BLInk   specifies that the field blinks.

REVerse specifies that the field is in reverse video.

UNDerline

specifies that the field is underlined.

NOJustify

specifies that justification is not done (left justified but blanks are not stripped).

LEft   specifies that the field is left justified (leading blanks are stripped).

RIght   specifies that the field is right justified (trailing blanks are stripped).

PAd()	specified only in the context of fields having variables. In an unprotected field the pad character fills the character positions that are not occupied by a variable value. In a protected field, the pad character is similar but the scope of the fill area is not the whole field as in the unprotected field. It is bound by where the variable starts, within the protected field to either the end of the field or the start of the next variable or text.
NULls	specifies that a field will be padded with the null character.
BLAnks	specifies that a field will be padded with blanks.
char	specifies a single character to be used to pad a field.
NUMeric	specifies a field is numeric (unprotected field only).
CUrsor	specifies that the cursor is positioned at the beginning of this field. If multiple cursor fields are defined, then the last one defined contains the cursor. The cursor is placed in the top left corner if a cursor field is not defined.
MDT	sets the modify bit tag on for the field. Always return this field on a read, even if the field was not modified by the operator.

24.5 Defining the Actual PANEL Layout with the '.PANEL' Verb

Following the .DEFINE verbs should be a .PANEL verb that signals the start of the panel layout. This verb must also start at column one and be capitalized. The lines in the panel definition should be specified in the same positions that you want them displayed. All text entered after the panel verb is significant to the panel generator, including blank lines, and therefore comment lines are not allowed. The first character has to be a protect/skip or unprotect control character. The panel definition ends at the end of file or the next .PANEL verb starting in column one. Only one panel definition per file is supported.

The panel layout is close to what you see, with the exception of the control characters and the imbedded variables which are not shown when the panel is displayed. A field typed on the third line after the .PANEL starting at column ten is positioned on the terminal screen third line, column ten.

The characteristics of the .PANEL verb follow.

- It must start in the first column, followed by a space, and capitalized.

- It must have a panel name on the same line as .PANEL, unless it is an end of panel indicator.

- It must have at least one field and the first character following the .PANEL line has to be a protect, skip, or unprotect control character.

- The field ends at the start of the next field unless an explicit input field length is being used. An empty field can be used to terminate a field.

- To use a control character as a regular displayable character, type two consecutive control characters. To display two consecutive control characters, type four consecutive control characters. For each pair of control characters that get displayed as one character, the subsequent text in the field will be

displayed one position to the left, except if there are 7 or more consecutive spaces following the pair of control characters.

- The protect/skip field can have any number of variables or text that will fit in that field. A variable in a protected field has area available for substitution, starts at the variable identifier control character, and ends just before the next control character or text.

- Space is not allowed between a variable identifier control character and the variable name. A space causes the variable name to be interpreted as plain text.

- The unprotect field can have only one associated variable.

- The unprotected field can have a number explicitly stating the field length. The number must be between the unprotect and variable control characters. When the explicit field length is used, a field does not need termination. If an explicit length field is the last field of a line then a terminating field is created with a skip attribute to force a field end.

- When an explicit input field length is used, all of the following field's column positions are adjusted left or right to force proper alignment, so the explicit length field and the field immediately after abut each other. Other fields on that line keep its spacing intact. This alignment only affects that one line, the fields on the next line are unaffected. For example, a line has 5 fields with the 1st and 4th fields having explicit length and spacing. Between fields 1 and 2 is 4, between 2 and 3 is 5, between 3 and 4 is 6, and between 4 and 5 is 7. When this line is displayed, field 2 starts right after field 1 (no space), separation between fields 2 and 3 is still 5, separation between fields 3 and 4 is still 6, and field 5 follows right after field 4 (no space). If the explicit length causes the current field or subsequent fields to overflow into the next field, an error is returned when the panel is displayed.

- When a field does not terminate on the same line (for example, field spans lines), the field length varies with the width of the screen on which the panel is being displayed. Also, if the last field has no terminator, that field wraps the screen until the first field start is encountered.

- Only the fields containing a variable can have its attributes dynamically changed during panel output.

- If a protect/skip field is dynamically changed to an unprotected field, only the first variable in the field has the operator input assigned to it. All of the contents of the input field are assigned.

- The panel source file must reside in the REXX File System before it can be processed into the intermediate file (panel object) used by the runtime panel facility.

24.6 .PANEL

The format of the .PANEL verb follows.

```
►►──.PANEL──panel_name───────────────────────────────────────►

 ►──────┬─protect_cc─┬──────┬─variable_pair─┬───────────────────►
        └─skip_cc────┘      ├─text──────────┤
                            └─variable_cc───┘
        └─unprotect_cc──────┬─────────────┬──┬─variable_cc───┐
                            └─field_length┘  └─variable_pair─┘

 ►─────┬─────────┬───────────────────────────────────────────►◄
       └─.PANEL──┘
```
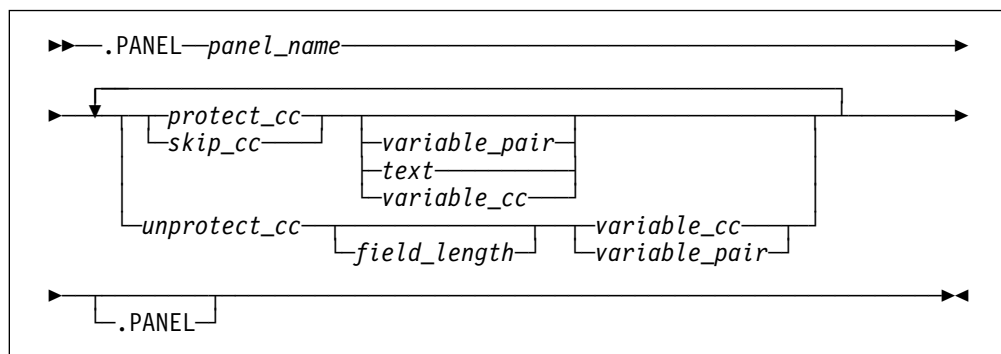
24.6.1.1 Operands

panel_name

specifies the panel being defined. It must be one to eight characters in length and follow the rules for REXX File System file names. (See Chapter 19, "REXX/CICS File System" on page 299, for more information.)

Note: The *panel_name* must be the same as the RFS file name. The complete RFS file name should be *panel_name*.PANSRC.

protect_cc

specifies the protect field control character.

skip_cc   specifies the skip field control character.

variable_pair

specifies the variable control character followed by the variable name. (There cannot be a space between them.)

text   displayable characters.

variable_cc

specifies the variable identifier control character.

unprotect_cc

specifies unprotect control character.

field_length

specifies the explicit input field length value.

24.7 Panel Generation and Panel Input/Output

Panel definition can be done outside the REXX environment; however, panel generation and input/output is performed in a REXX exec or in the REXX interactive environment. The REXX interactive environment is an ideal place to test the initial panel development. To test display the panel use the TEST panel command. This displays the panel with no panel object file created. Also, there is no substitution for the variables on the panel. To create the panel object use either the GENERATE, SEND, or the CONVERSE panel commands. Use the FILE keyword to explicitly state what directory in RFS to find the panel source, or you can let it default to the current directory. The panel source name must have the panel name as the file name and 'PANSRC' as the file type. The panel object is created and filed in the same directory as the panel source with the file name equal to the source file name and with a file type of 'PANOBJ'. GENERATE creates the

panel object and does not display the panel. SEND creates the panel object, displays the panel, and attempts variable substitution. CONVERSE is similar to SEND with an implied wait and receive.

Note: There are side affects of being in a REXX interactive environment. Several panel keywords act differently: the cursor position on the SEND is ignored and keyboard lock is also ignored for SEND and CONVERSE.

The characteristics of the PANEL command follow.

- All the arguments or keywords are not meaningful or valid for all commands.

- The last panel command in a REXX exec is the END command. This releases any storage held by previous panel commands. This command needs no other operands.

- Only the fields with associated variables can have their attributes changed dynamically.

- A panel object file is created for the panel that is being displayed if one does not exist. The file name is the same as the panel source file name and the file type will be `'PANOBJ'`.

- Only the first variable in a protect/skip field is assigned the input entered by the terminal operator when the field is changed to an input field.

- The dynamic attribute changes effect the present panel execution and do not last. Subsequent panel displays revert to what was defined statically by the panel definition step unless the attributes are again dynamically changed.

- A SEND must be performed before a RECEIVE and the panel names must match.

- Only the attributes indicate change, the others stay as defined statically.

- A panel sent with a position argument needs to be received with the same position values so the operator input is assigned correctly to the REXX variables.

- Enclose the field ID list within parenthesis, when multiple fields are listed.

- There has to be one ATTRIBUTE argument for each different attribute change.

Note: To change the number of ATTRIBUTE arguments dynamically, a REXX variable needs to be used (put the literal attribute string in REXX variable and use the variable). For example, a program needs to change one field to blue or to change one field to blue and another field to blinking depending on operator input. One solution is:

```
field_id = 'xxxx'; /* name of field needing attribute changed*/
attr_string = 'attr('  field_id  'blue )';
if operator_input = y then
  attr_string = attr_string  'attr(' field_id2  'blink )'
'panel send panel_name' attr_string;
```

Note: REXX panel facility generates the panel object if the object is not found or if the object is older than the source.

24.8 PANEL RUNTIME

The format of the PANEL RUNTIME commands follow.
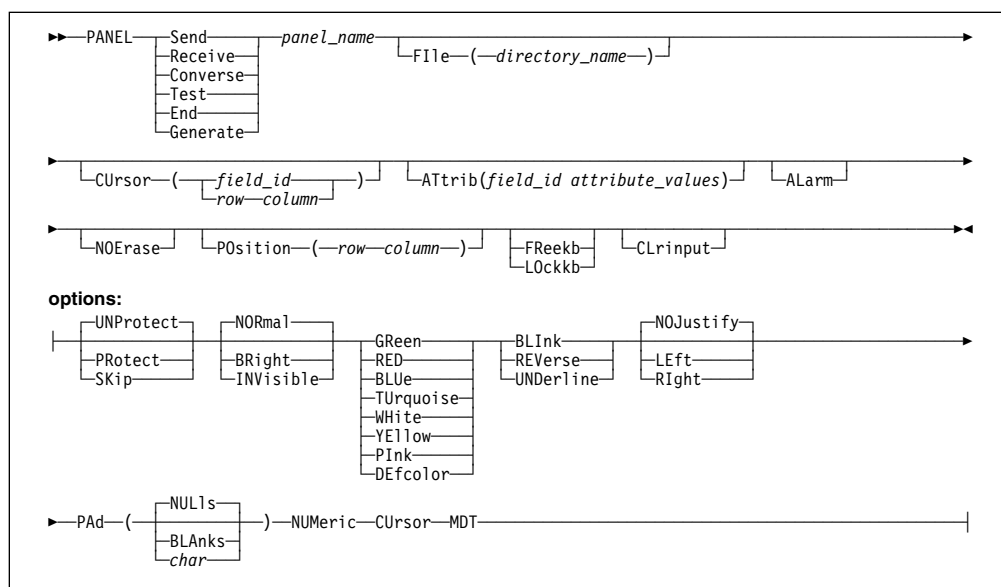
24.8.1.1 Operands

Send is the panel command that sends a panel.

Receive is the panel command that receives a panel.

Converse

  is the panel command that sends a panel and waits for operator input.

Test is the panel command that displays a panel. An intermediate file (panel object) is not created and variable substitutions are not attempted.

End is a command that terminates the panel session. Command releases all storage held by the panel facility. This command does not have arguments and any arguments supplied are ignored.

Generate is an explicit command that creates a panel object. The panel is not displayed.

panel_name

  specifies the name of the panel to input/output or generate.

FIle() specifies the name of the RFS directory (*directory_name*) containing this panel. (Specified for all panel commands except END.)

CUrsor() (specified for SEND and CONVERSE only) positions the cursor on the panel.

field_id specifies the REXX variable name where the cursor should be positioned on the panel.

row specifies the row within the panel where the cursor should be positioned. The row value is relative to the starting row of the panel. The default starting row of the panel is 1, but may be changed using the POSITION() keyword.

column   specifies the column within the panel where the cursor should be positioned. The column value is relative to the starting column of the panel. The default starting column of the panel is 1, but may be changed using the POSITION() keyword.

ATtrib(*field_id attribute_values***)**
          (specified for SEND and CONVERSE only)

          field_id   specifies the field whose attributes are dynamically set. It must be a variable name associated with the field. The field list must be enclosed with parenthesis. Only the attributes stated are changed and the other attributes default to what was statically defined. A field defined originally as RED and UNDERLINE remains underlined if only blue is stated dynamically.

ALarm   (specified for SEND and CONVERSE only) sounds the bell when displaying panel. (The default is no alarm).

NOErase   (specified for SEND and CONVERSE only) do not erase the screen before displaying this one. (The default is erase before a panel write).

POsition()
          (specified for SEND, CONVERSE, and RECEIVE only) positions the panel on the output screen. Row (*row*) and column (*column*) specifies where the top left corner of the panel should begin. (The default is row 1 col 1). For example, POS(5 10) means to have the panel start in row 5 and column 10, the actual movement is 4 rows down and 9 columns to the right.

FReekb   (specified for SEND and CONVERSE only) frees the keyboard, allowing operator input. (This is the default.)

LOckkb   (specified for SEND and CONVERSE only) locks the keyboard.

CLrinput   (specified for SEND and CONVERSE only) clears all input fields before displaying the panel. Variable substitution is not attempted and pad characters fill the input area.

24.8.1.2 Options
UNProtect
          specifies that the field is not protected from operator input.

PRotect   specifies that the field is protected from operator input.

SKip   specifies a protected field with the auto-skip feature. Operator entering a character in the last position of the previous unprotected field causes the cursor to skip over this field.

NORmal   specifies that the field is not highlighted.

BRight   specifies that the field is highlighted.

INVisible   specifies that the field is invisible.

GReen
RED
BLUe
TUrquoise
WHite
YEllow
PInk
DEfcolor   are the choices for the color.

> **Notes:**
>
> 1. When you do not specify a default color, the color is based on the field type and intensity values: protect/normal displays blue, protect/bright displays white, unprotect/normal displays green, and unprotect/bright displays red.
>
> 2. If any field on a panel has explicitly specified a color (including DEFCOLOR), all bright fields with DEFCOLOR or no color specified are displayed white and all normal fields with DEFCOLOR or no color specified are displayed green. This is a 3270 hardware limitation and not the panel facility.

BLInk   specifies that the field blinks.

REVerse   specifies that the field is in reverse video.

UNDerline
   specifies that the field is underlined.

NOJustify
   specifies that no justification is done (left justified but blanks are not stripped).

LEft   specifies that the field is left justified (leading blanks will be stripped).

RIght   specifies that the field is right justified (trailing blanks will be stripped).

PAd()   specified only in context of fields having variables. In an unprotected field the pad character fills the character positions that are not occupied by a variable value. In a protected field, the pad character is similar but the scope of the fill area is not the whole field as in the unprotected field. It is bound by where the variable starts, within the protected field to either the end of the field or the start of the next variable or text.

NULls   specifies that a field will be padded with the null character.

BLanks   specifies that a field will be padded with blanks.

char   specifies a single character to be used to pad a field.

NUMeric   specifies a field is numeric (unprotected field only).

CUrsor   specifies that the cursor is positioned at the beginning of this field. If multiple cursor fields are defined, then the last one defined contains the cursor. The cursor is placed in the top left corner if a cursor field is not defined.

MDT   (specified for SEND and CONVERSE only) sets the modify bit tag for all input fields on the panel.

24.8.1.3 PANEL Variables
The implicitly defined PANEL variables that can be used by the REXX program follow.

PAN.AID Attention identifier that last caused panel input.

```
ENTER    ENTER
CLEAR    CLEAR
CLRP     CLEAR PARTITION
PEN      SELECTOR PEN
OPID     OPERATOR ID
MSRE     MAGNETIC READER
STRF     STRUCTURE FIELD
TRIG     TRIGGER
PA1
PA2
PA3
PF1
PF2
PF3
PF4
PF5
PF6
PF7
PF8
PF9
PF10
PF11
PF12
PF13
PF14
PF15
PF16
PF17
PF18
PF19
PF20
PF21
PF22
PF23
PF24
```

PAN.CURS

> Position of cursor in last panel input. This is in the form of row column separated by a blank. For example, `'10 5'` would be row 10 and column 5. The row and column values are absolute to the start of the screen and are unaffected by the POSITION() keyword.

PAN.CNAM

> REXX variable name (field ID) associated with the cursor position. If the field has no associated variable then PAN_CNAM is not updated.

The PANEL error related variables follow.

PAN.REA Reason code if warning or error occurs. REXX return code, RC, should be examined first. If the RC is 10, PAN.REA contains state codes and input codes to help in error determination. See section 24.8.1.8, "State Codes and Input Codes" on page 357 for more information.

PAN.LOC Internal location code. Three to four-digit number used by IBM support. If the REXX variable RC contains the value 10, the PAN.LOC should be used in conjunction with PAN.REA for error determination.

PAN.LINE

Line number in the source panel definition where an error was detected, if error occurred during panel object generation.

24.8.1.4 Panel Facility Return Code Information

The main return code for the panel command is set in the REXX variable, RC. Each level of return code is accompanied by additional information in the form of a reason code and a line number (if applicable). When an error is detected while processing panel source code (location code of 11*xx* or 12*xx*) and RC is not 12 or 16, the REXX variable *PAN.LINE* contains the number of the line in error.

24.8.1.5 Return Codes

4   Warning. Panel facility continues processing. Processing stops for other return code values.

8   Programmer error

10   Programmer error, PAN.REA contains more information to help determine the cause of the error. See section 24.8.1.8, "State Codes and Input Codes" on page 357 for more information.

12   CICS command error; the CICS EIBRESP is returned in the panel reason code. If the error is not programmer resolvable, save and collect as much information as needed to recreate the error and contact IBM support.

14   RFS errors; reason code contains the RFS return code

16   Internal system error; save and collect as much information as needed to recreate the error and contact IBM support.

24.8.1.6 System Error Reason Codes

401   Panel facility ran out of storage space while processing the command

402   Internal control character identifier table and control character informational table are out of synchronization

403   Panel object data has been corrupted. First check to see the file is correct and that it is a panel object.

404   CICS receive buffer corrupted

405   Bad validation request

406   Storage free request failed

407   Storage get request failed

408   Attempt to get/put REXX variable failed

409   Aid is unknown

410   Dynamic match error

24.8.1.7 Programmer Introduced Warning/Error Reason Codes

101 Keyword repeated or keyword within a like category was repeated. For example, RED and BLUE; UNDERLINE and REVERSE.

102 Keywords are incompatible. For example, PROTECT and NUMERIC

103 Missing keyword or panel name

104 The control character being defined is invalid or missing

105 Panel is too large for the screen

108 Parenthesis missing

109 Pad character invalid

110 Task has no associated terminal

111 Panel has no input fields defined during a receive

112 Panel name is invalid

115 REXX variable name (or field ID) is invalid

116 Number is specified incorrectly. Explicit length value in panel source or row col value

117 Variable value was too long and was truncated to fit output field

118 Text field was truncated. Check to see if explicit length did not force a subsequent field to overlay another field.

119 Bad or missing panel command. It should be SEND, RECEIVE, CONVERSE, TEST, or END.

122 A modified field was received but it had no corresponding input field definition.

124 Empty received buffer. Clear, ENTER, and the PA keys will cause this.

125 File name is invalid

126 Field identifier for the ATTRIBUTE keyword was not found in the panel.

129 Too many arguments supplied with the command

130 SEND must be performed before a RECEIVE

131 Panel source has no panel definition. '.PANEL' was probably not in column 1 or did not have a space after.

132 Keyword encountered is unknown

133 DROP asked for a control character not presently active

136 Continuation is in effect but end of source was encountered

137 Row or column specified is too large for the current display CURSOR() or POSITION() the likely source of error.

138 Variable identifier control character was being defined and more than one stem name was listed.

139 More variable fields defined than listed in the control character definition.

140 Explicit input field value caused field to go past screen end

143 Panel name in file does not match the panel name in the panel runtime command.

144 Panel has more rows than the present screen allows

145 Panel has more columns than the present screen allows

24.8.1.8 State Codes and Input Codes

For return code 10 the reason code (in PAN.REA) has a special meaning. It consists of two 2-digit numeric codes, the first number stating what keyword is being processed (**state code**) and the second number stating the input that caused the error condition (**input code**). Use the location code (in PAN.LOC) to determine which state code list and input code list to use.

For example:

```
RC = 10
PAN.REA = 0199
PAN.LOC = 1177
```

When you use the state code and input code lists for the 11*xx* location codes, the keyword being processed was .DEFINE for state code 01 and the input code contained an unknown symbol for input code 99.

State codes: State codes for 11*xx* location codes:

01   .DEFINE and control char
02   field type (protect/skip/unprotect)
03   color (red/blue/green/...)
04   Intensity (bright/normal/invisible)
05   Justify (left/right/nojustify)
06   Numeric
07   extended highlight (blink/reverse/underline)
08   MDT
09   Cursor
10   Pad()
11   Variable
12   Drop

State codes: State codes for 20*xx* location codes:

01   panel commands (send/receive/converse/...)
02   File()
03   Cursor()
04   Position()
05   Alarm
06   Noerase
07   Keyboard lock (lockkb/freekb)
08   Clrinput
09   Attribute
10   field type (protect/skip/unprotect)
11   color (red/blue/green/...)
12   Intensity (bright/normal/invisible)
13   Justify (left/right/nojustify)
14   Numeric
15   extended highlight (blink/reverse/underline)
16   MDT
17   Cursor
18   Pad()
19   closing parenthesis of the ATTRIBUTE argument

Input codes: Input codes for 11*xx* and 20*xx* location codes:

01	field type (protect/skip/unprotect)
02	color (red/blue/green/...)
03	Intensity (bright/normal/invisible)
04	Justify (left/right/nojustify)
05	Numeric
06	extended highlight (blink/reverse/underline)
07	MDT
08	Cursor
09	Pad()
10	Variable
11	Drop
12	(not used)
13	File()
14	Cursor()
15	Position()
16	Alarm
17	Noerase
18	Keyboard lock (lockkb/freekb)
19	Clrinput
20	Attribute
21	closing parenthesis of the ATTRIBUTE argument
98	End of command; more operands were expected
99	Unknown symbol; keyword or control character was expected

State Codes: State codes for 12*xx* location codes:

01	panel name
02	protect/skip field
03	unprotect field
04	text within a protect/skip field
05	(not implemented yet)
06	explicit input field length number
07	unprotect variable
08	protect/skip variable

Input codes: Input codes for 12*xx* location codes:

01	Plain displayable text
02	Explicit length number
03	Protect field control character
04	Unprotect field control character
05	Variable control character
07	End of panel
08	Invalid or unknown input

24.8.1.9 Location Codes

Numbers under 1000 are in the mainline processor.

10*xx*	Panel generator common processor
11*xx*	.DEFINE verb processor
12*xx*	.PANEL verb processor
20*xx*	Panel runtime command processor
21*xx*	Dynamic attribute resolution processor
30*xx*	Output 3270 data stream processor

40*xx*   Input 3270 data stream and REXX variable assignment processor

90*xx*   CICS interface processor

24.9 Examples of Sample Panels

Five examples of sample panel definitions follow.

24.9.1.1 Example 1

```
*** Sample panel one:

.DEFINE  >  prot  blue
.DEFINE ?  prot red
.DEFINE  #  unprot  num green
.DEFINE <  unprot invisible num
.DEFINE  @ protect  turq
.DEFINE  +  prot  blue underline
.PANEL signon
> Panel signon                              &companyname

?&message

@                     Welcome to ACME On-Line Tax Services

+Please enter your Account Number and Personal ID Number and press ENTER>


>Account Number :#7&account_num

>PIN           :<4&pin
```

24.9.1.2 Example 2
*** Sample panel two:

```
.DEFINE  >  prot  green
.DEFINE <  unprot underline white
.DEFINE  +  var service.
.DEFINE % skip turq
.PANEL   service
> Panel service          &disp_date          &companyname


% &salutation
% Tab the cursor to the type of service wanted and press the ENTER key.


<+> Itemized tax preparation

<+> Non-itemized tax preparation

<+> Query return status

<+> Show calendar

<+> Exit
```

24.9.1.3 Example 3
*** Sample panel three:

```
.DEFINE # protect bright
.DEFINE + protect
A panel to display a static message without erasing previous panel.
Notice the position of the escape sequence in lines 1 and 6.
See manual for an explanation about escape sequences.
.PANEL msgbox1
#++-------------------------------------+++
#|                                       |+
#|   We are sorry but the service you have   |+
#|   chosen is not available at this time.   |+
#|   Press ENTER to continue.            |+
#|                                       |+
#++-------------------------------------+++
```

24.9.1.4 Example 4
*** Sample panel four:

```
.DEFINE ) protect bright
.DEFINE + drop
.DEFINE & var msg.
A panel to display output dynamic messages.
.PANEL msgbox2
)+----------------------------------------+#
)|                                        |#
)|   &                                    |#
)|   &                                    |#
)|                                        |#
)+----------------------------------------+#
```

24.9.1.5 Example 5
*** Sample panel five:

```
.DEFINE  >  skip  blue
.DEFINE <  skip green right
.DEFINE % var center_days.
.DEFINE + var right_days.
.DEFINE # VAR left_days.
.DEFINE @ var pf3 pf7 pf8
.PANEL   calendar
> Panel calendar         &disp_date                        &companyname


      > &disp_left_mon              &disp_center_mon                &disp_right_mon
>su mo tu we th fr sa         su mo tu we th fr sa          su mo tu we th fr sa
<# <# <# <# <# <# <# >      <% <% <% <% <% <% <% >       <+ <+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# <# >      <% <% <% <% <% <% <% >       <+ <+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# <# >      <% <% <% <% <% <% <% >       <+ <+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# <# >      <% <% <% <% <% <% <% >       <+ <+ <+ <+ <+ <+ <+ >
<# <# <# <# <# <# <# >      <% <% <% <% <% <% <% >       <+ <+ <+ <+ <+ <+ <+ >
<# <# >                     <% <% >                      <+ <+ >




>@    = Leave Calendar  >@    = Backup a month   >@    = Go forward a month
   IF RC=4 & PAN.REA = 124 THEN  /* warning and no input received */
```

24.10 Example of a REXX Panel Program

```
                /* REXX program sample using Panel Facility */
                /* data base */
                ACCOUNT.1234561      = '1231 John W. Smith Mr.'
                ACCOUNT.1234562      = '1232 Jane M. Brown Miss'
                ACCOUNT.1234563      = '1233 Mary R. Scott Mrs.'

                MESSAGE = ''             /* no output message yet */
                COMPANYNAME = 'ACME On-Line Tax Services'
                CURS_NAME = 'ACCOUNT_NUM'   /* put cursor on LNAME field */
                ATTR_STRING = ''        /* no dynamic attributes on first send */
                PATH_NAME = 'FILE(POOL1:\USERS\BLAKELY)'
                CLR_INP_FIELDS = 'CLR'


                DO FOREVER
                  'PANEL SEND SIGNON' CLR_INP_FIELDS PATH_NAME ,
                       'CURSOR(' CURS_NAME ')' ATTR_STRING
                  IF RC > 4 THEN  /* more than a warning */
                    SIGNAL ERROR  /* clean up and exit */
                  'PANEL RECEIVE SIGNON '
                  IF RC > 4 THEN
                    SIGNAL ERROR  /* clean up and exit */



                  /* check return code and reason code to see if any data received */
                    ITERATE                /* redisplay panel */

                  CLR_INP_FIELDS = ''    /* display input fields with variable values */
                  IF ¬SEARCH(ACCOUNT_NUM) THEN      /* search for account number */
                    DO;
                      MESSAGE = ' Account Number not found, Please re-ENTER Number'
                      CURS_NAME = 'ACCOUNT_NUM'  /* put cursor in ACCOUNT field */
                      ATTR_STRING = 'ATTR( ACCOUNT_NUM REV)'
                      ITERATE
                    END;


                  IF WORD(ACCOUNT.ACCOUNT_NUM,1) ¬= PIN THEN /* pin mismatch ?  */
                    DO;
                      MESSAGE = ' PIN Number is incorrect, Please check to see your',
                      'Account Number is correct and re-ENTER your PIN';
                      CURS_NAME = 'PIN';
                      ATTR_STRING = 'ATTR( PIN REV)';
                      ITERATE ; /* display the panel again */
                    END;
                 LEAVE ;
                END ; /* forever */


                SERVICE. = '';
                DISP_DATE = DATE('U');     /* set to display current date */
                MSG.1 = 'Be sure cursor is in the first column!';
                MSG.2 = 'Press ENTER or and PF key to continue.';
                SALUTATION = 'Hi' WORD(ACCOUNT.ACCOUNT_NUM,5) ,
                             WORD(ACCOUNT.ACCOUNT_NUM,4) ||,
                        ', How may we be a service to you?';
```

```
DO FOREVER;
  PAN.CNAM = '';
  'PANEL SEND SERVICE CURSOR(SERVICE.1)' PATH_NAME
  IF RC > 4 THEN
    SIGNAL ERROR; /* clean up and exit */
  'PANEL RECEIVE SERVICE'
  IF RC > 4 THEN
    SIGNAL ERROR; /* clean up and exit */
  SALUTATION = '';  /* greeting only once */


  SELECT;
    WHEN PAN.CNAM = 'SERVICE.1' THEN
        CALL ITEMIZE_ROUTINE;
    WHEN PAN.CNAM = 'SERVICE.2' THEN
        CALL NON_ITEMIZE_ROUTINE;
    WHEN PAN.CNAM = 'SERVICE.3' THEN
        CALL QUERY_RET_ROUTINE;
    WHEN PAN.CNAM = 'SERVICE.4' THEN
        CALL CAL;
    WHEN PAN.CNAM = 'SERVICE.5' THEN
        CALL EXIT_ROUTINE;

    OTHERWISE
      DO;
        'PANEL SEND MSGBOX2 POS(7 10) NOERASE' PATH_NAME
        IF RC > 4 THEN
          SIGNAL ERROR;
        'PANEL RECEIVE MSGBOX2'
        IF RC > 4 THEN
          SIGNAL ERROR;
      END;
  END; /* select */
END; /* do forever */
EXIT


SEARCH: ; ARG ACC_NUM ;
IF SYMBOL('ACCOUNT.ACC_NUM') == 'VAR' THEN
  RETURN(1)
ELSE
  RETURN(0);


ITEMIZE_ROUTINE:
NON_ITEMIZE_ROUTINE:
QUERY_RET_ROUTINE:
'PANEL SEND MSGBOX1 POS(7 10) NOERASE' PATH_NAME
 IF RC > 4 THEN
   SIGNAL ERROR;
'PANEL RECEIVE MSGBOX1'
IF RC > 4 THEN
  SIGNAL ERROR;
RETURN;
```

```
CAL: PROCEDURE
COMPANYNAME = 'ACME On-Line Tax Service';
PATH_NAME   = 'FILE(POOL1:\USERS\BLAKELY\)'
DISP_DATE = DATE('U');

/* calling date function in on statement ensures consistent date */
/* data save has format of YYYYMMDDNNNNNN                        */
DATE_SAVE = DATE('S') || DATE('B');


/* set number of days each month has */
NUM_OF_DAYS.1 = 31;
NUM_OF_DAYS.3 = 31;
NUM_OF_DAYS.4 = 30;
NUM_OF_DAYS.5 = 31;
NUM_OF_DAYS.6 = 30;
NUM_OF_DAYS.7 = 31;
NUM_OF_DAYS.8 = 31;
NUM_OF_DAYS.9 = 30;
NUM_OF_DAYS.10 = 31;
NUM_OF_DAYS.11 = 30;
NUM_OF_DAYS.12 = 31;


/* set names of each month for panel display */
MONTH_NAME.1 = 'January';
MONTH_NAME.2 = 'February';
MONTH_NAME.3 = 'March';
MONTH_NAME.4 = 'April';
MONTH_NAME.5 = 'May';
MONTH_NAME.6 = 'June';
MONTH_NAME.7 = 'July';
MONTH_NAME.8 = 'August';
MONTH_NAME.9 = 'September';
MONTH_NAME.10 = 'October';
MONTH_NAME.11 = 'November';
MONTH_NAME.12 = 'December';


/* get number of complete days from year 1900 to 1st of month */
TOT_DAYS = SUBSTR(DATE_SAVE,9,6)-SUBSTR(DATE_SAVE,7,2) +1;

/* save current year and month to highlight today date on display */
CUR_YEAR = SUBSTR(DATE_SAVE,1,4);
/* get month part of date. adding 0 strips the leading zero */
CUR_MONTH = SUBSTR(DATE_SAVE,5,2) +0;

YEAR = CUR_YEAR;/* these variables will change with whats displayed */
MONTH = CUR_MONTH;

IF YEAR // 400  ¬= 0  & YEAR // 4 = 0 THEN /* leap year? */
  NUM_OF_DAYS.2 = 29;
ELSE
  NUM_OF_DAYS.2 = 28;
```

```
/* find 1st weekday of the month. Sun = 0 AND  Sat = 6 */
FIRST_WEEKDAY = (TOT_DAYS+1) // 7;
FIRST_WEEKDAY_SAVE = FIRST_WEEKDAY;

DISP_CENTER_MON = MONTH_NAME.MONTH; /* center display month name */
CENTER_DAYS. = ''; /* null out all unused month days */
/* starting at the first weekday of the month fill in center month */
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.MONTH + FIRST_WEEKDAY ;
  CENTER_DAYS.I = I - FIRST_WEEKDAY;
END;


/* set up to fill in the left side month */
IF MONTH = 1 THEN
  LEFT_MONTH = 12;
ELSE
  LEFT_MONTH = MONTH - 1;

DISP_LEFT_MON = MONTH_NAME.LEFT_MONTH;  /* left display month name */
FIRST_WEEKDAY = (TOT_DAYS - NUM_OF_DAYS.LEFT_MONTH+1) // 7;
LEFT_DAYS. = '';
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.LEFT_MONTH + FIRST_WEEKDAY ;
  LEFT_DAYS.I = I - FIRST_WEEKDAY;
END;


/* set up to fill in the right side month */
FIRST_WEEKDAY = (TOT_DAYS + NUM_OF_DAYS.MONTH +1) // 7;
IF MONTH = 12 THEN
  RIGHT_MONTH = 1;
ELSE
  RIGHT_MONTH = MONTH + 1;
DISP_RIGHT_MON = MONTH_NAME.RIGHT_MONTH; /* right display month name */
RIGHT_DAYS. = '';
DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.RIGHT_MONTH + FIRST_WEEKDAY ;
  RIGHT_DAYS.I = I - FIRST_WEEKDAY;
END;

CUR_DAY_FIELD = 'CENTER_DAYS.'|| (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE)
ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;

 'PANEL SEND CALENDAR' PATH_NAME ATTR_STRING
 'PANEL RECEIVE CALENDAR'


DO FOREVER;
  IF PAN.AID = 'PF3' THEN
    RETURN;

  IF PAN.AID = 'PF7' THEN /* go back one month request */
    DO;
      IF MONTH = 1 THEN  /* always keep track of center month */
        DO;
          MONTH = 12;
          YEAR = YEAR - 1;
          IF YEAR // 400  ¬= 0  &  YEAR // 4  = 0  THEN /* leap year? */
            NUM_OF_DAYS.2 = 29;
          ELSE
            NUM_OF_DAYS.2 = 28;
        END;
```

```
      ELSE
        MONTH = MONTH - 1;
      TOT_DAYS = TOT_DAYS - NUM_OF_DAYS.MONTH;
      DISP_RIGHT_MON = DISP_CENTER_MON;
      DISP_CENTER_MON = DISP_LEFT_MON;
      DO I = 1 TO 37;
        RIGHT_DAYS.I = CENTER_DAYS.I;
        CENTER_DAYS.I = LEFT_DAYS.I;
      END;

      IF MONTH = 1 THEN
        LEFT_MONTH = 12;
      ELSE
        LEFT_MONTH = MONTH - 1;
      FIRST_WEEKDAY = (TOT_DAYS - NUM_OF_DAYS.LEFT_MONTH +1) // 7;
      DISP_LEFT_MON = MONTH_NAME.LEFT_MONTH;
      LEFT_DAYS. = '';
      DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.LEFT_MONTH + FIRST_WEEKDAY ;
        LEFT_DAYS.I = I - FIRST_WEEKDAY;
      END;
    END; /* if pan.aid = 'pf7' */
  ELSE

  IF PAN.AID = 'PF8' THEN /* go forward one month request */
    DO;
      TOT_DAYS = TOT_DAYS + NUM_OF_DAYS.MONTH;
      IF MONTH = 12 THEN  /* always keep track of center month */
        DO;
          MONTH = 1;
          YEAR = YEAR + 1;
          IF YEAR // 400  ¬= 0  &  YEAR // 4  = 0  THEN /* leap year? */
            NUM_OF_DAYS.2 = 29;
          ELSE
            NUM_OF_DAYS.2 = 28;
        END;
      ELSE
        MONTH = MONTH + 1;

      DISP_LEFT_MON = DISP_CENTER_MON;
      DISP_CENTER_MON = DISP_RIGHT_MON;
      DO I = 1 TO 37
        LEFT_DAYS.I = CENTER_DAYS.I; /* shift the months to left    */
        CENTER_DAYS.I = RIGHT_DAYS.I;
      END;

      IF MONTH = 12 THEN  /* need a new right month  */
        RIGHT_MONTH = 1;
      ELSE
        RIGHT_MONTH = MONTH + 1;

      DISP_RIGHT_MON = MONTH_NAME.RIGHT_MONTH;
      FIRST_WEEKDAY = (TOT_DAYS + NUM_OF_DAYS.MONTH +1) // 7;
      RIGHT_DAYS. = '';
      DO I = FIRST_WEEKDAY+1 TO NUM_OF_DAYS.RIGHT_MONTH + FIRST_WEEKDAY;
        RIGHT_DAYS.I = I - FIRST_WEEKDAY;
      END;
    END; /* if pan.aid = 'pf8'  */
```

```
/* see if today's date is in any of the three month being displayed */
/* and set it to red.                                               */

  ATTR_STRING = ''; /* assume current day not on screen */
  IF YEAR = CUR_YEAR THEN
    SELECT;
      WHEN MONTH = CUR_MONTH THEN /* current month in middle */
        DO;
          CUR_DAY_FIELD = 'CENTER_DAYS.'||,
                    (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
          ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
        END;

      WHEN MONTH = CUR_MONTH + 1 THEN /* current month in left display */
        DO;
          CUR_DAY_FIELD = 'LEFT_DAYS.'||,
                    (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
          ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
        END;

      WHEN MONTH = CUR_MONTH - 1 THEN /* current month in right display */
        DO;
          CUR_DAY_FIELD = 'RIGHT_DAYS.'||,
                    (SUBSTR(DATE_SAVE,7,2)+FIRST_WEEKDAY_SAVE);
          ATTR_STRING = 'ATTRIB(' CUR_DAY_FIELD 'RED )' ;
        END;


      OTHERWISE;
    END; /* select */


 'PANEL SEND CALENDAR' PATH_NAME ATTR_STRING
 'PANEL RECEIVE CALENDAR'

END; /* do forever loop */

ERROR:
  SAY 'RETURN CODE ' RC
  SAY 'REA    CODE ' PAN.REA
  SAY 'LOC    CODE ' PAN.LOC
  EXIT;
EXIT_ROUTINE:
  'PANEL END';
  SENDE;
  EXIT;



************ panel definitions ***********************
each definition needs to be in a separate RFS file.
****************************************************
```

Panel Facility

Chapter 25. REXX/CICS Commands

This chapter provides you with detailed reference information for all REXX/CICS commands. Return code information for all commands is returned after command execution in the special REXX variable RC.

You can use all the commands in this chapter with the command environment name REXXCICS. This is also the default. However, depending on how you define the command, you can use a more specific environment name (such as CICS) instead. If you need to reset the command environment, because another command environment is in use, enter: `ADDRESS REXXCICS` before you issue commands from this chapter.

REXX/CICS supports all EXEC CICS commands except: Handle Condition, Handle Aid, Handle Abend, Ignore Condition, Push, and Pop. The syntax for CICS commands under REXX/CICS is documented in the *CICS Transaction Server for VSE/ESA Application Programming Reference*. Mapping between the existing EXEC CICS command definitions to the REXX commands follows:

- CICS, rather than EXEC CICS, should be used as the prefix for CICS commands.

- All data value fields can be specified as a literal character string or as a REXX variable name.

- All data area fields can be specified as a REXX variable name, which is either the source or the target for the desired data.

- The same REXX variable should not be used as both the source and target fields on a CICS command. If this is done, the result of the command execution will be unpredictable.

- Whenever you do not specify a LENGTH option, the length is automatically determined from the length of the related REXX variable or character string.

- When using the CICS ENQ command from REXX/CICS, the LENGTH parameter should be used or unpredictable results may occur.

- NOHANDLE is automatically specified for all CICS commands. The EIBRESP value from the execution of each command is returned in the REXX special variable RC. Also, EIB fields are placed in REXX variables DFHEIBLK, EIBRESP, EIBRESP2, and EIBRCODE.

- For an explanation of the return code values see the *CICS Transaction Server for VSE/ESA Application Programming Reference*. For information on return codes with negative values, see Appendix B, "Return Codes" on page 441.

Example of EXEC CICS to REXX/CICS command mapping:

```
Non-REXX:  EXEC CICS XCTL PROGRAM('PGMA') COMMAREA(COMA) LENGTH(COMAL)
REXX/CICS: "CICS XCTL PROGRAM('PGMA') COMMAREA(COMA)"
```

Note: The EXEC CICS READ, WRITE, and DELETE commands are implemented by default as REXX/CICS authorized commands, to control their use. Refer to G.1, "Authorized REXX/CICS Commands/Authorized Command Options" on page 477 and to Appendix H, "Security" on page 481.

25.1 AUTHUSER

Note: This is an **authorized** command.

```
►►──AUTHUSER──┬──▼──userid──┬──────────────────────────────────►◄
```
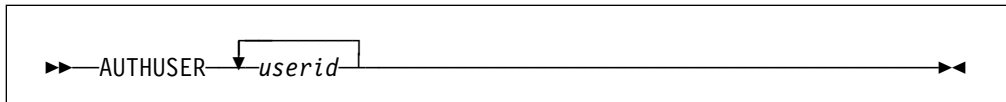
AUTHUSER authorizes a list of user IDs.

25.1.1.1 Operands
userid   is a CICS signon user ID that becomes REXX/CICS authorized.

25.1.1.2 Return Codes
0   Normal return
2602 Invalid operand or operand missing
2621 Specified user ID invalid length
2642 Error storing user ID

25.1.1.3 Example
```
'AUTHUSER USER2 SYSPGMR'
```

This example makes USER2 and SYSPGMR REXX/CICS authorized users.

25.1.1.4 Notes
1. If you are an authorized user you can use REXX/CICS authorized commands, regardless of whether you are running in an exec that was loaded from an authorized REXX/CICS library.

2. If an error is detected in the list of user IDs, the user ID with the error is placed in the REXX special variable, RESULT and processing of the list stops.

3. AUTHUSER commands are cumulative, that is, previous AUTHUSER command definitions remain in effect until the CICS region is recycled.

25.2 CD

```
►►──CD───────────────────────────────────────►◄
        └─dirid─┘
```
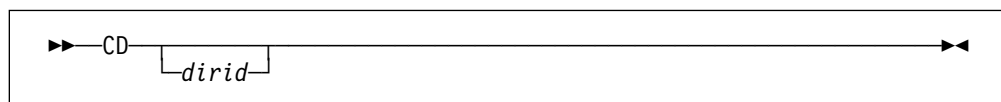
CD changes the RFS file system directory.

25.2.1.1 Operands

dirid   specifies a partial or full REXX File System directory that becomes the new current working directory for you.

If *dirid* is not specified, the current working directory is retrieved and placed in the REXX special variable RESULT, instead of changing the current working directory.

A full directory ID is in the form: *poolid:\dirid1\...\diridn*

When a full directory ID is specified, it completely replaces the previous directory setting.

A partial directory ID does not begin with a *poolid*. In this case, the partial directory ID is appended to the end of the existing directory ID. If the partial directory ID begins with two periods, this indicates that one directory level is removed (from the right), before the new partial directory ID is appended to the end. In this case, a backslash is required before the directory ID.

For example: If the current directory is POOL1:\USERS\USER1\ABC and you enter `CD ..\XYZ` the new current directory will be POOL1:\USERS\USER1\XYZ.

The default directory ID for you is *poolid*:\USERS*userid*\, where *poolid:* is the file pool identifier of the first RFS file pool defined and *userid* is your CICS signon user ID. If you are not signed onto CICS, *userid* defaults to the value in CICS DFLTUSER.

25.2.1.2 Return Codes

0   Normal return
521   Error in retrieving file pool definition
522   Error in creating default RFS directory
523   Error in storing current RFS directory information
524   RFS directory does not exist or access not authorized
525   Error in retrieving directory information
526   Invalid file pool/directory
527   Cannot go back past root directory
528   Error setting result value

25.2.1.3 Examples

```
'CD \USERS\USER2\XYZ'
```

This example changes your current working directory to \USERS\USER2\XYZ in the file pool you are currently using regardless of the previous directory setting.

If your current directory is \USERS\USER2 and you enter:

```
'CD XYZ'
```

then your current directory is changed to \USERS\USER2\XYZ.

25.2.1.4 Note
The CD command works in conjunction with the PATH command identifying the search order for the execution of REXX execs. The current directory (specified by the CD command) is always searched first for execs.

25.3 CEDA

```
►►──CEDA──RDO_Command───────────────────────────────────────►◄
```

Executes a CEDA command for resource definition online (RDO).

25.3.1.1 Operands

RDO_Command

specifies a command string passed as input to the CEDA transaction program.

25.3.1.2 Return Codes

n   specifies the return code passed back by CICS if an error is detected

0   Normal return

-101 Invalid command

Any warning or error messages are placed in the variable CEDATOUT. The results of the execution, if any, are placed in the variable CEDAEOUT. The maximum length returned in CEDAEOUT is approximately 28K bytes. Each variable has the following format:

- Binary halfword containing inclusive length of field.
- Binary halfword containing the number of messages produced.
- Binary halfword containing the highest message-severity: 0 and 4 continue to execution; 8 and 12 do not continue to execution.
- Variable-length data containing:

  – For CEDATOUT: messages produced from translation stage of command

  – For CEDAEOUT: messages produced from execution stage of command

The format of this data is not guaranteed from release to release, but it is the same as that displayed by CEDA.

25.3.1.3 Example

```
'CEDA INSTALL PROGRAM(XYZ) GROUP(ABC)'
```

This example shows a CICS command that is passed to the CEDA transaction program for execution.

25.4 CEMT

```
►►──CEMT──master_term_cmd──────────────────────────────────►◄
```

CEMT executes a CICS master terminal command from REXX.

25.4.1.1 Operands

master_term_cmd
> specifies a command string passed as input to the CEMT transaction program.

25.4.1.2 Return Codes

n   specifies the return code passed back by CICS if an error is detected

0   Normal return

-101 Invalid command

Any warning or error messages are placed in the variable CEMTTOUT. The results of the execution, if any, are placed in the variable CEMTEOUT. The maximum length returned in CEMTEOUT is approximately 28K bytes. Each variable has the following format:

- Binary halfword containing inclusive length of field.
- Binary halfword containing the number of messages produced.
- Binary halfword containing the highest message-severity: 0 and 4 continue to execution; 8 and 12 do not continue to execution.
- Variable-length data containing:

    - For CEMTTOUT: messages produced from translation stage of command

    - For CEMTEOUT: messages produced from execution stage of command

The format of this data is not guaranteed from release to release, but it is the same as that displayed by CEMT.

25.4.1.3 Example

```
'CEMT SET PROGRAM(XYZ) NEWCOPY'
```

This example shows a CICS command passed to the CEMT transaction program for execution.

25.5 CLD

```
►►──CLD──────────────────────────────────────────────►◄
         └─dirid─┘
```
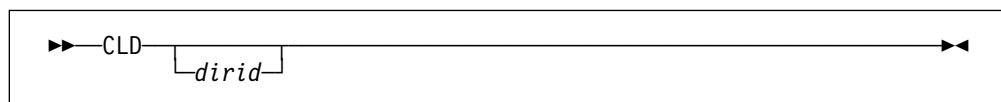
CLD changes your current RLS list directory.

25.5.1.1 Operands
dirid   specifies a partial or full REXX List System directory that becomes the new current working directory for you.

If *dirid* is not specified, the current working directory is retrieved and placed into the REXX variable RESULT, instead of changing the current working directory.

A full directory ID starts with a slash and is in the form: *dirid1*\\...*diridn*

When you specify a full directory ID, it completely replaces the previous directory setting.

A partial directory ID does not begin with a slash. In this case, the partial directory ID is appended to the end of the existing directory ID. If the partial directory ID begins with two periods, this indicates that one directory level is removed (from the right), before the new partial directory ID is appended to the end. In this case, a backslash is required before the directory ID.

For example: If the current directory is \USERS\USER1\ABC and you enter CLD `..\XYZ` the new current directory will be \USERS\USER1\XYZ.

The default directory ID for you is \USERS*genid*\\, where *genid* is your CICS signon user ID. If you are not signed onto CICS, *genid* defaults to the value in DFLTUSER.

25.5.1.2 Return Codes
0   Normal return
923   Error in storing current RLS directory information
924   RLS directory does not exist or access not authorized
925   Error in retrieving directory information
926   Invalid directory
927   Cannot go back past root directory
928   Error setting result value

25.5.1.3 Examples
```
'CLD \USERS\USER2\XYZ'
```

This example changes your current working list directory to \USERS\USER2\XYZ regardless of the previous directory setting.

If your current directory is \USERS\USER2 and you enter:

```
'CLD XYZ'
```

then your current directory is changed to \USERS\USER2\XYZ.

25.5.1.4 Notes

1. The current directory (specified by the CLD command) is always searched first, attempting to locate an RLS list with the appropriate list name.

2. A fully qualified RLS file name bypasses the search of your directories.

25.6 CONVTMAP

```
►►──CONVTMAP──lib.sublib(mem.type)──rfs_fileid────────────────────►◄
```

CONVTMAP reads a VSE Librarian sublibrary member and converts a DSECT (created by a previously assembled BMS map) into a structure, and stores the result in a REXX File System file. The BMS map used as input to CONVTMAP must be in assembler language format. The resulting output file is formatted as a REXX file structure.

25.6.1.1 Operands

lib.sublib(mem.type)
    specifies a VSE Librarian sublibrary member. ˚

rfs_fileid   specifies a fully qualified REXX File system file, or only the REXX file name. If a fully qualified name is not supplied, the curent REXX directory is used to store the file.

25.6.1.2 Return Codes

n   specifies the return code from the attempt to process the VSE Librarian sublibrary member.
0   Normal return
-302 Invalid operand
-321 Invalid input record
-322 RFS error writing output file

25.6.1.3 Example

```
'CONVTMAP REXX.REXC(MAP1) POOL1:\USERS\USER1\MAP1.DATA'
```

This example shows the input BMS map DSECT, MAP1, a member in the VSE Librarian sublibrary REXX.REXC, being formatted and written out to the RFS file POOL1:\USERS\USER1\MAP1.DATA.

25.7 COPYR2S

Note: This is an **authorized** command.

```
►►──COPYR2S─────────────────────────────────────────────────────►

►──┬─source_vname─┬──stor_anchor──┬─offset─────────────────┬──►◄
   └─*────────────┘               │        └─length─┘      │
                                  ├─struct_vname─fieldname──┤
                                  └─struct_name──fieldname──┘
```

COPYR2S copies REXX variable contents to GETMAINed storage.

25.7.1.1 Operands

source_vname

        specifies the REXX variable containing the value copied to the previously GETMAINed area.

        Note: This value should be in quotes so that substitution does not occur.

*   specifies that all the REXX variables are copied. If you specify an asterisk (*) you cannot specify *fieldname*.

stor_anchor

        specifies the REXX variable containing the anchor for the target storage area that was GETMAINed earlier. This anchor consists of four bytes, containing the address of the earlier GETMAINed storage.

offset   specifies the displacement into the previously GETMAINed storage area, that the contents of the REXX variable is copied to. The first byte of the area is indicated by a displacement of zero.

length   specifies the length in decimal bytes of the copy performed. If this length is specified, then the contents of the source REXX variable is truncated, or padded with blanks to match this length, and then copied. However, the source REXX variable is not altered in this process. If this length is omitted, then the current length of the source REXX variable, is used.

struct_vname

        specifies a REXX variable containing a structure definition (or mapping) of the fields in the GETMAINed storage area. The format of the data in this variable is: *field1_name length ... fieldn_name length*. This capability is provided so that field displacements are easily calculated and changed, from a central location.

struct_name

        specifies the structure file ID containing a structure definition (or mapping) of the fields in the GETMAINed storage area.

        Structures are made up of records in the following format: *fieldname location length type*, where:

        fieldname specifies a 1 to 12 character symbolic name of the field.
        location   specifies the position in structure that this field starts (the first position is 1).
        length   specifies decimal length of this field in bytes.

type specifies the field data type: C (character), F (fullword), or H (halfword).

fieldname is a 1 to 12 character symbolic name associated with the destination field for this copy. This name must exist in the above specified REXX variable or structure definition file. The *fieldname* must be specified when *struct_vname* or *struct_name* is specified.

25.7.1.2 Return Codes
0   Normal return
2002 Invalid operand
2021 Invalid structure definition
2022 Invalid variable structure definition
2023 Field name not found
2025 Failure processing GETVAR request
2026 Invalid numeric input
2027 RFS read error
2028 Invalid offset
2029 Invalid length value

25.7.1.3 Examples
```
/* Needed if entering example from the REXXTRY utility */
'PSEUDO OFF'
'CICS GETMAIN SET(WORKANC) LENGTH(200)'/* get 200 bytes of working storage */
VAR1 = '00000000'x    /* set a REXX variable with 4 bytes of hex */
'COPYR2S VAR1 WORKANC 4'
```

This example requests 200 bytes of virtual storage and copies the hex value of '00000000'x into bytes 4 through 7 of that area.

```
'CICS GETMAIN SET(WORKANC) LENGTH(200)'/* get 200 bytes of working storage */
VAR1 = 'ABC'           /* set a REXX variable with 3 characters */
struct1 = 'flda 4 fldb 2 fldc 3 fldd 8 flde 5'
'COPYR2S VAR1 WORKANC STRUCT1 FLDC'
```

This example requests 200 bytes of virtual storage and copies the character string ABC to position 7 of the GETMAINed area referenced by anchor WORKANC.

25.8 COPYS2R

Note: This is an **authorized** command.

```
►►──COPYS2R──stor_anchor──┬─vname─┬──┬─offset──length────────┬──►◄
                          └──*────┘  ├─struct_vname──fieldname─┤
                                     └─struct_name──fieldname──┘
```
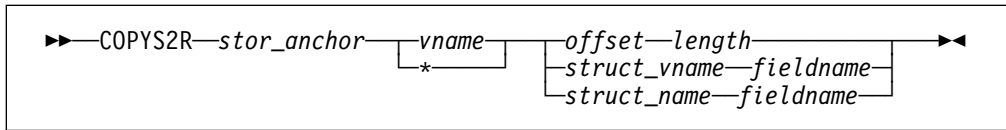
COPYS2R copies data from GETMAINed storage to a REXX variable.

25.8.1.1 Operands

stor_anchor
> specifies the REXX variable containing the anchor for the target storage area that was GETMAINed earlier. This anchor consists of four bytes, containing the address of the earlier GETMAINed storage.
> specifies that all the REXX variables are copied. If you specify an asterisk (*) you cannot specify *fieldname*.

vname specifies the REXX variable containing the value to be copied from the previously GETMAINed area.

> **Note:** This value should be in quotes so that substitution does not occur.

offset specifies the displacement into the previously GETMAINed storage area, that the contents of the REXX variable is copied to. The first byte of the area is indicated by a displacement of zero.

length specifies the length in decimal bytes of the copy performed. If this length is specified, then the contents of the source REXX variable is truncated, or padded with blanks to match this length, and then copied. However, the source REXX variable is not altered in this process. If this length is omitted, then the current length of the source REXX variable, is used.

struct_vname
> specifies a REXX variable containing a structure definition (or mapping) of the fields in the GETMAINed storage area. The format of the data in this variable is: *field1_name length ... fieldn_name length*. This capability is provided so that field displacements are easily calculated and changed, from a central location.

struct_name
> specifies the structure file ID containing a structure definition (or mapping) of the fields in the GETMAINed storage area.
>
> Structures are made up of records in the following format: *fieldname location length type*, where:
>
> *fieldname* specifies a 1 to 12 character symbolic name of the field.
> *location* specifies the position in structure that this field starts (the first position is 1).
> *length* specifies decimal length of this field in bytes.
> *type* specifies the field data type: C (character), F (fullword), or H (halfword).

fieldname is a 1 to 12 character symbolic name associated with the destination field for this copy. This name must exist in the above specified REXX variable or structure definition file. The *fieldname* must be specified when *struct_vname* or *struct_name* is specified.

25.8.1.2 Return Codes

0   Normal return
2102 Invalid operand
2121 Invalid structure definition
2122 Invalid variable structure definition
2123 Field name not found
2125 Failure processing GETVAR request
2126 Invalid numeric input
2127 RFS read error
2128 Invalid offset
2129 Invalid length value

25.8.1.3 Example

```
var1 = '' /* set REXX variable VAR1 to null */
struct1 = 'flda 4 fldb 2 fldc 3 fldd 8 flde 5'
'COPYS2R WORKANC VAR1 STRUCT1 FLDC'
```

This example copies three bytes of data from positions 7 through 9 of the earlier GETMAINed storage area anchored by the fullword address in REXX variable WORKANC, and copies it to REXX variable VAR1.

25.8.1.4 Notes

1. Anchor addresses are not limited to being set by the GETMAIN command. For example, a COPYS2R can be used to copy a fullword address of a GETMAINed area to a REXX variable that is used as the anchor for a subsequent COPYS2R or COPYR2S.

2. There can be multiple structure (field) definitions for a GETMAINed area; they can overlap, be nested, and used to redefine fields.

3. Anchor addresses do not need to point to the beginning of a GETMAINed area. It is important, however, that anchor addresses are within a GETMAINed area that you own, and that any operation on them does not exceed their boundaries.

25.9 C2S

```
►►──C2S──client_rexx_varname───────────────────────────────►◄
                            └─server_rexx_varname─┘
```
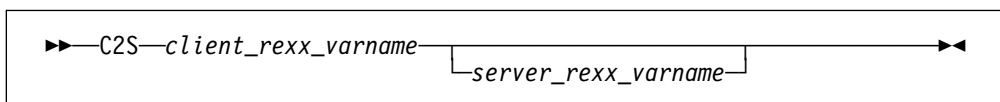
C2S copies a client REXX variable to a server REXX variable.

25.9.1.1 Operands

client_rexx_varname
> specifies the client REXX variable to copy from.

server_rexx_varname
> is an optional name that specifies the server REXX variable to copy into. If it is not specified, it defaults to the same as the *client_rexx_varname*.

25.9.1.2 Return Codes

0   Normal return
2440 No variable name specified
2441 Error retrieving variable
2442 Error storing variable
2448 No client available

25.9.1.3 Example

`'C2S VARA VARB'`

This example shows that the contents of the client REXX variable VARA are copied into the server REXX variable VARB. The length of VARB is the same as the length of VARA.

25.9.1.4 Notes

- The maximum variable name length supported for this command is 250 characters.

- This command is intended for use only by REXX/CICS server execs (for example, exec defined by DEFCMD or DEFSCMD).

25.10 DEFCMD

```
►►──DEFCMD──┬──etarget──ecmdnm──┬──┬──itarget──icmdnm──┬─────────────────────►
            └──etarget──*────────┘  ├──itarget──*───────┤
                                    └──=────=───────────┘

►──┬──pgmname──────┬──(──┬──CICSLINK──┬──┬──────┬──────────────────────────►◄
   ├──CICS_loadmod─┤     ├──CICSLOAD──┤  └─AUTH─┘
   ├──execname─────┤     └──REXX──────┘
   └──(──CLEAR─────┘
```
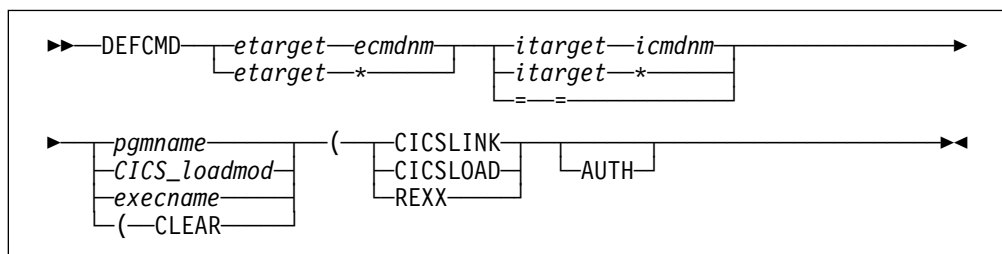
DEFCMD defines (or redefines) REXX user commands.

25.10.1.1 Operands

etarget   is the 1 to 8 character name of the external target environment you used in a REXX exec issuing this command. This is the external environment name that you directed the command string to. This environment name is looked up in a table and together with the command name determines which REXX program the command string is directed to for processing.

 Note: The external target can match the environment name on the ADDRESS keyword instruction or, if REXXCICS is the current environment (the default condition), can be specified as the first token of the command string.

ecmdnm   is the first command name token that you used issuing this command. This is the first word of the command name as it is known to you. If a special value of asterisk (*) is specified (as part of this definition), then all commands that you issued with an environment name of *etarget* and that are not more explicitly defined elsewhere, are covered by this command definition. Command names may be up to 16 characters long.

itarget   specifies an internal environment name that this command definition passes to the agent that processes the command string. This is needed so that the external environment names known to you can be redefined without breakage of the agents that process these commands. If the internal and external names are identical, then there is no need for you to specify the internal name. A special value of "=" indicates that *itarget* is the same as *etarget*.

icmdnm   is the first word of the internal command name. This is the first part of the command name that is passed to the REXX command agent to specify what command is processed. This is specified only if it is different from *ecmdnm*. A special value of "=" indicates that *icmdnm* is the same as *ecmdnm*.

pgmname specifies the CICS program that is called by an EXEC CICS LINK to process the command.

CICS_loadmod
          specifies the name of the CICS program called because the CICSLOAD option was specified.

 Note: The program is only loaded on the first instance of a command and its address is remembered for subsequent commands.

execname

specifies the exec called as a REXX command server processing this command (or commands). If this server exec is already running then this command is routed to the executing server. If a REXX server by this name is not running, then Automatic Server Initiation (ASI) is used to start the server automatically. The *execname* can be either a file name (where the file type defaults to EXEC) or it can be in the form *filename.filetype*.

CICSLINK

is a keyword indicating that the processing agent for the defined REXX command is a standard CICS program that is called by an EXEC CICS LINK.

CICSLOAD

is a keyword indicating that the processing agent is a CICS program that is loaded by an EXEC CICSLOAD.

REXX   is a keyword indicating that the processing agent for this REXX command is a REXX exec that operates as a command server.

AUTH   **Note:** This is an **authorized** option.

is a keyword indicating that this is an authorized REXX/CICS command. It is a command that can only be executed by an authorized REXX/CICS user (specified on AUTHUSER command) or from within an exec loaded from an authorized library.

CLEAR   is a keyword indicating that the purpose of this DEFCMD is to clear any previous definitions for the specified external target environment and command names.

25.10.1.2 Return Codes

0   Normal return
1001 Invalid command
1021 Cannot load program
1023 Entry not found
1048 No client available
1099 Internal error

25.10.1.3 Example

```
'DEFCMD CICS SEND = = SENDPGM (CICSLINK'
```

This example defines a command called SEND for this user only. The user can issue this command, under the default command environment of REXXCICS, by entering:

```
'CICS SEND arg1 arg2 ... argn'
```

This example shows program SENDPGM being called by an EXEC CICS LINK command to process this command.

25.10.1.4 Notes

1. When the REXX/CICS environment name is REXXCICS (which is the default when all execs or macros are called), then the first token of the command string is the environment name that could have been used with an ADDRESS environment REXX instruction. This provides a more integrated command environment and removes the need for constant environment switching by ADDRESS instructions.

2. The calling and parameter passing sequences for command programs receiving control by an EXEC CICS LINK and an Assembler BASSM instruction (the CICSLOAD option) are similar. Refer to Chapter 21, "REXX/CICS Command Definition" on page 325 for more information on writing command programs.

3. You can use DEFCMD to dynamically tailor a user's command set on a user by user, or application by application basis. DEFCMD commands can be placed in the user's PROFILE EXEC or in application execs. DEFCMD can also be used to override system command definitions.

4. `DEFCMD REXXCICS *` is not allowed.

5. User command definitions are searched before system command definitions (except for DEFCMD, which cannot be overridden).

6. REXX commands can be written in REXX. These REXX commands in turn call other REXX commands which are written in REXX, in a building block fashion. Since DEFCMD hides the implementation detail from the REXX user (programmer), a command can be quickly written in REXX and later transparently rewritten in another language, if it becomes performance critical.

25.11 DEFSCMD

Note: This is an **authorized** command.

```
►►──DEFSCMD──┬──etarget──ecmdnm──┬──┬──itarget──icmdnm──┬──────────►
             ├──etarget──*────────┤  ├──itarget──*───────┤
             └──*──*──────────────┘  └──=──=─────────────┘

  ►──┬──pgmname──────┬──(──┬──CICSLINK──┬────────────────────►◄
     ├──CICS_loadmod─┤     ├──CICSLOAD──┤  ┌──AUTH──┐
     ├──execname─────┤     └──REXX──────┘
     └──(──CLEAR─────┘
```
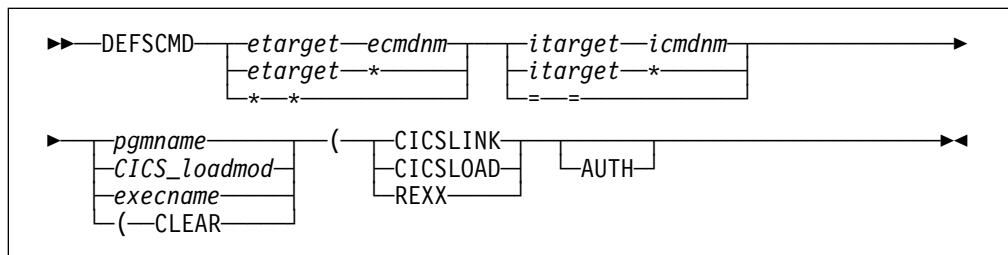
DEFSCMD defines (or redefines) REXX system commands.

25.11.1.1 Operands

etarget   is the name of the external target environment you used in a REXX exec issuing this command. This is the external environment name that you directed the command string to. This environment name is looked up in a table and together with the command name determines which program, REXX exec, or queue this command string is directed to for processing.

Note: The external target can match the environment name on the ADDRESS keyword instruction or can be specified as the first token of the command string, if REXXCICS is the current environment (which is the default).

ecmdnm   is the first command name token that you used issuing this command. This is the first word of the command name as it is known to you. If a special value of asterisk (*) is specified (as part of this definition), then all commands that you issued with an environment name of *etarget* and that are not more explicitly defined elsewhere, are covered by this command definition.

itarget   specifies an internal environment name that this command definition passes to the agent that processes the command string. This is needed so that the external environment names known to you can be redefined without breakage of the agents that process these commands. If the internal and external names are identical, then there is no need for you to specify the internal name. A special value of "=" indicates that *itarget* is the same as *etarget*.

icmdnm   is the first word of the internal command name. This is the first part of the command name that is passed to the REXX command agent to specify what command is processed. This is specified only if it is different from *ecmdnm*. A special value of "=" indicates that *icmdnm* is the same as *ecmdnm*.

pgmname specifies the CICS program that is called by an EXEC CICS LINK to process the command.

CICS_loadmod
             specifies the name of the CICS program called because the CICSLOAD option was specified.

> **Note:** The program is only loaded on the first instance of a command and its address is remembered for subsequent commands.

execname
specifies the exec called as a REXX command server processing this command (or commands). If this server exec is already running then this command is routed to the executing server. If a REXX server by this name is not running, then Automatic Server Initiation (ASI) is used to start the server automatically. The *execname* can be either a file name (where the file type defaults to EXEC) or it can be in the form *filename.filetype*.

CICSLINK
is a keyword indicating that the processing agent for the defined REXX command is a standard CICS program that is called by an EXEC CICS LINK.

CICSLOAD
is a keyword indicating that the processing agent is a CICS program that is loaded by an EXEC CICSLOAD.

REXX
is a keyword indicating that the processing agent for this REXX command is a REXX exec that operates as a command server.

AUTH
is a keyword indicating that this is an authorized REXX/CICS command. It is a command that can only be executed by an authorized REXX/CICS user (specified on AUTHUSER command) or from within an exec loaded from an authorized library.

CLEAR
is a keyword indicating that the purpose of this DEFSCMD is to clear any previous definitions for the specified external target environment and command names.

25.11.1.2 Return Codes
0   Normal return
1101 Invalid command
1121 Cannot load program
1123 Entry not found
1148 No client available
1199 Internal error

25.11.1.3 Example
```
'DEFSCMD CICS SEND = = SENDPGM (CICSLINK'
```

This example defines a command called SEND for this user only. The user can issue this command, under the default command environment of REXXCICS, by entering:

```
'CICS SEND arg1 arg2 ... argn'
```

This example shows program SENDPGM being called by an EXEC CICS LINK command to process this command.

25.11.1.4 Notes

1. When the REXX/CICS environment name is REXXCICS (which is the default when all execs or macros are called), then the first token of the command string is the environment name that could have been used with an ADDRESS environment REXX instruction. This provides a more integrated command environment and removes the need for constant environment switching by ADDRESS instructions.

2. The calling and parameter passing sequences for command programs receiving control by an EXEC CICS LINK and an Assembler BASSM instruction (the CICSLOAD option) are similar. Refer to Chapter 21, "REXX/CICS Command Definition" on page 325 for more information on writing command programs.

3. If the first two operands of DEFSCMD are all asterisks (* *) then this is a catch-all definition that specifies a command processing agent issued for REXX commands that are not under the scope (do not match) of any more specific command definitions.

4. User command definitions are searched before system command definitions (except for DEFSCMD, which cannot be overridden).

5. REXX commands can be written in REXX. These REXX commands in turn call other REXX commands which are written in REXX, in a building block fashion. Since DEFSCMD hides the implementation detail from the REXX user (programmer), a command can be quickly written in REXX and later transparently rewritten in another language, if it becomes performance critical.

25.12 DEFTRNID

Note: This is an **authorized** command.

```
►►──DEFTRNID──trnid──execname──────────────────────────►◄
                              └─CLEAR─┘
```

DEFTRNID is a region-wide authorized command that can be used to define the name of an exec to be invoked for a particular CICS transaction identifier.

25.12.1.1 Operands

trnid   specifies a one to four character CICS transaction ID.

execname
               specifies a 1 to 17 character REXX/CICS exec name, in the form: *filename.filetype* if it is in the REXX File System. If the exec exists in a VSE Librarian sublibrary, this is the name (a member type of *.proc* is assumed).

CLEAR   is a keyword indicating that the definition be removed for this transaction ID.

25.12.1.2 Return Codes

0   Normal return
1202 Invalid operand
1222 Invalid option
1223 Error storing trantable information
1225 Error retrieving trantable information
1226 Exec name length error
1228 Error setting trantable value
1233 Transaction not found in table

25.12.1.3 Example

Define a new CICS transaction ID named XYZ, make it call exec TESTEXEC when it is started, then take the following steps:

  1. Create TESTEXEC in the RFS or in a VSE Librarian sublibrary.

  2. Under REXX/CICS REXXTRY utility enter command DEFTRNID XYZ TESTEXEC and exit the REXXTRY utility.

  3. Under RDO, CEDA DEFINE TRAN(XYZ) PROGRAM(CICREXD) TWASIZE(32) GROUP(REXXCICS) and install TRAN(XYZ) GROUP(REXXCICS).

     Note: TWASIZE is required on CEDA define

  4. Clear the screen and type CICS transaction identifier XYZ and press ENTER. The TESTEXEC exec should now run.
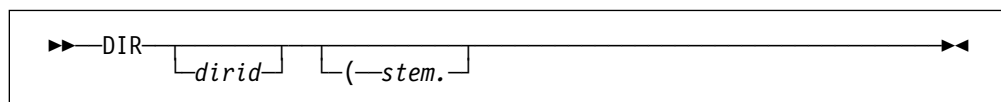
25.12.1.4 Notes

1. DEFTRNID definitions should usually be placed in the CICSTART exec that executes at REXX/CICS startup.

2. The transaction ID has to be defined (by CEDA) to invoke the supplied CICREXD program.

25.13 DIR

```
►►──DIR───────────────────────────────────────────────────►◄
          └─dirid─┘   └─(──stem.─┘
```

DIR displays the current directory contents or optionally returns the directory contents in a REXX compound variable.

25.13.1.1 Operands

dirid   specifies the partial or full REXX File System directory that is displayed. If you omit this, then the current directory is displayed.

stem.   specifies the name of a stem. (A stem must end in a period.) Stem.0 contains the number of elements in the entry. (Refer to section 12.4.4, "Stems" on page 145 for more information.) If you omit this, the contents are displayed on the screen.

25.13.1.2 Return Codes

0   Normal return
321   Cannot access current RFS directory information
322   Invalid stem name
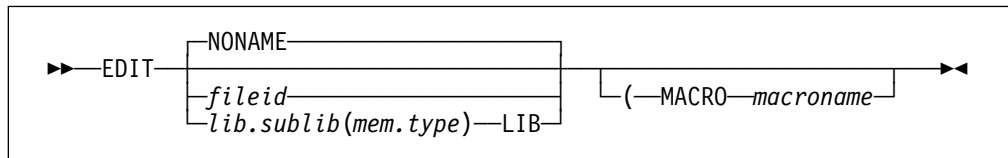325   Error retrieving RFS directory

25.13.1.3 Examples

```
'DIR \USERS\USER2 (X.'
```

This example places the directory contents of \USERS\USER2 into the REXX compound variable X.1 through X.n. X.0 contains the number of elements returned.

25.13.1.4 Note

The current directory is specified by the CD command.

25.14 EDIT

```
►►──EDIT─┬─────NONAME──────┬─────────────────────────────┬──►◄
         ├──fileid─────────┤    └─(──MACRO──macroname─┘
         └──lib.sublib(mem.type)──LIB─┘
```

EDIT opens a new edit session.

25.14.1.1 Operands

NONAME a file ID is not specified. This is the default.

fileid   specifies the file ID of the file to be created or edited.

lib.sublib(mem.type)
              specifies a VSE Librarian sublibrary and member to be edited.

LIB   is a keyword that follows a VSE Librarian sublibrary member name
              when a sublibrary member is being edited.

MACRO   is a keyword specifying a group of instructions applied to the file being
              edited.

macroname
              specifies the file name portion of the profile macro file ID (REXX exec
              name).

25.14.1.2 Return Codes

0   Normal return
201   Invalid command
211   Invalid file ID
226   File is currently being edited
299   Internal error

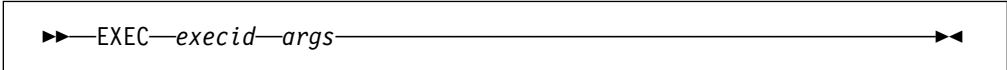### 25.14.1.3 Example

```
'CD \USERS\USER2\' /* specify current working directory */
'EDIT TEST.EXEC'   /* edit an existing file in the PATH */
                   /* or create new file in current dir */
```

This example edits member TEST.EXEC in directory \USERS\USER2.

25.14.1.4 Note

Refer to Chapter 18, "REXX/CICS Text Editor" on page 271 for more information
about the REXX/CICS editor.

25.15 EXEC

```
►►──EXEC──execid──args─────────────────────────────────────────►◄
```

EXEC calls a REXX exec at a lower level (as a nested exec). All variables for this new exec are kept separate from the higher level exec, which is suspended until the nested exec ends.

25.15.1.1 Operands

execid   specifies the 1 to 17 character identifier of the exec.

args   specifies the argument string being passed to the called exec.

25.15.1.2 Return Codes

n   specifies the return code set by the exit of the called exec
0   Normal return
-3   Exec not found
-10   Exec name not specified
-11   Invalid exec name
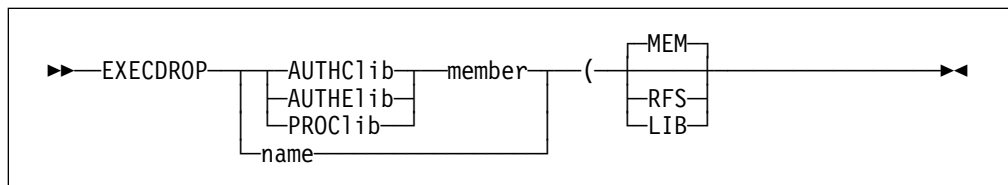-12   GETMAIN error
-99   Internal error

25.15.1.3 Example

```
'EXEC ABC'
```

This example executes exec ABC.EXEC.

25.16 EXECDROP

Note: This is an **authorized** command.

```
►►──EXECDROP──┬─AUTHClib─┬──member──(──┬──MEM──┬──────────►◄
              ├─AUTHElib─┤             ├──RFS──┤
              ├─PROClib──┤             └──LIB──┘
              └─name─────────────┘
```

EXECDROP removes an EXECLOADed exec from virtual storage.

25.16.1.1 Operands
AUTHClib

> indicates that *member* was loaded from an authorized command sublibrary.

AUTHElib

> indicates that *member* was loaded from an authorized exec sublibrary.

LIB   indicates that a VSE Librarian sublibrary and member has been specified.

member   specifies the VSE Librarian sublibrary member that contains the exec to be removed from virtual storage.

> **Note:** The member type must not be specified. A member type of PROC is assumed.

MEM   indicates that a member name has been specified.

name   specifies a fully qualified RFS file name or a VSE Librarian sublibrary and member.

> **Note:** The member type must not be specified for a sublibrary and member. A member type of PROC is assumed.

PROClib indicates that *member* was loaded from a sublibrary in the LIBDEF PROC search chain for the CICS partition.

RFS   indicates that an RFS file has been specified.

25.16.1.2 Return Codes
0   Normal return
1401 Invalid command
1402 Invalid operand
1423 Error storing EXECLOAD information
1425 Error retrieving EXECLOAD information
1448 No client available
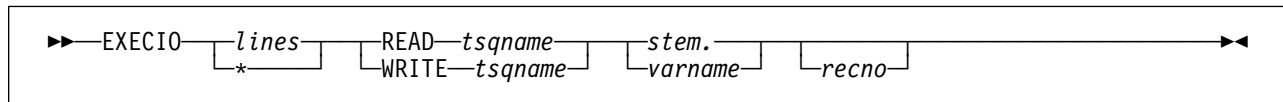
25.16.1.3 Example
```
'EXECDROP POOL1:\USERS\USER2\TEST.EXEC (RFS'
```

This example removes RFS file from virtual storage.

25.16.1.4 Note
If a partial directory ID is given, it is temporarily appended to the end of the current working directory value to get a fully qualified directory ID.

25.17 EXECIO

```
►►──EXECIO──┬─lines─┬──┬─READ──tsqname─┬──┬─stem.───┬──┬───────┬──►◄
            └─*─────┘  └─WRITE─tsqname─┘  └─varname─┘  └─recno─┘
```

EXECIO performs file input/output to a CICS temporary storage queue.

25.17.1.1 Operands

lines   specifies the number of lines to read or write. An asterisk (*) is a
            special case that is specified for READ operations only, and indicates
            that the file is read from the target line (or line 1 if no target line is
            specified) to the end of the file.

READ   reads one or more records from a CICS temporary storage queue
            (TSQ).

WRITE   writes (or re-writes) one or more records to a CICS temporary storage
            queue.

tsqname   specifies a 1 to 8 character temporary storage queue name.

stem.   specifies the name of a stem. (A stem must end in a period.) Refer to
            section 12.4.4, "Stems" on page 145 for more information.

varname   specifies a REXX variable name that is the source or target for this
            EXECIO operation.

recno   specifies a record number in the temporary storage queue that READ or
            WRITE begins with.

25.17.1.2 Return Codes

n   specifies the return code passed back by CICS if an error is detected
0   Normal return
-202 Invalid operand
-221 Too many operands specified
-222 Recno operand out of range
-224 Lines operand invalid

25.17.1.3 Examples

```
x.1 = 'line 1'
x.2 = 'Line Two'
'EXECIO 2 WRITE QUEUE1 X.'
```

This example writes data to a CICS temporary storage queue.

```
'EXECIO 2 READ QUEUE1 Y.'
say y.0  /* ==>  2          */
say y.1  /* ==>  'line 1'   */
say y.2  /* ==>  'Line Two' */
```

This example reads data from a temporary storage queue.

25.17.1.4 Notes

1. The maximum record length allowed is 256 bytes.

2. If a *stem* is specified for a READ operation (and a *stem* should be specified if more than one record is read), the actual number of records read is placed into stem.0.

3. Use the CICS-supplied CEBR transaction to browse temporary storage queues. For example, enter: `CEBR QUEUE1` to look at the queue created above.

4. CEBR provides PUT and GET functions that you can use to copy between external files and CICS temporary storage queues.

25.18 EXECLOAD

Note: This is an **authorized** command.

```
►►──EXECLOAD──┬──AUTHClib──┬──member──┬──(──┬──MEM──┬────────►◄
              ├──AUTHElib──┤          │     ├──RFS──┤
              ├──PROClib───┤          │     └──LIB──┘
              └──name─────────────────┘
```
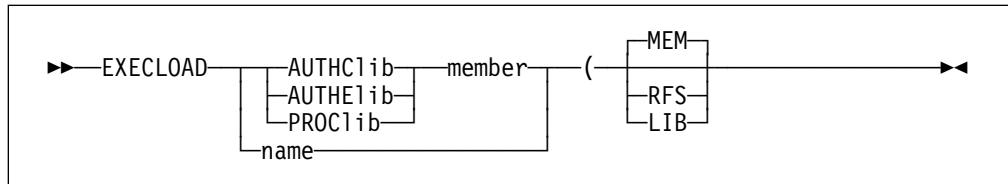
EXECLOAD loads an exec into virtual storage.

25.18.1.1 Operands

AUTHClib
>indicates that *member* must be loaded from an authorized command sublibrary.

AUTHElib
>indicates that *member* must be loaded from an authorized exec sublibrary.

LIB   indicates that a VSE Librarian sublibrary and member has been specified.

MEM   indicates that a member name has been specified.

member   specifies the VSE Librarian sublibrary member that contains the exec to be loaded into virtual storage.

>**Note:** The member type must not be specified. A member type of PROC is assumed.

name   specifies a fully qualified RFS file name or a VSE Librarian sublibrary and member.

>**Note:** For a sublibrary and member, the member type must not be specified. A member type of PROC is assumed.

PROClib indicates that *member* must be loaded from a sublibrary in the LIBDEF PROC search chain for the CICS partition.

RFS   indicates that an RFS file has been specified.

25.18.1.2 Return Codes

0   Normal return
1501 Invalid command
1502 Invalid operand
1523 Error storing EXECLOAD information
1525 Error retrieving EXECLOAD information
1530 Unable to link to CICLIBR routine
1531 Error returned from CICLIBR routine
1532 Error returned from RFS READ
1547 GETMAIN error
1548 No client available
1599 Internal error

25.18.1.3 Example
`'EXECLOAD POOL1:\USERS\USER2\TEST.EXEC (RFS'`

This example loads the exec TEST.EXEC from RFS into storage. Subsequent calls of TEXT.EXEC will use the loaded copy.

25.18.1.4 Notes
1. If an exec is loaded into virtual storage, it is automatically shared by all users.

2. If an EXECLOAD is performed to replace an in-storage exec with a newer copy, a shadow of previous copies is kept in virtual storage until all active execs based on these copies end. This is accomplished by a use count.

25.19 EXECMAP

```
►►──EXECMAP──────────────────────────────────────────────────►◄
```
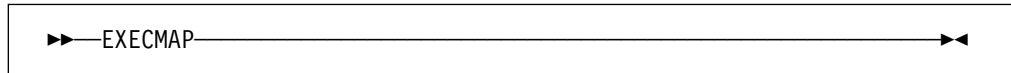
EXECMAP returns the sublibraries and members, the number of users, the descriptor table start (in hex), and the amount of storage required of the execs that have been loaded using EXECLOAD.

25.19.1.1 Return Codes
0   Normal return
1623 EXECLOAD directory not found

25.19.1.2 Example
```
'EXECMAP'
```

If the exec POOL1:\USERS\USER1\TEST.EXEC had been EXECLOADed then this display would result.

```
EXEC name   Use     Location    Size     Fully Qualified Name
TEST.EXEC    0      09369083     287     POOL1:\USERS\USER1\TEST.EXEC
```

25.20 EXPORT

```
►►──EXPORT──rfs_fileid──lib.sublib(mem.type)────────────────────────►◄
```

EXPORT exports an RFS file to a VSE Librarian sublibrary member.

25.20.1.1 Operands
rfs_fileid   specifies a fully qualified REXX File System file ID.

lib.sublib(mem.type)
        specifies a VSE Librarian sublibrary and member.

25.20.1.2 Return Codes
0   Normal return
1701 Invalid command
1702 Invalid operand
1723 RFS write error
1724 RFS read error
1733 Input for export not found
1736 Unexpected CICS error
1738 Invalid VSE Librarian member name
1741 Unsupported record format
1799 Internal error

25.20.1.3 Example
'EXPORT POOL1:\USERS\USER1\TEST.DATA USER1.TEST(MEM1.PROC)
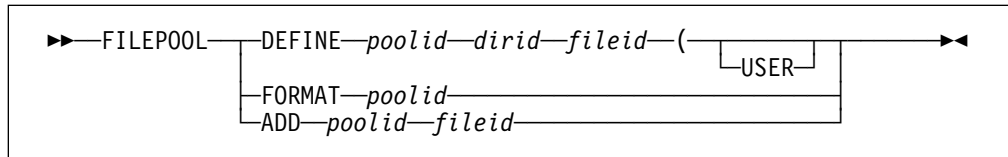
This example copies POOL1:\USERS\USER1\TEST.DATA from an RFS file to a
VSE Librarian sublibrary and member.

25.20.1.4 Notes
The sublibrary specified in *lib.sublib* must exist. If the member specified in
lib.sublib(mem.type) exits, it is replaced by the contents of the RFS file
specified in *rfs_fileid*. If the member does not exist, it is created in the
specified sublibrary.

25.21 FILEPOOL

Note: This is an **authorized** command.

```
►►──FILEPOOL───┬─DEFINE──poolid──dirid──fileid──(─┬──────────┬─────────►◄
               │                                  └──USER──┘
               ├─FORMAT──poolid──────────────────
               └─ADD──poolid──fileid─────────────
```

FILEPOOL performs RFS file pool administration activities.

25.21.1.1 Operands
DEFINE   defines a new RFS file pool.

poolid   specifies the name of the target file pool.

dirid   specifies the CICS file identifier of the file pool directory.

fileid   specifies the CICS file identifier of the VSAM file for the file pool.

USER   is an optional keyword indicating that this is a user file pool, and you automatically have a \USERS directory created, so that multiple users share this file pool, using RFS security (as compared to CICS security) to control access to directories.

FORMAT formats the first file in a new RFS file pool.

ADD   adds an additional VSAM file to an existing file pool.

25.21.1.2 Return Codes
0   Normal return
1802 Invalid operand
1821 Invalid file pool subcommand
1822 File pool subcommand not specified
1823 Error storing file pool information
1824 File pool ID not specified
1825 Error retrieving file pool information
1826 Invalid file pool ID
1827 Invalid file pool data retrieved
1828 File pool not defined
1829 RFS could not add library to file pool
1830 RFS could not create users directory
1831 CICS file identifier for the file pool must be specified
1832 Invalid CICS file identifier
1833 File pool variable corrupted
1834 Pool ID already exists
1835 CICS file identifier already used
1836 Could not format file pool
1837 File pool needs to be formatted first
1838 File pool ADD record is full
1839 File ID is not found

25.21.1.3 Example.

```
'FILEPOOL DEFINE POOL1 REXXDIR1 REXXLIB1 (USER'
```

This example defines file pool POOL1 and tells RFS the CICS file definition to use is REXXLIB1. It also indicates to the FILEPOOL FORMAT command to issue an RFS MKDIR to build the \USERS directory.

25.21.1.4 Note

This is an authorized command, performed by a REXX/CICS administrator or systems programmer.

25.22 FLST

```
▶▶──FLST──────────────────────────────────────────────▶◀
          └─dirid─┘
```
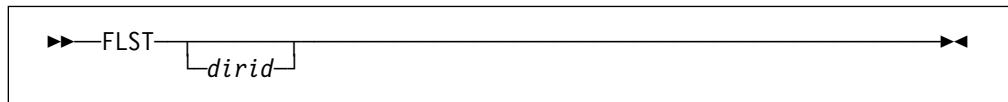
FLST calls the file list utility to work with the files.

25.22.1.1 Operands

dirid   specifies an optional full or partial directory ID that a file list is displayed. If you do not specify *dirid*, it defaults to the current working directory.

25.22.1.2 Return Codes

FLST returns the return code given by RFS.

25.22.1.3 Example

`'FLST'`

This example displays the file list for the member of the current working directory.

25.22.1.4 Notes

1. The default user profile macro that the FLST tries to call is EXEC ID FLSTPROF. The FLSTPROF macro provides the capability for you (or a group of users) to specify your own unique defaults.

2. Refer to Chapter 19, "REXX/CICS File System" on page 299 for more information about the REXX File System.

25.23 GETVERS

```
►►──GETVERS──────────────────────────────────────────────────────►◄
```

GETVERS retrieves the current REXX/CICS, program name, version, and compile time information, and places it into the REXX variable VERSION. The returned information is in the form: V*x*R*y*M*mmmm mm/dd/yy hh.mm*, where:

x   specifies the REXX/CICS Version number.

y   specifies the REXX/CICS Release number.

mm/dd/yy   specifies the compile date for the REXX/CICS base program.

hh.mm   specifies the compile time for the REXX/CICS base program.

25.23.1.1 Return Codes
0   Normal return
1910 Request failed

25.23.1.2 Example
`'GETVERS'`

This example retrieves the current REXX/CICS version and compile time information and places it into the REXX variable VERSION. Its format could look like: `V1R1M0000 03/07/94 12:00`.

25.24 HELP

```
►►──HELP────────────────────────────────────────────────►◄
          └─search_term─┘
```
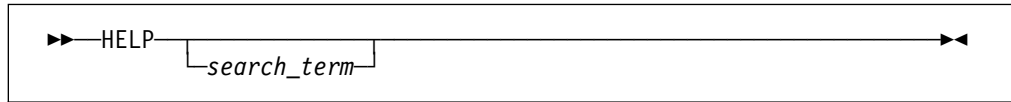
HELP browses or searches this book (the *IBM REXX Development System for CICS/ TS for VSE/ESA*) online.

25.24.1.1 Operands
search_term
>   specifies the string you want located.

25.24.1.2 Return Codes
n   specifies the return code passed by from internal RFS or PANEL commands
0   Normal return

25.25 IMPORT

```
►►──IMPORT──lib.sublib(mem.type)──rfs_fileid──────────────────────►◄
```

IMPORT imports a VSE Librarian sublibrary member to an RFS file.

25.25.1.1 Operands

lib.sublib(mem.type)
>   specifies a VSE Librarian sublibrary and member.

rfs_fileid   specifies a fully qualified REXX File System file ID.

25.25.1.2 Return Codes

0   Normal return
1701 Invalid command
1702 Invalid operand
1723 RFS write error
1724 RFS read error
1733 Input for export not found
1736 Unexpected CICS error
1738 Invalid VSE Librarian member name
1741 Unsupported record format
1799 Internal error

25.25.1.3 Example

`'IMPORT USER1.TEST(MEM1.PROC) POOL1:\USERS\USER1\TEST.DATA'`

This example copies a VSE Librarian sublibrary member
USER1.TEST(MEM1.PROC) to the RFS file POOL1:\USERS\USER1\TEST.DATA.

25.26 LISTCMD

```
►►──LISTCMD──────────────┬──────────┬──┬──────────┬───────────────────►◄
                         └─envname──┘  └─cmdname──┘
```

LISTCMD lists REXX command definition information (previously specified by DEFCMD).

25.26.1.1 Operands

envname specifies the name of the command environment defined using DEFCMD or DEFSCMD.

cmdname specifies the name of a command specified in DEFCMD or DEFSCMD.

25.26.1.2 Return Codes

0   Normal return
821 Invalid environment name
822 Invalid command name

25.26.1.3 Example

`'LISTCMD EDITSVR MYCMD'`

This example returns the information defined in DEFCMD for the subcommand MYCMD.

25.27 LISTCLIB

```
►►──LISTCLIB──────────────────────────────────────────────►◄
              └─stem.─┘
```

LISTCLIB displays the names of the authorized command libraries to the terminal
or to a specified stem array, if a stem has been specified. The libraries are
displayed in their search order.

25.27.1.1 Operands
stem.   specifies the name of a stem. (A stem must end in a period.) The
             information returned is the name of each VSE Librarian sublibrary
             specified on the last SETSYS AUTHCLIB command.

25.27.1.2 Return Codes
2226 Invalid stem varible name
2245 Error retrieving authorized library information

25.27.1.3 Example
```
'LISTCLIB LST.'
```

This example places the names of the VSE Librarian sublibraries containing
authorized commands in compound variable LST.1 through LST.n. They are
returned in the search order specified on the last SETSYS AUTHCLIB command.
LST.0 contains the number of elements returned.

25.28 LISTELIB

```
►►──LISTELIB──────────────────────────────────────►◄
            └─stem.─┘
```

LISTELIB displays the names of the authorized exec libraries to the terminal or to a specified stem array, if a stem has been specified. The libraries are displayed in their search order.

25.28.1.1 Operands
stem.   specifies the name of a stem. (A stem must end in a period.) The information returned is the name of each VSE Librarian sublibrary specified on the last SETSYS AUTHELIB command.

25.28.1.2 Return Codes
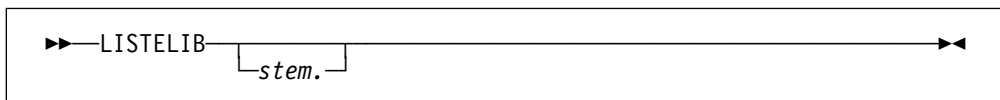2226 Invalid stem varible name
2245 Error retrieving authorized library information

25.28.1.3 Example
```
'LISTELIB LST.'
```

This example places the names of the VSE Librarian sublibraries containing authorized commands in compound variable LST.1 through LST.n. They are returned in the search order specified on the last SETSYS AUTHELIB command. LST.0 contains the number of elements returned.

25.29 LISTPOOL

```
►►──LISTPOOL──────────────────────────────────────────────►◄
              └─stem.─┘
```
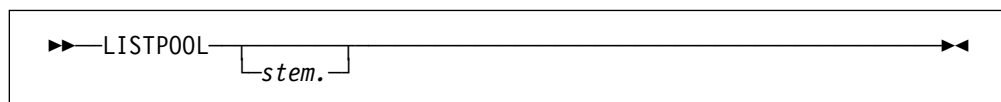
LISTPOOL displays RFS file pool information to the terminal or to a specified stem array, if a stem has been specified.

25.29.1.1 Operands
stem.   specifies the name of a stem. (A stem must end in a period.) Refer to section 12.4.4, "Stems" on page 145 for more information. Three columns of information are returned:

- the pool ID
- the DLBL of the first file in the file pool
- whether or not it contains a user's directory.

25.29.1.2 Return Codes
0   Normal return
2225 Error retrieving file pool information
2226 Invalid stem variable name

25.29.1.3 Example
`'LISTPOOL LST.'`

This example places information about the RFS file pools into REXX compound variable `LST.1` through `LST.n`. `LST.0` contains the number of elements returned.

25.29.1.4 Note
This is a general user command that displays all defined RFS file pools.

25.30 **LISTTRNID**

Note: This is an **authorized** command.

```
►►──LISTTRNID──────────────────────────────────────────────────►◄
```
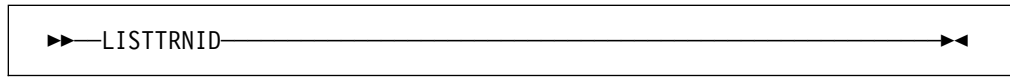
LISTTRNID lists the current transaction ID definitions created by the DEFTRNID command.

25.30.1.1 **Return Codes**
0   Normal return
2325 Error retrieving trantable information

25.30.1.2 **Example**
'LISTTRNID'

The CICSTART exec defines the default transactions and their EXEC names. The resulting display is:

```
TRNID           EXEC name
REXX            CICRXTRY
EDIT            CICEDIT
FLST            CICFLST
End of Transaction table list.
```

25.31 PATH

```
►►──PATH─┬──────────────┬──────────────────────►◄
         │  ┌──────────┐ │
         └──┼──dirid────┼─┘
            └──lib.sublib──┘
```
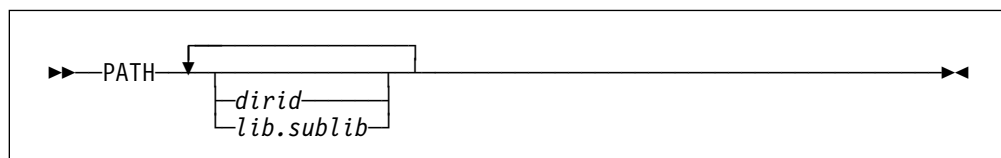
PATH defines the search path for REXX execs.

25.31.1.1 Operands

dirid   specifies one or more fully qualified REXX File System directories that are searched when you are attempting to locate an exec to be executed.

A full RFS directory ID starts with a pool ID and is in the form: POOL1:*dirid1*\\...*diridn*

When more than one directory ID is specified, a blank is used to separate them.

lib.sublib   specifies one or more VSE Librarian sublibraries.

25.31.1.2 Return Codes

0   Normal return
625   Error retrieving path information
626   Invalid RFS directory name
628   Error setting RESULT value
629   Invalid VSE Librarian sublibrary
630   Error storing path information
631   No path currently defined
632   Resulting PATH contains no RFS directories or library names

25.31.1.3 Examples

```
'PATH POOL1:\USERS\USER2 POOL2:\USERS\USER2\PROJECT1'
```

This example shows that the directories in this list are searched in the order specified (from left to right).

```
'PATH REXX1.EXECS REXX2.EXECS'
```

This example shows you that the search is started with the first VSE Librarian sublibrary.

25.31.1.4 Notes

1. The *dirid*s and libraries can be intermixed in a single PATH statement.

2. A very long PATH directory list can be created by concatenating strings together into a variable, and then specifying this variable on the PATH command.

3. PATH command definitions are not carried across CICS restarts. To permanently change a PATH definition, insert a PATH command into your PROFILE exec in the base directory for your user ID.

4. The PATH command is not cumulative, that is, the last PATH command replaces the previous PATH definition.

5. If *dirid* or the library is not specified, the users current working path is retrieved and placed in the REXX special variable RESULT.

6. If you receive a non-zero return code, the contents of the RESULT special variable are unpredictable.

25.32 PSEUDO

```
►►──PSEUDO──┬──ON──┬──────────────────────────────────────────────►◄
            └──OFF──┘
```
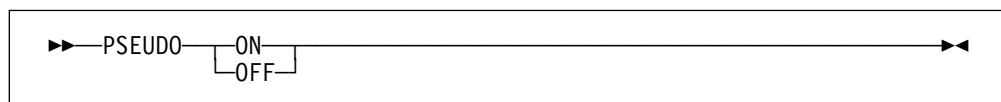
PSEUDO turns the pseudo-conversational mode on or off.

25.32.1.1 Operands
ON   enables automatic pseudo-conversational support so that when the next
            REXX PULL instruction or REXX/CICS WAITREAD command is
            encountered in the current exec, instead of a conversational terminal
            read occurring immediately, an EXEC CICS RETURN TRANSID is used
            to suspend the exec until terminal input occurs, and then the terminal
            read occurs.

OFF   disables (turns off) automatic pseudo-conversational support.

25.32.1.2 Return Codes
0   Normal return
2502 Invalid operand
2521 Operand not specified

25.32.1.3 Example
`'PSEUDO ON'`

This example turns the pseudo-conversational mode on.

25.32.1.4 Note
The PSEUDO ON/OFF setting is temporary. The pseudo setting exists while the
current exec is executing. Any nested execs inherit the current setting of PSEUDO
When the current exec ends, the value of pseudo upon entry to the exec will be
restored. The SETSYS PSEUDO command defines the systems default setting of
PSEUDO.

CAUTION:
**PSEUDO ON will cause your REXX exec to immediately become
pseudo-conversational which will cause CICS to commit any file changes,
and free any non-shared GETMAINed areas, at the next
pseudo-conversational terminal read.**

25.33 RFS

```
►►──RFS───AUTH──dirid──┬──────PRIVATE──────┬──────────────────►◄
                       │  ┌──PRIVATE──┐     │
                       │  ├──PUBLICR──┤     │
                       │  ├──PUBLICW──┤     │
                       │  └──SECURED──┘     │
         ├─CKDIR──dirid────────────────────┤
         ├─CKFILE──fileid──────────────────┤
         ├─COPY──fileid1──fileid2──────────┤
         ├─DELETE──fileid──────────────────┤
         │              ┌──DATA.──┐        │
         ├─DISKR──fileid─┼────────┤        │
         │              └─stem.───┘        │
         │              ┌──DATA.──┐        │
         ├─DISKW──fileid─┼────────┤        │
         │              └─stem.───┘        │
         │            ┌─────────┐          │
         ├─GETDIR──stem.─┴──dirid─┘        │
         ├─MKDIR──dirid────────────────────┤
         ├─RDIR──dirid─────────────────────┤
         └─RENAME──fileid1──fileid2────────┘
```
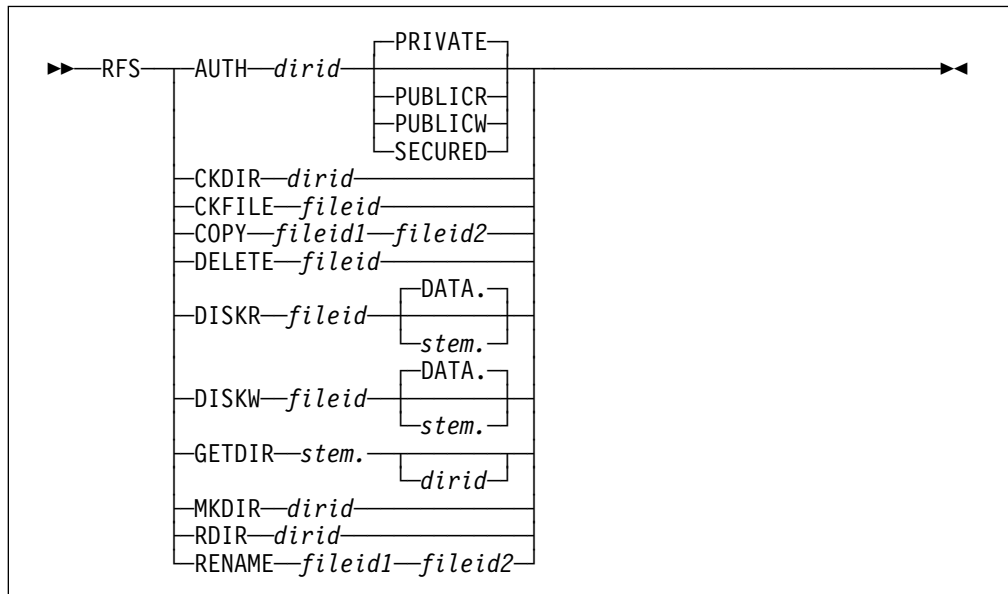
RFS performs file input/output to the REXX File System.

25.33.1.1 Operands

AUTH   is a command that authorizes access to RFS directories. The specified access applies to all files in this directory.

dirid   specifies a REXX File System directory identifier. This is partially or fully qualified. Refer to the CD command, section 25.2, "CD" on page 371, for more information.

PRIVATE   specifies that only the owner of the directory has read/write access to the files. This is the default.

PUBLICR   specifies that any user has read-only access to the files in the directory.

PUBLICW   specifies that any user has read/write access to the files in the directory.

SECURED   specifies that an external security manager grants access to the files in the directory.

CKDIR   is a command that checks for an existing RFS directory level.

CKFILE   is a command that checks to see if the specified, partially or fully qualified, file ID exists.

fileid   specifies the file identifier.

COPY   is a command that copies a file, replacing any existing file.

fileid1   specifies the source file identifier, it may be a fully or partially qualified directory and file identifier.

fileid2   specifies the target file identifier, it may be a fully or partially qualified directory and file identifier.

DELETE   is a command that deletes an RFS file.

fileid   specifies the source file identifier, it may be fully or partially qualified.

DISKR   is a command that reads records from an RFS file.

stem   specifies the name of a stem. (A stem must end in a period.) Refer to section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA.

DISKW   is a command that writes records to an RFS file.

GETDIR   is a command that returns a list of the contents of the current or specified directory into the specified REXX array.

MKDIR   is a command that creates a new RFS directory level.

RDIR   is a command that removes the specified RFS directory.

RENAME   is a command that renames an RFS file to a new name.

fileid1   specifies the source file identifier, it may be a fully or partially qualified directory and file identifier.

fileid2   specifies the source target file identifier, it may be a fully or partially qualified directory and file identifier.

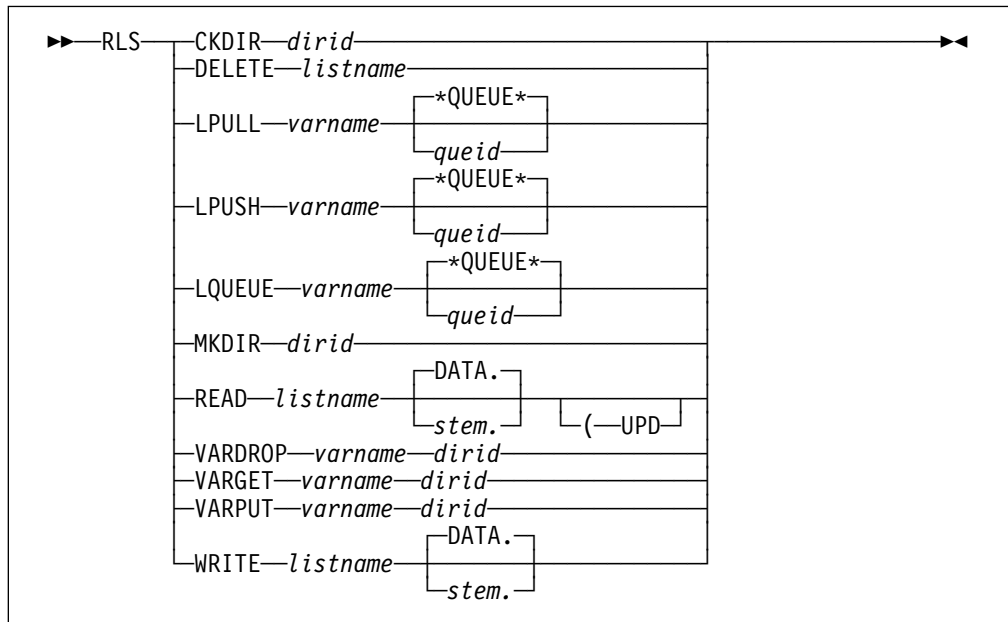## 25.33.1.2 Return Codes
0   Normal return
101   Invalid command
102   Invalid operand
103   File not found
104   Not authorized
105   File already exists
107   Insufficient space in file pool
110   Request failed
111   Invalid file ID
113   Directory not found
115   Directory already exists
116   Directory not specified
121   File corrupted
122   Invalid or out of range stem.0
126   Path error
127   CICS I/O error
128   Command not valid from this location
130   Directory not empty
131   Missing operand
132   Missing file pool data record. File pool is probably not formatted.
199   Internal error

25.33.1.3 Note
File access security checking is performed at the directory level, rather than the file level. If a specified file ID is not a fully qualified ID, the current directory or PATH directories are used in an attempt to resolve the partial name into a fully qualified name; in this case no further checking is necessary. If a fully qualified file ID is used, the directory it resides in is checked for proper access authorization, at file open time.

25.34 RLS

```
►►──RLS──┬─CKDIR──dirid──────────────────────────────────────────┬──►◄
         ├─DELETE──listname─────────────────────────────────────┤
         │                    ┌─*QUEUE*─┐                        │
         ├─LPULL──varname──────┴─queid───┴─────────────────────────┤
         │                    ┌─*QUEUE*─┐                        │
         ├─LPUSH──varname──────┴─queid───┴─────────────────────────┤
         │                    ┌─*QUEUE*─┐                        │
         ├─LQUEUE──varname─────┴─queid───┴─────────────────────────┤
         ├─MKDIR──dirid─────────────────────────────────────────┤
         │                  ┌─DATA.─┐                            │
         ├─READ──listname────┴─stem.─┴──┬──────────┬──────────────┤
         │                              └─(──UPD─┘              │
         ├─VARDROP──varname──dirid──────────────────────────────┤
         ├─VARGET──varname──dirid───────────────────────────────┤
         ├─VARPUT──varname──dirid───────────────────────────────┤
         │                  ┌─DATA.─┐                            │
         └─WRITE──listname───┴─stem.─┴───────────────────────────┘
```

RLS performs list input/output to the REXX List System.

25.34.1.1 Operands

CKDIR is a command that checks for an existing RLS directory level.

dirid specifies a REXX List System directory level identifier. This is partially or fully qualified. Refer to the CLD command, section 25.5, "CLD" on page 375, for more information.

DELETE is a command that deletes an RLS directory level or RLS list.

listname specifies a REXX List System list identifier. This is partially or fully qualified.

LPULL is a command that pulls a record from the top of the queue.

varname specifies a simple REXX variable name. It does not end in a period, distinguishing a variable name from a stem name.

QUEUE specifies the special default name.

queid is the identifier for a special type of RLS list accessed by LPULL, LPUSH, or LQUEUE.

LPUSH is a command that pushes a record onto the top of the queue (LIFO).

LQUEUE is a command that adds a record to the end of the queue (FIFO).

MKDIR is a command that creates a new RLS directory level.

READ is a command that reads records from an RLS list into a stem.

listname specifies the list identifier.

stem. specifies the name of a stem. (A stem must end in a period.) Refer to section 12.4.4, "Stems" on page 145 for more information. The default stem is DATA..

UPD   enqueues on a list for update.

VARDROP   is a keyword indicating that an RLS variable is deleted.

VARGET   is a command that takes an RLS variable and copies it into a REXX variable of the same name.

VARPUT   is a command that takes a REXX variable and copies it into an RLS variable of the same name.

WRITE   is a command that writes records to an RLS list from a stem.

25.34.1.2 Return Codes

0   Normal return
701   Invalid command
702   Invalid operand
713   Directory not found
715   Directory already exists
716   Directory not specified
723   List not found
726   List not specified
728   List is in update mode
729   List is not in update mode
730   User is not signed on
732   Queue empty
733   Named queue not found
736   Stem or variable not specified
737   Stem or variable name too long
738   Stem or variable count invalid
743   Block not found
746   CICGETV error
747   GETMAIN error
748   FREEMAIN error
749   ENQ error
750   DEQ error
751   Dynamic area GETMAIN error
752   Error in saved variable data
753   Saved variable not found
754   User not owner of list

25.35 SCRNINFO

```
►►──SCRNINFO──────────────────────────────────────────────────────►◄
```
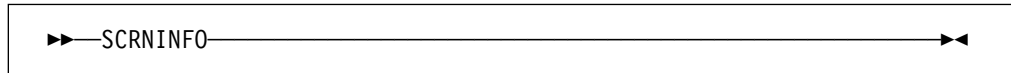
SCRNINFO returns a two-digit decimal screen height (in lines) in the variable
SCRNHT, and returns a three-digit decimal screen width (in columns) in the
variable SCRNWD.

25.35.1.1 Return Codes
n   specifies the return code passed back by CICS if an error is detected
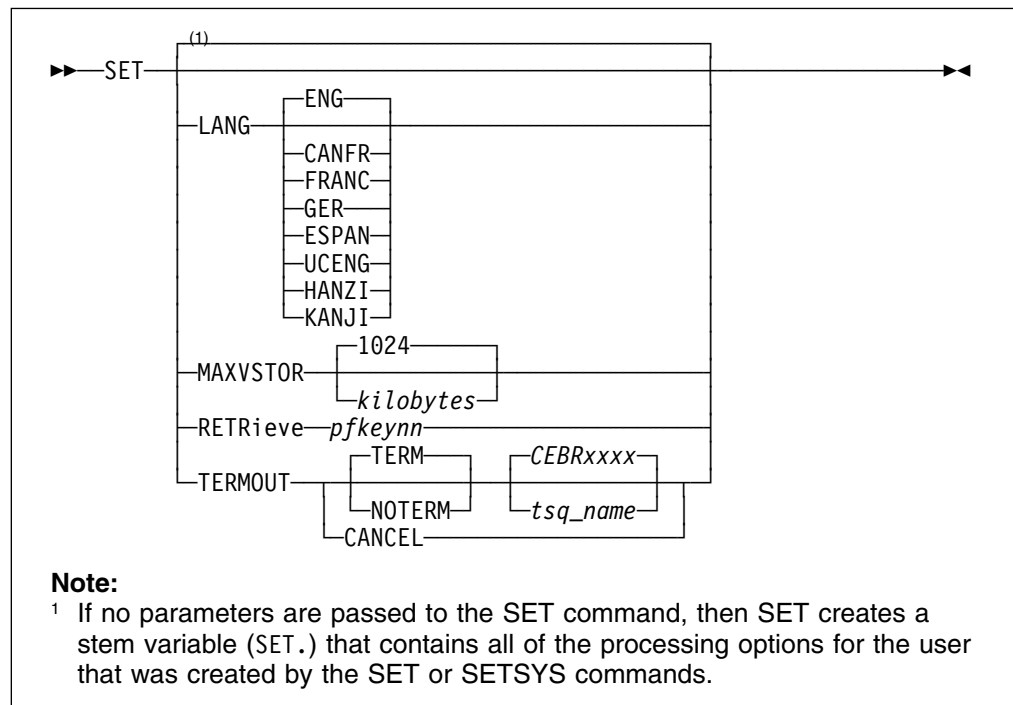0   Normal return
-499 Internal error

25.35.1.2 Example
```
'SCRNINFO'
```

25.35.1.3 Notes
1. The screen information that is returned is obtained by doing an EXEC CICS
   ASSIGN SCRNHT(*scrnht*) SCRNWD(*scrnwd*).

2. The values for screen height and screen width is 0 if there is no terminal
   attached.

25.36 SET

```
         (1)
►►─SET─┬──────────────────────────────────────────┬─►◄
       │         ┌─ENG────┐                        │
       ├─LANG────┼────────┼────┐                   │
       │         ├─CANFR──┤    │                   │
       │         ├─FRANC──┤    │                   │
       │         ├─GER────┤    │                   │
       │         ├─ESPAN──┤    │                   │
       │         ├─UCENG──┤    │                   │
       │         ├─HANZI──┤    │                   │
       │         └─KANJI──┘    │                   │
       │            ┌─1024──────┐                  │
       ├─MAXVSTOR───┼───────────┼─────┐            │
       │            └─kilobytes─┘     │            │
       ├─RETRieve─pfkeynn─────────────┘            │
       │           ┌─TERM────┐  ┌─CEBRxxxx─┐       │
       └─TERMOUT───┼─────────┼──┼──────────┼───────┘
                   ├─NOTERM──┘  └─tsq_name─┘
                   └─CANCEL───
```

Note:
[1] If no parameters are passed to the SET command, then SET creates a
stem variable (SET.) that contains all of the processing options for the user
that was created by the SET or SETSYS commands.

SET sets the REXX/CICS processing options for the current user.

25.36.1.1 Operands

LANG specifies that one of the following languages are available:

ENG	English
CANFR	Canadian French
FRANC	French
GER	German
ESPAN	Spanish
UCENG	Uppercase English
HANZI	Traditional Chinese
KANJi	Kanji

MAXVSTOR
is a keyword placing limits on virtual storage use.

kilobytes specifies the number of kilobytes.

RETRieve allows a PF key being set to retrieve the last line entered.

pfkeynn specifies the PF key number.

TERMOUT sends terminal line-mode output to a CICS temporary storage queue
(for example: SAY and TRACE output) even when a terminal is
attached.

TERM specifies that linemode output will be sent to the terminal.

NOTERM specifies that terminal line-mode output is not displayed on the
terminal.

CANCEL specifies that linemode output will not be sent to a CICS temporary storage queue.

tsq_name specifies the CICS temporary storage queue name. The default *tsq_name* is CEBRxxxx where *xxxx* is your terminal ID. (If you enter the CEBR transaction without specifying a TSQ name, this is the default name that is used.)

25.36.1.2 Return Codes
0 Normal return
421 Invalid SET subcommand
422 Error storing variable
423 Invalid language
425 Invalid MAXVSTOR operand
426 Invalid RETRIEVE PFkey operand
427 Invalid TERMOUT operand

25.36.1.3 Example
```
'SET TERMOUT TERM TSQ1'
```

This example sets the processing option to send terminal line-mode output to temporary storage queue TSQ1.

25.36.1.4 Notes
1. SET LANG only affects REXX syntax messages.
2. Multiple REXX/CICS users can be using different languages at the same time in the same region.

25.37 SETSYS

Note: This is an **authorized** command.

```
►►──SETSYS──LANG──┬─ENG───┬──────────────────────────────────────►◄
                  ├─CANFR─┤
                  ├─FRANC─┤
                  ├─GER───┤
                  ├─ESPAN─┤
                  ├─UCENG─┤
                  ├─HANZI─┤
                  └─KANJI─┘
              ┌─1024──────┐
   ─MAXVSTOR──┴─kilobytes─┘
   ─RETRieve─pfkeynn─
           ┌─ON──┐
   ─PSEUDO─┤     ├
           └─OFF─┘
                  ┌◄─────────┐
   ─AUTHClib──────┴─lib.sublib─┘
                  ┌◄─────────┐
   ─AUTHElib──────┴─lib.sublib─┘
```
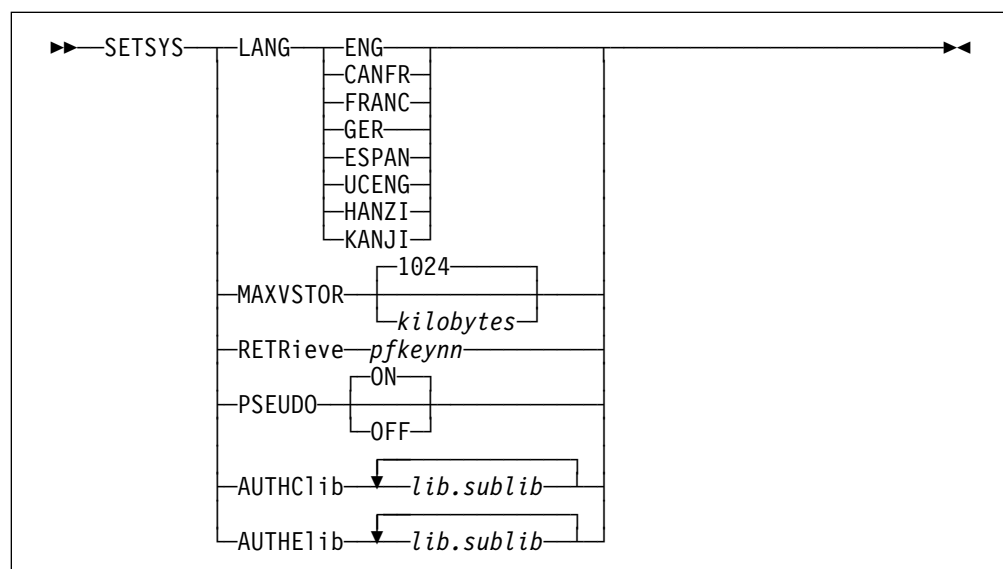
SETSYS sets the REXX/CICS processing options for the system.

25.37.1.1 Operands

LANG   specifies that one of the following languages are available:

ENG	English
CANFR	Canadian French
FRANC	French
GER	German
ESPAN	Spanish
UCENG	Uppercase English
HANZI	Traditional Chinese
KANJi	Kanji

MAXVSTOR

is a keyword placing limits on virtual storage use.

kilobytes   specifies the number of kilobytes.

RETRieve   allows a PF key being set to retrieve the last line entered.

pfkeynn   specifies the PF key number.

PSEUDO   establishes the default region-wide REXX/CICS automatic pseudo-conversational setting. For more information on the PSEUDO command, see section 25.32, "PSEUDO" on page 415.

ON   specifies that the automatic pseudo-conversational setting is on. This is the default.

OFF   specifies that the automatic pseudo-conversational setting is off.

AUTHCLIB   establishes the VSE Librarian sublibraries containing the REXX/CICS authorized commands.

lib.sublib   the name of the VSE Librarian sublibraries containing the REXX/CICS authorized commands.

AUTHELIB   establishes the VSE Librarian sublibraries containing the REXX/CICS authorized execs.

lib.sublib   the name of the VSE Librarian sublibraries containing the REXX/CICS authorized execs.

25.37.1.2 Return Codes
0   Normal return
2721 Invalid SETSYS subcommand
2722 Error storing variable
2723 Invalid language
2725 Invalid MAXVSTOR operand
2726 Invalid RETRIEVE PFkey operand
2727 Invalid TERMOUT operand
2732 Invalid PSEUDO operand
2735 Invalid VSE Librarian sublibrary name
2739 VARGET for AUTHCLIB or AUTHELIB failed
2740 Too many sublibraries specified

25.37.1.3 Example
`'SETSYS RETRIEVE 12'`

This example set PF key 12 as the retrieve key.

25.38 S2C

```
►►──S2C──server_rexx_varname──────────────────────────────►◄
                          └─client_rexx_varname─┘
```
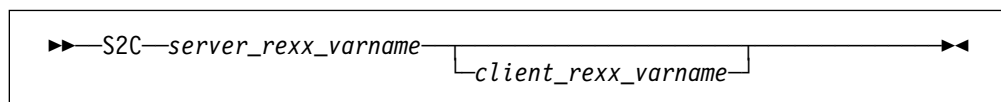
S2C copies a server REXX variable to a client REXX variable.

25.38.1.1 Operands

server_rexx_varname
> is the name of the server REXX variable copying from.

client_rexx_varname
> is the optional name of the client REXX variable copying into. If you do not specify, it defaults to the same as the *server_rexx_varname*.

25.38.1.2 Return Codes

0   Normal return
2840 No variable name specified
2841 Error retrieving variable
2842 Error storing variable
2848 No client available

25.38.1.3 Example

```
'S2C VARA VARB'
```

This example shows the contents of the server REXX variable VARA copying into the client REXX variable VARB. The length of VARB is the same as the length of VARA.

25.38.1.4 Notes

1. The maximum length allowed of a *varname*, for this command, is 250 characters. If a longer name is specified, only the first 250 characters are used.

2. The maximum length of a variable copied by S2C is 6000 bytes.

3. You can only use the S2C command from a server exec.

25.39 TERMID

```
►►──TERMID─────────────────────────────────────────────────────────►◄
```
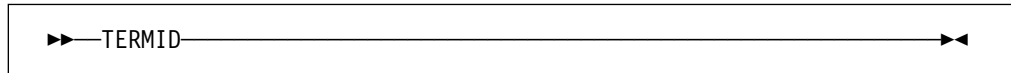
TERMID returns the four-character CICS terminal ID from the CICS field EIBTRMID in the variable TERMID.

25.39.1.1 Return Codes
0   Normal return
2921 Error in obtaining terminal ID
2928 Error setting TERMID value

25.39.1.2 Example
`'TERMID'`

This example places the CICS terminal ID from the CICS field EIBTRMID in the variable TERMID.

25.40 WAITREAD

```
►►──WAITREAD──────────────────────────────────────────────────────────►◄
```
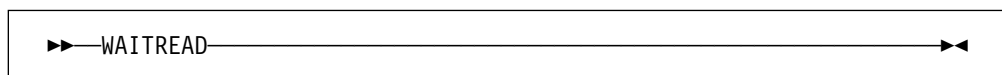
WAITREAD performs full screen terminal input and places the results into the compound variable with:

WAITREAD.0
   containing the number of elements returned.

WAITREAD.1
   containing the AID description.

WAITREAD.2
   containing the cursor position.

WAITREAD.3 through **WAITREAD.***n*
   remaining 3270 fields that have been modified.

25.40.1.1 Return Codes
0   Normal return
3021 No terminal is attached
3099 Internal error

25.40.1.2 Example
```
'WAITREAD'
```

This example reads input from the terminal screen and places the results in the REXX compound variable WAITREAD.1 through WAITREAD.*n*.

25.40.1.3 Note
A read modified is performed to the terminal and an array is returned with information from this read. The format of these elements is: *field_row field_column data*

25.41 WAITREQ

```
►►──WAITREQ──────────────────────────────────────────────────►◄
```
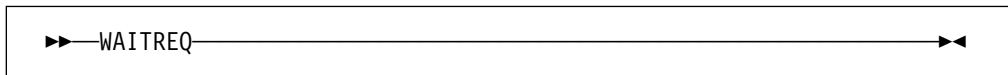
WAITREQ is used only in REXX servers causing the server to wait for a request.
After a request is received, it is placed into REXX variable REQUEST.

25.41.1.1 Return Codes
0   Normal return
3121 WAITREQ not enabled
3122 Exec not a server
3123 Error saving request variable
3199 Internal error

Note: The return code reflected to the client program is the value of the REXX
server variable at entry to the WAITREQ command or at exit of the server exec.

25.41.1.2 Example
```
'WAITREQ'
```

This example causes the server exec to be suspended until another request for the
server is encountered. When the request is encountered the server exec is
restored to its presuspended status with a new request value.

Appendix A. Error Numbers and Messages

External interfaces to the language processor can generate three of the error messages either before the language processor gains control or after control has left the language processor. Therefore, SIGNAL ON SYNTAX cannot trap these errors. The error numbers involved are: 3 and 5 (if the initial requirements for storage could not be met) and 26 (if on exit the returned string could not be converted to form a valid return code). Error 4 can be trapped only by SIGNAL ON HALT or CALL ON HALT. Five errors the language processor detects cannot be trapped by SIGNAL ON SYNTAX unless the label SYNTAX appears earlier in the program than the clause with the error. These errors include: 6, 12, 13, 22, and 30. The following is a list of the error codes and their associated CICS messages:

Table 4. List of Error Codes and CICS Messages

Error code	CICS message	Error code	CICS message
No number	CICREX255T	Error 26	CICREX466E
Error 3	CICREX451E	Error 27	CICREX467E
Error 4	CICREX452E	Error 28	CICREX486E
Error 5	CICREX450E	Error 29	CICREX487E
Error 6	CICREX453E	Error 30	CICREX468E
Error 7	CICREX454E	Error 31	CICREX469E
Error 8	CICREX455E	Error 32	CICREX492E
Error 9	CICREX456E	Error 33	CICREX488E
Error 10	CICREX457E	Error 34	CICREX470E
Error 11	CICREX458E	Error 35	CICREX471E
Error 12	CICREX459E	Error 36	CICREX472E
Error 13	CICREX460E	Error 37	CICREX473E
Error 14	CICREX461E	Error 38	CICREX489E
Error 15	CICREX462E	Error 39	CICREX474E
Error 16	CICREX463E	Error 40	CICREX475E
Error 17	CICREX465E	Error 41	CICREX476E
Error 18	CICREX491E	Error 42	CICREX477E
Error 19	CICREX482E	Error 43	CICREX478E
Error 20	CICREX483E	Error 44	CICREX479E
Error 21	CICREX464E	Error 45	CICREX480E
Error 22	CICREX449E	Error 46	CICREX218E
Error 23	CICREX1106E	Error 47	CICREX219E
Error 24	CICREX484E	Error 48	CICREX490E
Error 25	CICREX485E	Error 49	CICREX481E

In these messages, the term "language processor" refers to the REXX/CICS interpreter.

In addition to the following error messages, the language processor issues this terminal (unrecoverable) message:

CICREX255T Insufficient storage for Exec interpreter

Explanation: There is insufficient storage for the language processor to initialize itself.

System Action: Execution is terminated at the point of the error.

User Response: Redefine storage and reissue the command.

The following are the REXX error messages:

CICREX218E Error 46 Invalid variable reference

Explanation: Within an ARG, DROP, PARSE, PULL, or PROCEDURE instruction, the syntax of a variable reference (a variable whose value is to be used, indicated by its name being enclosed in parentheses) is incorrect. The right parenthesis that should immediately follow the variable name may be missing.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX219E Error 47 Unexpected label

Explanation: A label, being used incorrectly, was encountered in the expression being evaluated for an INTERPRET instruction or in an expression entered during interactive debug.

System Action: Execution stops.

User Response: Do not use a label in these expressions.

CICREX449E Error 22 running *fn ft*, **line** *nn*: **Invalid character string**

Explanation: A character string scanned with OPTIONS ETMODE in effect contains one of the following:

- Unmatched shift-out (SO) and shift-in (SI) control characters

- An odd number of bytes between the shift-out (SO) and shift-in (SI) characters.

System Action: Execution stops.

User Response: Correct the incorrect character string in the EXEC file.

CICREX450E Error 5 running *fn ft*, **line** *nn*: **User storage exhausted or request exceeds limit**

Explanation: While trying to process a program, the language processor was unable to get the resources it needed to continue. (For example, it could not get the space needed for its work areas, variables, and so on.) The program that called the language processor may already have used up most of the available storage itself. Or a request for storage may have been for more than the implementation maximum.

System Action: Execution stops.

User Response: Run the exec or macro on its own, or check a program issuing NUCXLOAD for a possible loop that has not terminated properly. See your system administrator for additional storage requirements.

CICREX451E Error 3 running *fn ft*: **Program is unreadable**

Explanation: The REXX program could not be read. This is probably due to bad data in the exec file or an I/O error.

System Action: Execution stops.

User Response: Examine and correct the exec file.

Note: Sequence numbers are not allowed in columns 73 through 80 in a REXX exec.

CICREX452E Error 4 running *fn ft*, **line** *nn*: **Program interrupted**

Explanation: The system interrupted execution of your REXX program. Certain utility modules may force this condition if they detect a disastrous error condition.

System Action: Execution stops.

User Response: Look for a problem with a utility module called in your exec or macro.

CICREX453E Error 6 running *fn ft*, **line** *nn*: **Unmatched "/*" or quote**

Explanation: A comment or literal string was started but never finished. This could be because the language processor detected:

- The end of the file (or the end of data in an INTERPRET statement) without finding the ending "*/" for a comment or the ending quote for a literal string
- The end of the line for a literal string.

System Action: Execution stops.

User Response: Edit the exec and add the closing "*/" or quote. You can also insert a TRACE SCAN statement at the top of your program and rerun it. The resulting output should show where the error exists.

CICREX454E Error 7 running *fn ft*, **line** *nn*: **WHEN or OTHERWISE expected**

Explanation: The language processor expects a series of WHENs and an OTHERWISE within a SELECT statement. This message is issued when any other instruction is found or if all WHEN expressions are found to be false and an OTHERWISE is not present. The error is often caused by forgetting the DO and END instructions around the list of instructions following a WHEN. For example:

```
   WRONG                        RIGHT

Select                     Select
  When a1=b1 then            When a1=b1 then DO
    Say 'A1 equals B1'          Say 'A1 equals B1'
    exit                        exit
  Otherwise nop               end
  end                        Otherwise nop
                             end
```

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX455E Error 8 running *fn ft*, **line** *nn*: **Unexpected THEN or ELSE**

Explanation: The language processor has found a THEN or an ELSE that does not match a corresponding IF clause. This situation is often caused by using an incorrect DO-END in the THEN part of a complex IF-THEN-ELSE construction. For example,

```
        WRONG                      RIGHT
If a1=b1 then do;          If a1=b1 then do;
  Say EQUALS                 Say EQUALS
  exit                       exit
 else                        end
   Say NOT EQUALS           else
                              Say NOT EQUALS
```

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX456E Error 9 running *fn ft*, **line** *nn***: Unexpected WHEN or OTHERWISE**

Explanation: The language processor has found a WHEN or OTHERWISE instruction outside of a SELECT construction. You may have accidentally enclosed the instruction in a DO-END construction by leaving off an END instruction, or you may have tried to branch to it with a SIGNAL statement (which cannot work because the SELECT is then terminated).

System Action: Execution stops.

User Response: Make the necessary correction.

CICREX457E Error 10 running *fn ft*, **line** *nn***: Unexpected or unmatched END**

Explanation: The language processor has found more ENDs in your program than DOs or SELECTs, or the ENDs were placed so that they did not match the DOs or SELECTs. Putting the name of the control variable on ENDs that close repetitive loops can help locate this kind of error.

This message can be caused if you try to signal into the middle of a loop. In this case, the END will be unexpected because the previous DO will not have been executed. Remember, also, that SIGNAL terminates any current loops, so it cannot be used to transfer control from one place inside a loop to another.

This message can also be caused if you place an END immediately after a THEN or ELSE construction or if you specified a *name* on the END keyword that does not match the *name* following DO.

System Action: Execution stops.

User Response: Make the necessary corrections. It may be helpful to use TRACE Scan to show the structure of the program, making it easier to find your error. Putting the name of the control variable on ENDs that close repetitive loops can also help locate this kind of error.

CICREX458E Error 11 running *fn ft*, **line** *nn***: Control stack full**

Explanation: This message is issued if you exceed the limit of 250 levels of nesting of control structures (DO-END, IF-THEN-ELSE, and so forth) or when user storage limit is reached, whichever is less.

This message could be caused by a looping INTERPRET instruction, such as:

```
line='INTERPRET line'
INTERPRET line
```

These lines would loop until they exceeded the nesting level limit and this message would be issued. Similarly, a recursive subroutine that does not terminate correctly could loop until it causes this message.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX459E Error 12 running *fn ft*, **line** *nn*: **Clause too long**

Explanation: You have exceeded the limit for the length of the internal representation of a clause. The actual limit is the amount of storage that can be obtained on a single request.

If the cause of this message is not obvious to you, it may be due to a missing quote that has caused a number of lines to be included in one long string. In this case, the error probably occurred at the start of the data included in the clause traceback (flagged by +++ on the console).

The internal representation of a clause does not include comments or multiple blanks that are outside of strings. Note also that any symbol (name) or string gains two characters in length in the internal representation.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX460E Error 13 running *fn ft*, **line** *nn*: **Invalid character in program**

Explanation: The language processor found an incorrect character outside of a literal (quoted) string. Valid characters are:

A-Z a-z 0-9   (Alphamerics)

@ # $ ¢ . ? ! _   (Name Characters)

& * ( ) - + = \ ¬ ' " ; : < , > / | (Special Characters)

If surrounded by X'0E' (shift-out) and X'0F' (shift-in), and if ETMODE is on, the following are also valid characters:

X'41' - X'FE'   (DBCS Characters)

Some causes of this error are:

1. Using accented and other language-specific characters in symbols.
2. Using DBCS characters without ETMODE in effect.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX461E Error 14 running *fn ft*, **line** *nn*: **Incomplete DO/SELECT/IF**

Explanation: The language processor has reached the end of the file (or end of data for an INTERPRET instruction) and has found that there is a DO or SELECT without a matching END, or an IF that is not followed by a THEN clause.

System Action: Execution stops.

User Response: Make the necessary corrections. You can use TRACE Scan to show the structure of the program, making it easier to find where the missing END or THEN should be. Putting the name of the control variable on ENDs that close repetitive loops can also help locate this kind of error.

CICREX462E Error 15 running *fn ft*, **line** *nn*: **Invalid hexadecimal or binary string**

Explanation: Binary strings are new in REXX and the language processor may now be considering the string in your statement to be binary when that was not your intention.

For the language processor, hexadecimal strings cannot have leading or trailing blanks and can have imbedded blanks only at byte boundaries. Only the digits 0–9 and the letters a–f and A–F are allowed. Similarly, binary strings can have blanks only at the boundaries of groups of four binary digits, and only the digits 0 and 1 are allowed.

The following are all valid hexadecimal or binary constants:

```
'13'x                          '0101 1100'b
'A3C2 1c34'x                   '001100'B
'1de8'x                        "0 11110000"b
```

You may have mistyped one of the digits, for example, typing a letter o instead of 0. Or you may have put the 1-character symbol X, x, B, or b (the name of the variable X or B, respectively) after a literal string, when the string is not intended as a hexadecimal or binary specification. In this case, use the explicit concatenation operator (||) to concatenate the string to the value of the symbol.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX463E Error 16 running *fn ft*, **line** *nn*: **Label not found**

Explanation: The language processor could not find the label specified by a SIGNAL instruction or a label matching an enabled condition when the corresponding (trapped) event occurred. You may have mistyped the label or forgotten to include it, or you may have typed it in mixed case when it needs to be in uppercase.

System Action: Execution stops. The name of the missing label is included in the error traceback.

User Response: Make the necessary corrections.

CICREX464E Error 21 running *fn ft*, **line** *nn*: **Invalid data on end of clause**

Explanation: You have followed a clause, such as SELECT or NOP, by some data other than a comment.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX465E Error 17 running *fn ft*, **line** *nn*: **Unexpected PROCEDURE**

Explanation: The language processor encountered a PROCEDURE instruction in an incorrect position. This could occur because no internal routines are active, because a PROCEDURE instruction has already been encountered in the internal routine, or because the PROCEDURE instruction was not the first instruction executed after the CALL or function invocation. This error can be caused by "dropping through" to an internal routine, rather than invoking it with a CALL or a function call.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX466E Error 26 running *fn ft*, **line** *nn*: **Invalid whole number**

Explanation: The language processor found an expression in the NUMERIC instruction, a parsing positional pattern, or the right-hand term of the exponentiation (**) operator that did not evaluate to a whole number, or was greater than the limit, for these uses, of 999999999.

This message can also be issued if the return code passed back from an EXIT or RETURN instruction (when a REXX program is called as a command) is not a whole number or will not fit in a general register. This error may be due to mistyping the name of a symbol so that it is not the name of a variable in the expression on any of these statements. This might be true, for example, if you entered "EXIT CR" instead of "EXIT RC."

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX467E Error 27 running *fn ft*, **line** *nn*: **Invalid DO syntax**

Explanation: The language processor found a syntax error in the DO instruction. You might have used BY, TO, FOR, WHILE, OR UNTIL twice, or used a WHILE and an UNTIL.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX468E Error 30 running *fn ft*, **line** *nn*: **Name or string > 250 characters**

Explanation: The language processor found a variable or a literal (quoted) string that is longer than the limit.

The limit for names is 250 characters, following any substitutions. A possible cause of this error is the use of a period (.) in a name, causing an unexpected substitution.

The limit for a literal string is 250 characters. This error can be caused by leaving off an ending quote (or putting a single quote in a string) because several clauses can be included in the string. For example, the string `'don't'` should be written as `'don''t'` or `"don't"`.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX469E Error 31 running *fn ft*, **line** *nn*: **Name starts with number or "."**

Explanation: The language processor found a symbol whose name begins with a numeric digit or a period (.). The REXX language rules do not allow you to assign a value to a symbol whose name begins with a number or a period because you could then redefine numeric constants, and that would be catastrophic.

System Action: Execution stops.

User Response: Rename the variable correctly. It is best to start a variable name with an alphabetic character, but some other characters are allowed.

CICREX470E Error 34 running *fn ft*, **line** *nn*: **Logical value not 0 or 1**

Explanation: The language processor found an expression in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a 0 or 1. Any value operated on by a logical operator (¬, \, l, &, or &&) must result in a 0 or 1. For example, the phrase "If result then exit rc" will fail if result has a value other than 0 or 1. Thus, the phrase would be better written as `If result¬=0 then exit rc`.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX471E Error 35 running *fn ft*, **line** *nn*: **Invalid expression**

Explanation: The language processor found a grammatical error in an expression. This could be because:

* You ended an expression with an operator.
* You specified, in an expression, two operators next to one another with nothing in between them.
* You did not specify an expression when one was required.
* You did not specify a right parenthesis when one was required.
* You used special characters (such as operators) in an intended character expression without enclosing them in quotation marks.

An example of the last case is that LISTFILE * * * should be written as `LISTFILE '* * *'` (if LISTFILE is not a variable) or even as `'LISTFILE * * *'`.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX472E Error 36 running *fn ft*, **line** *nn*: **Unmatched "(" in expression**

Explanation: The language processor found an unmatched parenthesis within an expression. You will get this message if you include a single parenthesis in a command without enclosing it in quotation marks. For example, COPY A B C A B D (REP should be written as `COPY A B C A B D '('REP`.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX473E Error 37 running *fn ft*, **line** *nn*: **Unexpected "," or ")"**

Explanation: The language processor found a comma (,) outside a routine invocation or too many right parentheses in an expression. You will get this message if you include a comma in a character expression without enclosing it in quotation marks. For example, the instruction:

`Say Enter A, B, or C`

should be written as:

`Say 'Enter A, B, or C'`

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX474E Error 39 running *fn ft*, **line** *nn*: **Evaluation stack overflow**

Explanation: The language processor was not able to evaluate the expression because it is too complex (many nested parentheses, functions, and so forth).

System Action: Execution stops.

User Response: Break up the expressions by assigning subexpressions to temporary variables.

CICREX475E Error 40 running *fn ft*, **line** *nn*: **Invalid call to routine**

Explanation: The language processor encountered an incorrectly used call to a routine. Some possible causes are:

- You passed incorrect data (arguments) to the built-in or external routine (this depends on the actual routine).
- You passed too many arguments to the built-in, external, or internal routine.
- The module invoked was not compatible with the language processor.

If you were not trying to invoke a routine, you may have a symbol or a string adjacent to a "(" when you meant it to be separated by a space or an operator. This causes it to be seen as a function call. For example, TIME(4+5) should probably be written as `TIME*(4+5)`.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX476E Error 41 running *fn ft*, **line** *nn*: **Bad arithmetic conversion**

Explanation: The language processor found a term in an arithmetic expression that was not a valid number or that had an exponent outside the allowed range of -999999999 to +999999999.

You may have mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotation marks. For example, the command MSG * Hi! should be written as `'MSG * Hi!'`, otherwise the language processor will try to multiply "MSG" by "Hi!."

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX477E Error 42 running *fn ft*, **line** *nn*: **Arithmetic overflow/underflow**

Explanation: The language processor encountered a result of an arithmetic operation that required an exponent greater than the limit of 9 digits (more than 999999999 or less than -999999999).

This error can occur during evaluation of an expression (often as a result of trying to divide a number by 0), or during the stepping of a DO loop control variable.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX478E Error 43 running *fn ft*, **line** *nn*: **Routine not found**

Explanation: The language processor was unable to find a routine called in your program. You invoked a function within an expression, or in a subroutine invoked by CALL, but the specified label is not in the program, or is not the name of a built-in function, and REXX/CICS is unable to locate it externally.

The simplest, and probably most common, cause of this error is mistyping the name. Another possibility may be that one of the standard function packages is not available.

If you were not trying to invoke a routine, you may have put a symbol or string adjacent to a "(" when you meant it to be separated by a space or operator. The language processor would see that as a function invocation. For example, the string 3(4+5) should be written as 3*(4+5).

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX479E Error 44 running *fn ft*, **line** *nn*: **Function did not return data**

Explanation: The language processor invoked an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

This may be due to specifying the name of a module that is not intended for use as a REXX function. It should be called as a command or subroutine.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX480E Error 45 running *fn ft*, **line** *nn*: **No data specified on function RETURN**

Explanation: A REXX program has been called as a function, but an attempt is being made to return (by a RETURN; instruction) without passing back any data. Similarly, an internal routine, called as a function, must end with a RETURN statement specifying an expression.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX481E Error 49 running *fn ft*, **line** *nn*: **Language processor failure**

Explanation: The language processor carries out numerous internal self-consistency checks. It issues this message if it encounters a severe error.

System Action: Execution stops.

User Response: Report any occurrence of this message to your IBM representative.

CICREX482E Error 19 running *fn ft***, line** *nn***: String or symbol expected**

Explanation: The language processor expected a symbol following the CALL or SIGNAL instructions, but none was found. You may have omitted the string or symbol, or you may have inserted a special character (such as a parenthesis) in it.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX483E Error 20 running *fn ft***, line** *nn***: Symbol expected**

Explanation: The language processor either expected a symbol following the CALL ON, CALL OFF, END, ITERATE, LEAVE, NUMERIC, PARSE, PROCEDURE, SIGNAL ON, or SIGNAL OFF keywords or expected a list of symbols or variable references following the DROP, UPPER, or PROCEDURE (with EXPOSE option) keywords. Either there was no symbol when one was required or some other characters were found.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX484E Error 24 running *fn ft***, line** *nn***: Invalid TRACE request**

Explanation: The language processor issues this message when:

- The action specified on a TRACE instruction, or the argument to the TRACE built-in function, starts with a letter that does not match one of the valid alphabetic character options. The valid options are A, C, E, F, I, L, N, O, R, or S.
- An attempt is made to request TRACE Scan when inside any control construction or while in interactive debug
- In interactive trace, you enter a number that is not a whole number.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX485E Error 25 running *fn ft***, line** *nn***: Invalid sub-keyword found**

Explanation: The language processor expected a particular sub-keyword at this position in an instruction and something else was found. For example, the NUMERIC instruction must be followed by the sub-keyword DIGITS, FUZZ, or FORM. If NUMERIC is followed by anything else, this message is issued.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX486E Error 28 running *fn ft***, line** *nn***: Invalid LEAVE or ITERATE**

Explanation: The language processor encountered an incorrect LEAVE or ITERATE instruction. The instruction was incorrect because of one of the following:

- No loop is active.
- The name specified on the instruction does not match the control variable of any active loop.

Note that internal routine calls and the INTERPRET instruction protect DO loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine.

You can cause this message to be issued if you use the SIGNAL instruction to transfer control within or into a loop. A SIGNAL instruction terminates all active loops, and any ITERATE or LEAVE instruction issued then would cause this message to be issued.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX487E Error 29 running *fn ft*, **line** *nn*: **Environment name too long**

Explanation: The language processor encountered an environment name specified on an ADDRESS instruction that is longer than the limit of 8 characters.

System Action: Execution stops.

User Response: Specify the environment name correctly.

CICREX488E Error 33 running *fn ft*, **line** *nn*: **Invalid expression result**

Explanation: The language processor encountered an expression result that is incorrect in its particular context. The result may be incorrect in one of the following:

- ADDRESS VALUE expression
- NUMERIC DIGITS expression
- NUMERIC FORM VALUE expression
- NUMERIC FUZZ expression
- OPTIONS expression
- SIGNAL VALUE expression
- TRACE VALUE expression.

(FUZZ must be smaller than DIGITS.)

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX489E Error 38 running *fn ft*, **line** *nn*: **Invalid template or pattern**

Explanation: The language processor found an incorrect special character, for example %, within a parsing template, or the syntax of a variable trigger was incorrect (no symbol was found after a left parenthesis). This message is also issued if the WITH sub-keyword is omitted in a PARSE VALUE instruction.

System Action: Execution stops.

User Response: Make the necessary corrections.

CICREX490E Error 48 running *fn ft*, **line** *nn*: **Failure in system service**

Explanation: The language processor halts execution of the program because some system service, such as user input or output or manipulation of the console stack, has failed to work correctly.

System Action: Execution stops.

User Response: Ensure that your input is correct and that your program is working correctly. If the problem persists, notify your system support personnel.

CICREX491E Error 18 running *fn ft*, **line** *nn*: **THEN expected**

Explanation: All REXX IF and WHEN clauses must be followed by a THEN clause. Another clause was found before a THEN statement was found.

System Action: Execution stops.

User Response: Insert a THEN clause between the IF or WHEN clause and the following clause.

CICREX492E Error 32 running *fn ft*, **line** *nn*: **Invalid use of stem**

Explanation: The REXX program attempted to change the value of a symbol that is a stem. (A stem is that part of a symbol up to the first period. You use a stem when you want to affect all variables beginning with that stem.) This may be in the UPPER instruction where the action in this case is unknown, and therefore in error.

System Action: Execution stops.

User Response: Change the program so that it does not attempt to change the value of a stem.

CICREX1106E Error 23 running *fn ft*, **line** *nn*: **Invalid SBCS/DBCS mixed string.**

Explanation: A character string that has unmatched SO-SI pairs (that is, an SO without an SI) or an odd number of bytes between the SO-SI characters was processed with OPTIONS EXMODE in effect.

System Action: Execution stops.

User Response: Correct the incorrect character string.

Appendix B. Return Codes

This appendix is a list of all the REXX/CICS return codes.

B.1.1 Panel Facility

4   Warning. Panel facility continues processing
8   Programmer error
10   Programmer error with state information
12   CICS command error
14   RFS errors; reason code contains the RFS return code
16   Internal system error

See Chapter 24, "REXX/CICS Panel Facility" on page 341, for additional codes (for example: state and reason codes).

B.1.2 SQL

n   specifies the SQLCODE if the SQL statement resulted in an error or warning
0   The SQL statement was processed by the EXECSQL environment
30   There was not enough memory to build the SQLDSECT variable
31   There was not enough memory to build the SQL statement area
32   There was not enough memory to build the SQLDA variable
33   There was not enough memory to build the results area for the SELECT statement

B.1.3 RFS and FLST

0   Normal return
101   Invalid command
102   Invalid operand
103   File not found
104   Not authorized
105   File already exists
107   Insufficient space in filepool
110   Request failed
111   Invalid file ID
113   Directory not found
115   Directory already exists
116   Directory not specified
121   File corrupted
122   Invalid or out of range stem.0
126   Path error
127   CICS I/O error
128   Command not valid from this location
130   Directory not empty
131   Missing operand
132   Missing file pool data record. File pool is probably not formatted.
199   Internal error

441

B.1.4 EDITOR and EDIT

0	Normal return
201	Invalid command
202	Invalid operand
203	File not found
204	Not authorized
207	Insufficient space in filepool
210	Request failed
211	Invalid file ID
223	Search argument not found
226	File is currently being edited
229	Number out of range
230	Cursor is not in file area
231	Out of virtual storage
232	Prefix command conflict
236	Not defined
299	Internal error

B.1.5 DIR

0	Normal return
321	Cannot access current RFS directory information
322	Invalid stem name
325	Error retrieving RFS directory

B.1.6 SET

0	Normal return
421	Invalid SET subcommand
422	Error storing variable
423	Invalid language
425	Invalid MAXVSTOR operand
426	Invalid RETRIEVE PFkey operand
427	Invalid TERMOUT operand

B.1.7 CD

0	Normal return
521	Error in retrieving filepool definition
522	Error in creating default RFS directory
523	Error in storing current RFS directory information
524	RFS directory does not exist or access not authorized
525	Error in retrieving directory information
526	Invalid filepool/directory
527	Cannot go back past root directory
528	Error setting result value

B.1.8 PATH

0	Normal return
625	Error retrieving path information
626	Invalid RFS directory name
628	Error setting RESULT value
629	Invalid VSE Librarian sublibrary
630	Error storing path information

631   No path currently defined
632   Resulting PATH contains no RFS directories or library names

B.1.9 RLS

0   Normal return
701   Invalid command
702   Invalid operand
713   Directory not found
715   Directory already exists
716   Directory not specified
723   List not found
726   List not specified
728   List is in update mode
729   List is not in update mode
730   User is not signed on
732   Queue empty
733   Named queue not found
736   Stem or variable not specified
737   Stem or variable name too long
738   Stem or variable count invalid
743   Block not found
746   CICGETV error
747   GETMAIN error
748   FREEMAIN error
749   ENQ error
750   DEQ error
751   Dynamic area GETMAIN error
752   Error in saved variable data
753   Saved variable not found
754   User not owner of list

B.1.10 LISTCMD

0   Normal return
821   Invalid environment name
822   Invalid command name

B.1.11 CLD

0   Normal return
923   Error in storing current RLS directory information
924   RLS directory does not exist or access not authorized
925   Error in retrieving directory information
926   Invalid directory
927   Cannot go back past root directory
928   Error setting result value

B.1.12 DEFCMD

0   Normal return
1001   Invalid command
1021   Cannot load program
1023   Entry not found
1048   No client available
1099   Internal error

B.1.13 DEFSCMD

0	Normal return
1101	Invalid command
1121	Cannot load program
1123	Entry not found
1148	No client available
1199	Internal error

B.1.14 DEFTRNID

0	Normal return
1202	Invalid operand
1222	Invalid option
1223	Error storing trantable information
1225	Error retrieving trantable information
1226	Exec name length error
1228	Error setting trantable value
1233	Transaction not found in table

B.1.15 EXECDROP

0	Normal return
1401	Invalid command
1402	Invalid operand
1423	Error storing EXECLOAD information
1425	Error retrieving EXECLOAD information
1448	No client available

B.1.16 EXECLOAD

0	Normal return
1501	Invalid command
1502	Invalid operand
1523	Error storing EXECLOAD information
1525	Error retrieving EXECLOAD information
1530	Unable to link to CICLIBR routine
1531	Error returned from CICLIBR routine
1532	Error returned from RFS READ
1547	GETMAIN error
1548	No client available
1599	Internal error

B.1.17 EXECMAP

0	Normal return
1623	EXECLOAD directory not found

B.1.18 EXPORT and IMPORT

0	Normal return
1701	Invalid command
1702	Invalid operand
1723	RFS write error
1724	RFS read error
1733	Input for export not found
1736	Unexpected CICS error

1738	Invalid VSE Librarian member name
1741	Unsupported record format
1799	Internal error

B.1.19 FILEPOOL

0	Normal return
1802	Invalid operand
1821	Invalid file pool subcommand
1822	Filepool subcommand not specified
1823	Error storing file pool information
1824	File pool ID not specified
1825	Error retrieving file pool information
1826	Invalid file pool ID
1827	Invalid file pool data retrieved
1828	File pool not defined
1829	RFS could not add library to file pool
1830	RFS could not create users directory
1831	CICS file indentifier for the file pool must be specified
1832	Invalid CICS file indentifier
1833	File pool variable corrupted
1834	Pool ID already exists
1835	CICS file indentifier already used
1836	Could not format file pool
1837	File pool needs to be formatted first
1838	File pool ADD record is full
1839	File ID is not found

B.1.20 LISTCLIB and LISTELIB

2226	Invalid stem variable name
2245	Error retrieving authorized library information

B.1.21 GETVERS

0	Normal return
1910	Request failed

B.1.22 COPYR2S

0	Normal return
2002	Invalid operand
2021	Invalid structure definition
2022	Invalid variable structure definition
2023	Field name not found
2025	Failure processing GETVAR request
2026	Invalid numeric input
2027	RFS read error
2028	Invalid offset
2029	Invalid length value

B.1.23 COPYS2R

0	Normal return
2102	Invalid operand
2121	Invalid structure definition
2122	Invalid variable structure definition
2123	Field name not found
2125	Failure processing GETVAR request
2126	Invalid numeric input
2127	RFS read error
2128	Invalid offset
2129	Invalid length value

B.1.24 LISTPOOL

0	Normal return
2225	Error retrieving filepool information
2226	Invalid stem variable name

B.1.25 LISTTRNID

0	Normal return
2325	Error retrieving trantable information

B.1.26 C2S

0	Normal return
2440	No variable name specified
2441	Error retrieving variable
2442	Error storing variable
2448	No client available

B.1.27 PSEUDO

0	Normal return
2502	Invalid operand
2521	Operand not specified

B.1.28 AUTHUSER

0	Normal return
2602	Invalid operand or operand missing
2621	Specified user ID invalid length
2642	Error storing user ID

B.1.29 SETSYS

0	Normal return
2721	Invalid SETSYS subcommand
2722	Error storing variable
2723	Invalid language
2725	Invalid MAXVSTOR operand
2726	Invalid RETRIEVE PFkey operand
2727	Invalid TERMOUT operand
2732	Invalid PSEUDO operand
2735	Invalid VSE Librarian sublibrary name
2739	VARGET for AUTHCLIB or AUTHELIB failed

2740 Too many sublibraries specified

B.1.30 S2C

0 Normal return
2840 No variable name specified
2841 Error retrieving variable
2842 Error storing variable
2848 No client available

B.1.31 TERMID

0 Normal return
2921 Error in obtaining terminal ID
2928 Error setting TERMID value

B.1.32 WAITREAD

0 Normal return
3021 No terminal is attached
3099 Internal error

B.1.33 WAITREQ

0 Normal return
3121 WAITREQ not enabled
3122 Exec not a server
3123 Error saving request variable
3199 Internal error

B.1.34 EXEC

n specifies the return code set by the exit of the called exec
0 Normal return
-3 Exec not found
-10 Exec name not specified
-11 Invalid exec name
-12 GETMAIN error
-99 Internal error

B.1.35 CEDA and CEMT

n specifies the return code passed back by CICS if an error is detected
0 Normal return
-101 Invalid command

B.1.36 EXECIO

n specifies the return code passed back by CICS if an error is detected
0 Normal return
-202 Invalid operand
-221 Too many operands specified
-222 Recno operand out of range
-224 Lines operand invalid

B.1.37 CONVTMAP

n	specifies the return code from the attempt to process the VSE Librarian sublibrary member
0	Normal return
-302	Invalid operand
-321	Invalid input record
-322	RFS error writing output file

B.1.38 SCRNINFO

n	specifies the return code passed back by CICS if an error is detected
0	Normal return
-499	Internal error

B.1.39 CICS

-521	Command not supported
-522	Invalid command or keyword
-523	Option must be specified
-524	Unsupported option specified
-525	Conflicting options specified
-526	Implied option not specified
-527	Redundant specification for option
-528	Value for option not specified
-529	Value specified for option which should not have a value
-530	Value specified for option is not numeric
-531	Invalid value
-532	Value specified is too long
-533	Value specified is too short
-534	Value not specified
-535	Variable table overflow
-536	Number of variables exceeds variable table limit
-537	Argument must be a variable
-538	Variable does not exist
-539	Invalid variable name
-540	Master system trace flag must be on for tracing
-541	Parsing error
-542	Generic name is invalid
-543	Missing right parenthesis
-544	Ambiguous value/keyword
-545	RIDFLD must be fullword variable
-546	RIDFLd must be variable
-547	Invalid GETVAR return code
-548	Internal GETVAR error
-549	Bad PUTVAR return code
-550	PUTVAR failed
-551	Unable to obtain storage
-552	Exec CICS command table not found

Appendix C. Double-Byte Character Set (DBCS) Support

A Double-Byte Character Set supports languages that have more characters than can be represented by 8 bits (such as Korean Hangeul and Japanese kanji). REXX has a full range of DBCS functions and handling techniques.

These include:

- Symbol and string handling capabilities with DBCS characters

- An option that allows DBCS characters in symbols, comments, and literal strings.

- An option that allows data strings to contain DBCS characters.

- A number of functions that specifically support the processing of DBCS character strings

- Defined DBCS enhancements to current instructions and functions.

Note: The use of DBCS does not affect the meaning of the built-in functions as described in Chapter 14, "Functions" on page 201. This explains how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. The appendix describes how the resulting characters are represented as bytes. This internal representation is not usually seen if the results are printed. It may be seen if the results are displayed on certain terminals.

C.1 General Description

The following characteristics help define the rules used by DBCS to represent extended characters:

- Each DBCS character consists of 2 bytes.

- There are no DBCS control characters.

- The codes are within the ranges defined in the table, which shows the valid DBCS code for the DBCS blank. You cannot have a DBCS blank in a simple symbol, in the stem of a compound variable, or in a label.

Table 5. DBCS Ranges

Byte	EBCDIC
1st	X'41' to X'FE'
2nd	X'41' to X'FE'
DBCS blank	X'4040'

- DBCS alphanumeric and special symbols

   A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

   Here are some examples:

```
X'42C1' is an EBCDIC double-byte A
X'4281' is an EBCDIC double-byte a
X'427D' is an EBCDIC double-byte quote
```

- No case translation

  In general, there is no concept of lowercase and uppercase in DBCS.

- Notational conventions

  This appendix uses the following notational conventions:

```
DBCS character            ->    .A .B .C .D
SBCS character            ->    a b c d e
DBCS blank                ->    '. '
EBCDIC shift-out (X'0E')  ->    <
EBCDIC shift-in  (X'0F')  ->    >
```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

C.1.1 Enabling DBCS Data Operations and Symbol Use

The OPTIONS instruction controls how REXX regards DBCS data. To enable DBCS operations, use the EXMODE option. To enable DBCS symbols, use the ETMODE option on the OPTIONS instruction; this must be the first instruction in the program. (See page 178 for more information.)

If OPTIONS ETMODE is in effect, the language processor does validation to ensure that SO and SI are paired in comments. Otherwise, the contents of the comment are not checked. The comment delimiters (/* and */) must be SBCS characters.

C.1.2 Symbols and Strings

In DBCS, there are DBCS-only symbols and strings and mixed symbols and strings.

C.1.2.1 DBCS-Only Symbols and Mixed SBCS/DBCS Symbols

A DBCS-only symbol consists of only non-blank DBCS codes as indicated in Table 5 on page 449.

A mixed DBCS symbol is formed by a concatenation of SBCS symbols, DBCS-only symbols, and other mixed DBCS symbols. In EBCDIC, the SO and SI bracket the DBCS symbols and distinguish them from the SBCS symbols.

The default value of a DBCS symbol is the symbol itself, with SBCS characters translated to uppercase.

A *constant symbol* must begin with an SBCS digit (0–9) or an SBCS period. The delimiter (period) in a compound symbol must be an SBCS character.

C.1.2.2 DBCS-Only Strings and Mixed SBCS/DBCS Strings

A DBCS-only string consists of only DBCS characters. A mixed SBCS/DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI bracket the DBCS data and distinguish it from the SBCS data. Because the SO and SI are needed only in the mixed strings, they are not associated with the DBCS-only strings.

In EBCDIC:

```
DBCS-only string      ->       .A.B.C
Mixed string          ->       ab<.A.B>
Mixed string          ->       <.A.B>
Mixed string          ->       ab<.C.D>ef
```

C.1.3 Validation

The user must follow certain rules and conditions when using DBCS.

C.1.3.1 DBCS Symbol Validation

DBCS symbols are valid only if you comply with the following rules:

- The DBCS portion of the symbol must be an even number of bytes in length
- DBCS alphanumeric and special symbols are regarded as different to their corresponding SBCS characters. Only the SBCS characters are recognized by REXX in numbers, instruction keywords, or operators
- DBCS characters cannot be used as special characters in REXX
- SO and SI cannot be contiguous
- Nesting of SO or SI is not permitted
- SO and SI must be paired
- No part of a symbol consisting of DBCS characters may contain a DBCS blank.
- Each part of a symbol consisting of DBCS characters must be bracketed with SO and SI.

These examples show some possible misuses:

```
<.A.BC>           ->   Incorrect because of odd byte length
<.A.B><.C>        ->   Incorrect contiguous SO/SI
<>                ->   Incorrect contiguous SO/SI (null DBCS symbol)
<.A<.B>.C>        ->   Incorrectly nested SO/SI
<.A.B.C           ->   Incorrect because SO/SI not paired
<.A. .B>          ->   Incorrect because contains blank
'. A<.B><.C>      ->   Incorrect symbol
```

C.1.3.2 Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If you use a mixed string with an instruction, operator, or function that does not allow mixed strings, this causes a **syntax error**.

The following rules must be followed for mixed string validation:

- DBCS strings must be an even number of bytes in length, unless you have SO and SI.

EBCDIC only:

- SO and SI must be paired in a string.
- Nesting of SO or SI is not permitted.

These examples show some possible misuses:

```
'ab<cd'        ->     INCORRECT - not paired
'<.A<.B>.C>    ->     INCORRECT - nested
'<.A.BC>'      ->     INCORRECT - odd byte length
```

The end of a comment delimiter is not found within DBCS character sequences. For example, when the program contains /* < */, then the */ is not recognized as

ending the comment because the scanning is looking for the > (SI) to go with the < (SO) and not looking for */.

When a variable is created, modified, or referred to in a REXX program under OPTIONS EXMODE, it is validated whether it contains a correct mixed string or not. When a referred variable contains a mixed string that is not valid, it depends on the instruction, function, or operator whether it causes a syntax error.

The ARG, PARSE, PULL, PUSH, QUEUE, SAY, TRACE, and UPPER instructions all require valid mixed strings with OPTIONS EXMODE in effect.

C.1.4 Instruction Examples

Here are some examples that illustrate how instructions work with DBCS.

C.1.4.1 PARSE

In EBCDIC:

```
x1 = '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1
        w1   ->   '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 1 w1
        w1   ->   '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1 .
        w1   ->   '<.A.B>'
```

The leading and trailing SO and SI are unnecessary for word parsing and, thus, they are stripped off. However, one pair is still needed for a valid mixed DBCS string to be returned.

```
PARSE VAR x1 . w2
        w2   ->   '<. ><.E><.F><>'
```

Here the first blank delimited the word and the SO is added to the string to ensure the DBCS blank and the valid mixed string.

```
PARSE VAR x1 w1 w2
        w1   ->   '<.A.B>'
        w2   ->   '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
        w1   ->   '<.A.B>'
        w2   ->   '<.E><.F>'
```

The word delimiting allows for unnecessary SO and SI to be dropped.

```
x2 = 'abc<>def <.A.B><><.C.D>'

PARSE VAR x2 w1 '' w2
        w1  ->   'abc<>def <.A.B><><.C.D>'
        w2  ->   ''

PARSE VAR x2 w1 '<>' w2
        w1  ->   'abc<>def <.A.B><><.C.D>'
        w2  ->   ''

PARSE VAR x2 w1 '<><>' w2
        w1  ->   'abc<>def <.A.B><><.C.D>'
        w2  ->   ''
```

Note that for the last three examples '', <>, and <><> are each a null string (a string of length 0). When parsing, the null string matches the end of string. For this reason, w1 is assigned the value of the entire string and w2 is assigned the null string.

C.1.4.2 PUSH and QUEUE

The PUSH and QUEUE instructions add entries to the program stack. Since a stack entry is limited to 255 bytes, the *expression* must be truncated less than 256 bytes. If the truncation splits a DBCS string, REXX will insure that the integrity of the SO-SI pairing will be kept under OPTIONS EXMODE.

C.1.4.3 SAY and TRACE

The SAY and TRACE instructions write data to the output stream. As was true for the PUSH and QUEUE instructions, REXX will guarantee the SO-SI pairs are kept for any data that is separated to meet the requirements of the output stream. The SAY and TRACE instructions display data on the user's terminal. As was true for the PUSH and QUEUE instructions, REXX will guarantee the SO-SI pairs are kept for any data that is separated to meet the requirements of the terminal line size. This is generally 130 bytes or fewer if the DIAG-24 value returns a smaller value.

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string is treated as SBCS data, because 4 is the minimum for mixed string data.

C.1.4.4 UPPER

Under OPTIONS EXMODE, the UPPER instruction translates only SBCS characters in contents of one or more variables to uppercase, but it never translates DBCS characters. If the content of a variable is not valid mixed string data, no uppercasing occurs.

C.2 DBCS Function Handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**—Logical character lengths are used when counting the length of a string (that is, 1 byte for one SBCS logical character, 2 bytes for one DBCS logical character). In EBCDIC, SO and SI are considered to be transparent, and are not counted, for every string operation.

2. **Character extraction from a string**—Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, `.A` and `.B` are extracted from `<.A.B>`, and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO and SI that are between characters are also extracted. For example, `.A><.B` is extracted from `<.A><.B>`, and when the string is finally used as a completed string, the SO prefixes it and the SI suffixes it to give `<.A><.B>`.

Here are some EBCDIC examples:

```
S1 = 'abc<>def'

SUBSTR(S1,3,1)        ->      'c'
SUBSTR(S1,4,1)        ->      'd'
SUBSTR(S1,3,2)        ->      'c<>d'

S2 = '<><.A.B><>'

SUBSTR(S2,1,1)        ->      '<.A>'
SUBSTR(S2,2,1)        ->      '<.B>'
SUBSTR(S2,1,2)        ->      '<.A.B>'
SUBSTR(S2,1,3,'x')    ->      '<.A.B><>x'

S3 = 'abc<><.A.B>'

SUBSTR(S3,3,1)        ->      'c'
SUBSTR(S3,4,1)        ->      '<.A>'
SUBSTR(S3,3,2)        ->      'c<><.A>'
DELSTR(S3,3,1)        ->      'ab<><.A.B>'
DELSTR(S3,4,1)        ->      'abc<><.B>'
DELSTR(S3,3,2)        ->      'ab<.B>'
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI and SO (or SO and SI) that are a result of string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.

4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than an SBCS one if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and trailing contiguous SO and SI (or SI and SO) in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

In EBCDIC, contiguous SO and SI (or SI and SO) between nonblank characters are also removed for comparison.

Note: The strict comparison operators do not cause syntax errors even if you specify mixed strings that are not valid.

In EBCDIC:

```
      '<.A>' = '<.A. >'       ->    1     /* true  */
  '<><><.A>' = '<.A><><>'     ->    1     /* true  */
   '<>  <.A>' = '<.A>'        ->    1     /* true  */
'<.A><><.B>' = '<.A.B>'       ->    1     /* true  */
        'abc' < 'ab<. >'      ->    0     /* false */
```

5. **Word extraction from a string**—"Word" means that characters in a string are delimited by an SBCS or a DBCS blank.

In EBCDIC, leading and trailing contiguous SO and SI (or SI and SO) are also removed when *word*s are separated in a string, but contiguous SO and SI (or SI and SO) in a word are not removed or separated for word operations. Leading and trailing contiguous SO and SI (or SI and SO) of a word are not removed if they are among words that are extracted at the same time.

In EBCDIC:

```
W1 = '<><. .A. . .B><.C. .D><>'

SUBWORD(W1,1,1)      ->      '<.A>'
SUBWORD(W1,1,2)      ->      '<.A. . .B><.C>'
SUBWORD(W1,3,1)      ->      '<.D>'
SUBWORD(W1,3)        ->      '<.D>'


W2 = '<.A. .B><.C><> <.D>'

SUBWORD(W2,2,1)      ->      '<.B><.C>'
SUBWORD(W2,2,2)      ->      '<.B><.C><> <.D>'
```

C.2.1 Built-in Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 14, "Functions" on page 201.

C.2.1.1 ABBREV
In EBCDIC:

```
ABBREV('<.A.B.C>','<.A.B>')      ->      1
ABBREV('<.A.B.C>','<.A.C>')      ->      0
ABBREV('<.A><.B.C>','<.A.B>')    ->      1
ABBREV('aa<>bbccdd','aabbcc')    ->      1
```

Applying the character comparison and character extraction from a string rules.

C.2.1.2 COMPARE
In EBCDIC:

```
COMPARE('<.A.B.C>','<.A.B><.C>')      ->      0
COMPARE('<.A.B.C>','<.A.B.D>')        ->      3
COMPARE('ab<>cde','abcdx')            ->      5
COMPARE('<.A><>','<.A>','<. >')       ->      0
```

Applying the character concatenation for padding, character extraction from a string, and character comparison rules.

C.2.1.3 COPIES
In EBCDIC:

```
COPIES('<.A.B>',2)        ->      '<.A.B.A.B>'
COPIES('<.A><.B>',2)      ->      '<.A><.B.A><.B>'
COPIES('<.A.B><>',2)      ->      '<.A.B><.A.B><>'
```

Applying the character concatenation rule.

C.2.1.4 DATATYPE

```
DATATYPE('<.A.B>')         ->   'CHAR'
DATATYPE('<.A.B>','D')   ->   1
DATATYPE('<.A.B>','C')   ->   1
DATATYPE('a<.A.B>b','D') ->   0
DATATYPE('a<.A.B>b','C') ->   1
DATATYPE('abcde','C')     ->   0
DATATYPE('<.A.B','C')     ->   0
DATATYPE('<.A.B>','S')   ->   1       /* if ETMODE is on */
```

Note: If *string* is not a valid mixed string and `C` or `D` is specified as *type*, `0` is returned.

C.2.1.5 FIND

```
FIND('<.A. .B.C> abc','<.B.C> abc')    ->   2
FIND('<.A. .B><.C> abc','<.B.C> abc')  ->   2
FIND('<.A. . .B> abc','<.A> <.B>')     ->   1
```

Applying the word extraction from a string and character comparison rules.

C.2.1.6 INDEX, POS, and LASTPOS

```
INDEX('<.A><.B><><.C.D.E>','<.D.E>')    ->   4
POS('<.A>','<.A><.B><><.A.D.E>')        ->   1
LASTPOS('<.A>','<.A><.B><><.A.D.E>')    ->   3
```

Applying the character extraction from a string and character comparison rules.

C.2.1.7 INSERT and OVERLAY

In EBCDIC:

```
INSERT('a','b<><.A.B>',1)             -> 'ba<><.A.B>'
INSERT('<.A.B>','<.C.D><>',2)         -> '<.C.D.A.B><>'
INSERT('<.A.B>','<.C.D><><.E>',2)     -> '<.C.D.A.B><><.E>'
INSERT('<.A.B>','<.C.D><>',3,,'<.E>') -> '<.C.D><.E.A.B>'

OVERLAY('<.A.B>','<.C.D><>',2)         -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',2)     -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',3)     -> '<.C.D><><.A.B>'
OVERLAY('<.A.B>','<.C.D><>',4,,'<.E>') -> '<.C.D><.E.A.B>'
OVERLAY('<.A>','<.C.D><.E>',2)         -> '<.C.A><.E>'
```

Applying the character extraction from a string and character comparison rules.

C.2.1.8 JUSTIFY

```
JUSTIFY('<><. .A. . .B><.C. .D>',10,'p')
                 ->   '<.A>ppp<.B><.C>ppp<.D>'
JUSTIFY('<><. .A. . .B><.C. .D>',11,'p')
                 ->   '<.A>pppp<.B><.C>ppp<.D>'
JUSTIFY('<><. .A. . .B><.C. .D>',10,'<.P>')
                 ->   '<.A.P.P.P.B><.C.P.P.P.D>'
JUSTIFY('<><.X. .A. . .B><.C. .D>',11,'<.P>')
                 ->   '<.X.P.P.A.P.P.B><.C.P.P.D>'
```

Applying the character concatenation for padding and character extraction from a string rules.

C.2.1.9 LEFT, RIGHT, and CENTER
In EBCDIC:

```
LEFT('<.A.B.C.D.E>',4)      ->    '<.A.B.C.D>'
LEFT('a<>',2)               ->    'a<> '
LEFT('<.A>',2,'*')          ->    '<.A>*'
RIGHT('<.A.B.C.D.E>',4)     ->    '<.B.C.D.E>'
RIGHT('a<>',2)              ->    ' a'
CENTER('<.A.B>',10,'<.E>')  ->    '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>',11,'<.E>')  ->    '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>',10,'e')     ->    'eeee<.A.B>eeee'
```

Applying the character concatenation for padding and character extraction from a string rules.

C.2.1.10 LENGTH
In EBCDIC:

```
LENGTH('<.A.B><.C.D><>')        ->    4
```

Applying the counting characters rule.

C.2.1.11 REVERSE
In EBCDIC:

```
REVERSE('<.A.B><.C.D><>')       ->    '<><.D.C><.B.A>'
```

Applying the character extraction from a string and character concatenation rules.

C.2.1.12 SPACE
In EBCDIC:

```
SPACE('a<.A.B. .C.D>',1)         ->    'a<.A.B> <.C.D>'
SPACE('a<.A><><. .C.D>',1,'x')   ->    'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>')  ->    'a<.A.E.C.D>'
```

Applying the word extraction from a string and character concatenation rules.

C.2.1.13 STRIP
In EBCDIC:

```
STRIP('<><.A><.B><.A><>',,'<.A>')    ->    '<.B>'
```

Applying the character extraction from a string and character concatenation rules.

C.2.1.14 SUBSTR and DELSTR
In EBCDIC:

```
SUBSTR('<><.A><><.B><.C.D>',1,2)  ->  '<.A><><.B>'
DELSTR('<><.A><><.B><.C.D>',1,2)  ->  '<><.C.D>'
SUBSTR('<.A><><.B><.C.D>',2,2)    ->  '<.B><.C>'
DELSTR('<.A><><.B><.C.D>',2,2)    ->  '<.A><><.D>'
SUBSTR('<.A.B><>',1,2)            ->  '<.A.B>'
SUBSTR('<.A.B><>',1)              ->  '<.A.B><>'
```

Applying the character extraction from a string and character concatenation rules.

C.2.1.15 SUBWORD and DELWORD
In EBCDIC:

```
SUBWORD('<><. .A. . .B><.C. .D>',1,2) -> '<.A. . .B><.C>'
DELWORD('<><. .A. . .B><.C. .D>',1,2) -> '<><. .D>'
SUBWORD('<><.A. . .B><.C. .D>',1,2)   -> '<.A. . .B><.C>'
DELWORD('<><.A. . .B><.C. .D>',1,2)   -> '<><.D>'
SUBWORD('<.A. .B><.C><> <.D>',1,2)     -> '<.A. .B><.C>'
DELWORD('<.A. .B><.C><> <.D>',1,2)     -> '<.D>'
```

Applying the word extraction from a string and character concatenation rules.

C.2.1.16 SYMBOL
In EBCDIC:

```
Drop A.3  ;  <.A.B>=3      /* if ETMODE is on */

SYMBOL('<.A.B>')   ->    'VAR'
SYMBOL(<.A.B>)     ->    'LIT' /* has tested "3" */
SYMBOL('a.<.A.B>') ->    'LIT' /* has tested A.3 */
```

C.2.1.17 TRANSLATE
In EBCDIC:

```
TRANSLATE('abcd','<.A.B.C>','abc')        -> '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','abc')      -> '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','ab<>c')    -> '<.A.B.C>d'
TRANSLATE('a<>bcd','<><.A.B.C>','ab<>c')  -> '<.A.B.C>d'
TRANSLATE('a<>xcd','<><.A.B.C>','ab<>c')  -> '<.A>x<.C>d'
```

Applying the character extraction from a string, character comparison, and character concatenation rules.

C.2.1.18 VALUE
In EBCDIC:

```
Drop A3  ;  <.A.B>=3  ;  fred='<.A.B>'

VALUE('fred')       ->    '<.A.B>' /* looks up FRED   */
VALUE(fred)         ->    '3'      /* looks up <.A.B> */
VALUE('a'<.A.B>)    ->    'A3'     /* if ETMODE is on */
```

C.2.1.19 VERIFY
In EBCDIC:

```
VERIFY('<><><.A.B><><.X>','<.B.A.C.D.E>')   -> 3
```

Applying the character extraction from a string and character comparison rules.

C.2.1.20 WORD, WORDINDEX, and WORDLENGTH
In EBCDIC:

```
W = '<><. .A. . .B><.C. .D>'

WORD(W,1)           ->     '<.A>'
WORDINDEX(W,1)      ->     2
WORDLENGTH(W,1)     ->     1

Y = '<><.A. . .B><.C. .D>'

WORD(Y,1)           ->     '<.A>'
WORDINDEX(Y,1)      ->     1
WORDLENGTH(Y,1)     ->     1

Z = '<.A  .B><.C> <.D>'

WORD(Z,2)           ->     '<.B><.C>'
WORDINDEX(Z,2)      ->     3
WORDLENGTH(Z,2)     ->     2
```

Applying the word extraction from a string and (for WORDINDEX and WORDLENGTH) counting characters rules.

C.2.1.21 WORDS

In EBCDIC:

```
W = '<><. .A. . .B><.C. .D>'

WORDS(W)            ->     3
```

Applying the word extraction from a string rule.

C.2.1.22 WORDPOS

In EBCDIC:

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc')          ->   2
WORDPOS('<.A.B>','<.A.B. .A.B><. .B.C. .A.B>',3) ->   4
```

Applying the word extraction from a string and character comparison rules.

C.3 DBCS Processing Functions

This section describes the functions that support DBCS mixed strings. These functions handle mixed strings regardless of the OPTIONS mode.

Note: When used with DBCS functions, *length* is always measured in bytes (as opposed to LENGTH(*string*), which is measured in characters).

C.3.1 Counting Option

In EBCDIC, when specified in the functions, the counting option can control whether the SO and SI are considered present when determining the length. **Y** specifies counting SO and SI within mixed strings. **N** specifies *not* to count the SO and SI, and is the default.

C.4 Function Descriptions

The following are the DBCS functions and their descriptions.

C.4.1 DBADJUST

```
►►──DBADJUST(string──────────────)──────────────────────────►◄
                      └─,operation─┘
```
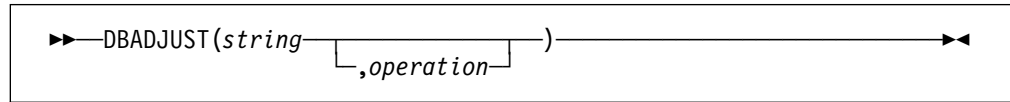
In EBCDIC, adjusts all contiguous SI and SO (or SO and SI) characters in *string* based on the *operation* specified. The following are valid *operation*s. Only the capitalized and highlighted letter is needed; all characters following it are ignored.

Blank   changes contiguous characters to blanks (X'4040').

Remove   removes contiguous characters, and is the default.

Here are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b','B')     ->     '<.A. .B>a  b'
DBADJUST('<.A><.B>a<>b','R')     ->     '<.A.B>ab'
DBADJUST('<><.A.B>','B')         ->     '<. .A.B>'
```

C.4.2 DBBRACKET

```
►►──DBBRACKET(string)────────────────────────────────────────►◄
```
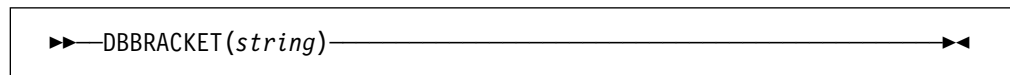
In EBCDIC, adds SO and SI brackets to a DBCS-only string. If *string* is not a DBCS-only string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some EBCDIC examples:

```
DBBRACKET('.A.B')      ->     '<.A.B>'
DBBRACKET('abc')       ->     SYNTAX error
DBBRACKET('<.A.B>')    ->     SYNTAX error
```

C.4.3 DBCENTER
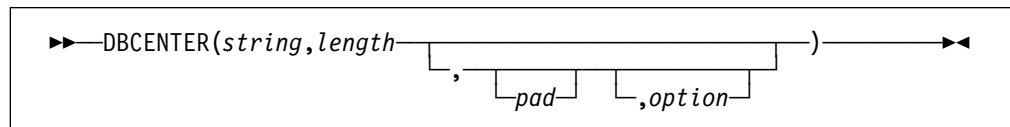
```
►►──DBCENTER(string,length──────────────────────────)────────►◄
                          └─,──────────────────────┘
                             └─pad─┘  └─,option─┘
```

returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is a blank. If *string* is longer than *length*, it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.
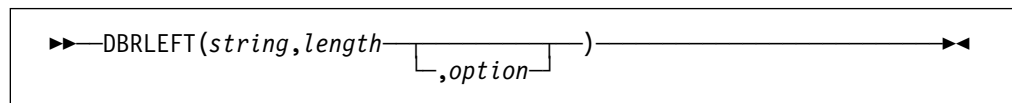
The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBCENTER('<.A.B.C>',4)              ->      ' <.B> '
DBCENTER('<.A.B.C>',3)              ->      ' <.B>'
DBCENTER('<.A.B.C>',10,'x')         ->      'xx<.A.B.C>xx'
DBCENTER('<.A.B.C>',10,'x','Y')     ->      'x<.A.B.C>x'
DBCENTER('<.A.B.C>',4,'x','Y')      ->      '<.B>'
DBCENTER('<.A.B.C>',5,'x','Y')      ->      'x<.B>'
DBCENTER('<.A.B.C>',8,'<.P>')       ->      ' <.A.B.C> '
DBCENTER('<.A.B.C>',9,'<.P>')       ->      ' <.A.B.C.P>'
DBCENTER('<.A.B.C>',10,'<.P>')      ->      '<.P.A.B.C.P>'
DBCENTER('<.A.B.C>',12,'<.P>','Y')  ->      '<.P.A.B.C.P>'
```

C.4.4 DBCJUSTIFY

```
►►──DBCJUSTIFY(string,length──────────────────────)──────►◄
                            └─,───────────┬──────────┘
                                └─pad─┘  └─,option─┘
```
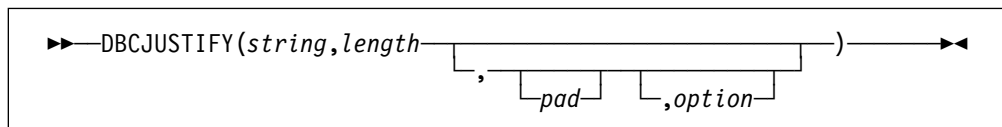
formats *string* by adding *pad* characters between nonblank characters to justify to both margins and length of bytes *length* (*length* must be nonnegative). Rules for adjustments are the same as for the JUSTIFY function. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some examples:

```
DBCJUSTIFY('<><AA  BB><CC>',20,,'Y')
                            ->  '<AA>      <BB>      <CC>'

DBCJUSTIFY('<>< AA    BB>< CC>',20,'<XX>','Y')
                            ->  '<AAXXXXXXXBBXXXXXXCC>'

DBCJUSTIFY('<>< AA    BB>< CC>',21,'<XX>','Y')
                            ->  '<AAXXXXXXXBBXXXXXXCC> '

DBCJUSTIFY('<>< AA    BB>< CC>',11,'<XX>','Y')
                            ->  '<AAXXXXXBB> '

DBCJUSTIFY('<>< AA    BB>< CC>',11,'<XX>','N')
                            ->  '<AAXXBBXXCC> '
```

C.4.5 DBLEFT
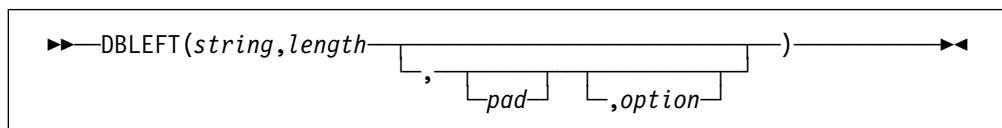
```
►►──DBLEFT(string,length──────────────────────)──────►◄
                        └─,───────────┬──────────┘
                            └─pad─┘  └─,option─┘
```

returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBLEFT('ab<.A.B>',4)              ->      'ab<.A>'
DBLEFT('ab<.A.B>',3)              ->      'ab '
DBLEFT('ab<.A.B>',4,'x','Y')      ->      'abxx'
DBLEFT('ab<.A.B>',3,'x','Y')      ->      'abx'
DBLEFT('ab<.A.B>',8,'<.P>')       ->      'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>')       ->      'ab<.A.B.P> '
DBLEFT('ab<.A.B>',8,'<.P>','Y')   ->      'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>','Y')   ->      'ab<.A.B> '
```

C.4.6 DBRIGHT
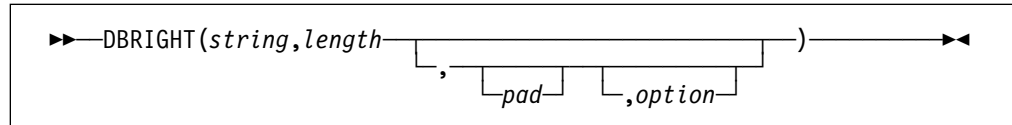
```
►►──DBRIGHT(string,length─┬────────────────────────┬──)──────────►◄
                          └─,─┬──────┬─┬──────────┬─┘
                             └─pad─┘ └─,option─┘
```

returns a string of length *length* containing the rightmost *length* characters of *string*.
The string returned is padded with *pad* characters (or truncated) on the left as
needed. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as
one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRIGHT('ab<.A.B>',4)              ->      '<.A.B>'
DBRIGHT('ab<.A.B>',3)              ->      ' <.B>'
DBRIGHT('ab<.A.B>',5,'x','Y')      ->      'x<.B>'
DBRIGHT('ab<.A.B>',10,'x','Y')     ->      'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>')       ->      '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>')       ->      ' <.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y')   ->      'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y')  ->      '   ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y')  ->      '<.P>ab<.A.B>'
```

C.4.7 DBRLEFT

```
►►──DBRLEFT(string,length─┬──────────┬──)──────────────────────────►◄
                          └─,option─┘
```

returns the remainder from the DBLEFT function of *string*. If *length* is greater than
the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as
one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRLEFT('ab<.A.B>',4)              ->      '<.B>'
DBRLEFT('ab<.A.B>',3)              ->      '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')          ->      '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')          ->      '<.A.B>'
DBRLEFT('ab<.A.B>',8)              ->      ''
DBRLEFT('ab<.A.B>',9,'Y')          ->      ''
```

C.4.8 DBRRIGHT

```
►►──DBRRIGHT(string,length──────────)────────────────►◄
                         └─,option─┘
```

returns the remainder from the DBRIGHT function of *string*. If *length* is greater
than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as
one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBRRIGHT('ab<.A.B>',4)            ->    'ab'
DBRRIGHT('ab<.A.B>',3)            ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5)            ->    'a'
DBRRIGHT('ab<.A.B>',4,'Y')        ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')        ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',8)            ->    ''
DBRRIGHT('ab<.A.B>',8,'Y')        ->    ''
```

C.4.9 DBTODBCS

```
►►──DBTODBCS(string)────────────────────────────────►◄
```

converts all passed, valid SBCS characters (including the SBCS blank) within *string*
to the corresponding DBCS equivalents. Other single-byte codes and all DBCS
characters are not changed. In EBCDIC, SO and SI brackets are added and
removed where appropriate.

Here are some EBCDIC examples:

```
DBTODBCS('Rexx 1988')      ->    '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')      ->    '<.A. .B>'
```

Note: In these examples, the .x is the DBCS character corresponding to an SBCS
x.

C.4.10 DBTOSBCS

```
►►──DBTOSBCS(string)────────────────────────────────►◄
```

converts all passed, valid DBCS characters (including the DBCS blank) within *string*
to the corresponding SBCS equivalents. Other DBCS characters and all SBCS
characters are not changed. In EBCDIC, SO and SI brackets are removed where
appropriate.

Here are some EBCDIC examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>')    ->    'Sd/2-1'
DBTOSBCS('<.X. .Y>')           ->    '<.X> <.Y>'
```

Note: In these examples, the `.d` is the DBCS character corresponding to an SBCS `d`. But the `.X` and `.Y` do not have corresponding SBCS characters and are not converted.

C.4.11 DBUNBRACKET

```
►►──DBUNBRACKET(string)──────────────────────────────►◄
```

In EBCDIC, removes the SO and SI brackets from a DBCS-only *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')      ->    '.A.B'
DBUNBRACKET('ab<.A>')      ->    SYNTAX error
```

C.4.12 DBVALIDATE

```
►►──DBVALIDATE(string────────────)──────────────────────►◄
                    └─,'C'─┘
```

returns `1` if the *string* is a valid mixed string or SBCS string. Otherwise, returns `0`. Mixed string validation rules are:

1. Only valid DBCS character codes

2. DBCS string is an even number of bytes in length

3. EBCDIC only — Proper SO and SI pairing.

In EBCDIC, if **C** is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on (that is, in EBCDIC, the leftmost byte range is from X'41' to X'FE').

Here are some EBCDIC examples:

```
z='abc<de'

DBVALIDATE('ab<.A.B>')       ->    1
DBVALIDATE(z)                ->    0

y='C1C20E111213140F'X

DBVALIDATE(y)                ->    1
DBVALIDATE(y,'C')            ->    0
```

C.4.13 DBWIDTH

```
►►──DBWIDTH(string──────────)──────────────────────────►◄
                  └─,option─┘
```

returns the length of *string* in bytes.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>','Y')    ->    8
DBWIDTH('ab<.A.B>','N')    ->    6
```

DBCS Support

Appendix D. Reserved Keywords and Special Variables

You can use keywords as ordinary symbols in many situations where there is no ambiguity. The precise rules are given here.

There are three special variables: RC, RESULT, and SIGL.

D.1 Reserved Keywords

The free syntax of REXX implies that some symbols are reserved for the language processor's use in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE in a DO instruction and the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an = or : are checked to see if they are instruction keywords. You can use the symbols freely elsewhere in clauses without their being taken as keywords.

It is not, however, recommended for users to run host commands or commands with the same name as REXX keywords (QUEUE, for example). This can create problems for any programmer whose REXX programs might be used for some time and in circumstances outside his or her control, and who wishes to make the program absolutely *watertight*.

In this case, a REXX program may be written with (at least) the first words in command lines enclosed in quotation marks.

Example:

```
'SCRNINFO'
```

This also has the advantage of being more efficient, and, with this style, you can use the SIGNAL ON NOVALUE condition to check the integrity of an exec.

An alternative strategy is to precede such command strings with two adjacent quotation marks, which concatenates the null string on to the front.

Example:

```
''SCRNINFO
```

A third option is to enclose the entire expression (or the first symbol) in parentheses.

Example:

```
(SCRNINFO)
```

More important, the choice of strategy (if it is to be done at all) is a personal one by the programmer. The REXX language does not impose it.

D.2 Special Variables

There are three special variables that the language processor can set automatically:

RC is set to the return code from any run host command (or subcommand). Following the SIGNAL events, SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number (see appendix on error messages, page 429) or the command return code. RC is unchanged following a NOVALUE or HALT event.

> **Note:** Host commands run manually from debug mode do not cause the value of RC to change.

RESULT is set by a RETURN instruction in a subroutine that has been called if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL contains the line number of the clause currently executing when the last transfer of control to a label took place. (A SIGNAL, a CALL, an internal function invocation, or a trapped error condition could cause this.)

None of these variables has an initial value. You can alter them, just as with any other variable, and the PROCEDURE and DROP instructions affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name that the program was called and the source of the program (which is available using the PARSE SOURCE instruction—see page 181). PARSE SOURCE output consists of the string CICS followed by the call type, the name of the exec in uppercase, the name of the file, and library member being run. These are followed by the name by which the program was called and the initial (default) command environment.

In addition, PARSE VERSION (see page 182) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and environment name, respectively.

Finally, you can obtain the current settings of the NUMERIC function by using the DIGITS, FORM, and FUZZ built-in functions.

Appendix E. Debug Aids

This appendix describes the interactive debugging of problems, interrupting execution, and controlling tracing.

E.1 Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program.

Changing the TRACE action to one with a prefix **?** (for example, `TRACE ?A` or the TRACE built-in function) turns on interactive debug and indicates to the user that interactive debug is active. Further TRACE instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see the following for the exceptions). When the language processor pauses, indicated by a READ in the lower corner of the screen, three debug actions are available:

1. **Entering a null line** (with no characters, even blanks) makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For `TRACE ?A`, for example, this is equivalent to single-stepping through the program.

2. **Entering an equal sign (=)** with no blanks makes the language processor rerun the clause last traced. For example: if an IF clause is about to take the wrong branch, you can change the value of the variable(s) on which it depends, and then rerun it.

   Once the clause has been rerun, the language processor pauses again.

3. **Anything else entered** is treated as a **line** of one or more clauses, and processed immediately (that is, as though `DO`; `line`; `END`; had been inserted in the program). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always run (that is, are not affected by the prefix **!** on TRACE instructions), but the variable RC is not set.

   Once the string has been processed, the language processor pauses again for further debug input unless a TRACE instruction was entered. In this latter case, the language processor immediately alters the tracing action (if necessary) and then continues executing until the next pause point (if any). To alter the tracing action (from All to Results, for example) and then rerun the instruction, you must use the built-in function TRACE (see page 230). For example, `CALL TRACE I` changes the trace action to **I** and allows re-execution of the statement after which the pause was made. Interactive debug is turned off, when it is in effect, if a TRACE instruction uses a prefix, or at any time when a `TRACE 0` or `TRACE` with no options is entered.

   You can use the numeric form of the TRACE instruction to allow sections of the program to be run without pause for debug input. `TRACE n` (that is, positive result) allows execution to continue, skipping the next n pauses (when

interactive debug is or becomes active).   `TRACE -n` (that is, negative result) allows execution to continue without pause and with tracing inhibited for `n` clauses that would otherwise be traced.

The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using `TRACE ?R` to trace Results) and then enter a subroutine in which you have no interest, you can enter `TRACE 0` to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a `TRACE ?R` instruction at its start. Having traced the routine, the original status of tracing is restored and (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

Since any instructions may be run in interactive debug, you have considerable control over execution.

Some examples:

```
Say expr     /* displays the result of evaluating the   */
             /* expression.                             */

name=expr    /* alters the value of a variable.         */

Trace 0      /* (or Trace with no options) turns off    */
             /* interactive debug and all tracing.      */

Trace ?A     /* turns off interactive debug but continues */
             /* tracing all clauses.                    */

Trace L      /* makes the language processor pause at labels */
             /* only.  This is similar to the traditional   */
             /* "breakpoint" function, except that you  */
             /* do not have to know the exact name and  */
             /* spelling of the labels in the program.  */

exit         /* stops execution of the program.         */

Do i=1 to 10; say stem.i; end
             /* displays ten elements of the array stem. */
```

Note: While in interactive debug, pauses may occur because of PULL statements as well as because of interactive debug. For programs containing PULL statements, it is important to be aware of the reason for each pause. In programs, PULL statements are often paired with SAY statements. The user should enter the data for the PULL at the pause after the trace line for the PULL (the pause specifically for entering data for the PULL). The user should not enter the data at the pause after the corresponding SAY statement (this is an interactive debug pause). **Exceptions**: Some clauses cannot safely be re-run, and, therefore, the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.

- All END clauses (not a useful place to pause in any case).

- All THEN, ELSE, OTHERWISE, or null clauses.

- All RETURN and EXIT clauses.

- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced).

- Any clause that raises a condition that CALL ON or SIGNAL ON traps (the pause takes place after the target label for the CALL or SIGNAL has been traced).

- Any clause that causes a syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-run.)

E.2 Interrupting Execution and Controlling Tracing

You can use standard CICS facilities to interrupt a REXX exec (the REXX transaction). If properly authorized, you can issue the CEMT SET TASK PURGE command to halt an exec. Refer to the *CICS Transaction Server for VSE/ESA CICS Supplied Transactions* for more information.

Debug Aids

Appendix F. REXX/CICS Business Value Discussion

CICS Transaction Server for VSE/ESA REXX provides an ideal system to deliver superior, valuable, and appropriate CICS-based business solutions in a more timely and cost-effective manner.

F.1 Business Solutions

The ability to deliver business solutions more quickly is an important advantage in today's competitive marketplace.

- *CICS Transaction Server for VSE/ESA REXX is a simple, uniform, self-contained development environment*

  CICS Transaction Server for VSE/ESA REXX supports development directly under CICS and provides much of what the average CICS developer needs in one manageable package.

  REXX is a high-level language that is natural to use and frees the programmer from unnecessary detail. REXX programs tend to be shorter and easier to follow than programs written in other languages. To use REXX for CICS Transaction Server for VSE/ESA REXX a new programmer does not have to learn JCL, COBOL or significant technical detail of CICS (such as the CICS translator).

- *CICS Transaction Server for VSE/ESA REXX allows solutions to be delivered quickly*

  These programs enable gains in application productivity and may reduce delivery time. The REXX language boosts application productivity because of its high level, simplicity, strong parsing, "naturalness," and interpretive support. The interpreter provides a reduced development cycle and has excellent source-level interactive debugging capability.

- *CICS Transaction Server for VSE/ESA REXX makes incremental development a reality*

  Larger, more sophisticated development efforts are feasible and productivity improvements can be achieved by using the powerful interactive source-level debugging capability and the fast development cycle of the REXX interpreter. The interpreter, with its fast, natural development cycle and excellent source-based interactive debugging, better enables the switch to an Incremental Development Methodology (also known as Prototyping Development Methodology).

  With this methodology, REXX can be used for quick and expressive prototyping. Because of the special performance techniques used and the robustness of the language, REXX is also suitable for professional application development.

  CICS Transaction Server for VSE/ESA REXX provides an ideal environment where prototypes can be developed quickly to test system feasibility and get user involvement. The prototypes can then be "grown" into useful production systems.

  Prototyping reduces the possibility of finding out late in the development cycle that the project is not technically feasible or cannot deliver what the customers

want. An additional benefit of incremental development is that the code is tested more thoroughly during development and may result in much higher quality.

- *CICS Transaction Server for VSE/ESA REXX applications are manageable and easy to maintain and support*.

  High-level, REXX-based applications are usually smaller than comparable applications in other languages (in lines of code) and are easier to read and understand. The interactive source-level debugging capability of the REXX interpreter simplifies problem determination and resolution, making REXX-based applications less expensive to support and enhance.

  CICS Transaction Server for VSE/ESA REXX organizes (breaks down) complex systems and makes them more manageable. REXX provides a natural building block approach made up of execs, application macros, and subcommands implemented transparently in a variety of languages. Closely related is the built-in client/server computing support that encourages greater host involvement in the enterprise-wide client/server distributed computing model. Another benefit of ESA/VSE is the facilities it has for integrating multiple applications, products, and system facilities into one seamless package, which can simplify systems development efforts.

- *CICS Transaction Server for VSE/ESA REXX is useable by business people*

  Quite often, business people who best understand the business and their required solutions have ideas on modifying or enhancing the applications they use. However, without extensive training and experience with a programming language, they may be unable to implement the ideas. One of the greatest strengths of REXX is its simplicity and "naturalness" on the one hand, and its powerful capabilities on the other. CICS Transaction Server for VSE/ESA REXX makes it possible, in some cases, for CICS application users to customize and even extend applications without expending scarce programmer resources. This allows more effective tailoring of applications to individual business needs.

  CICS Transaction Server for VSE/ESA REXX facilitates the use of a prototyping methodology. The users of an application under development can participate very closely in the application development process. The end result is that those who have the best understanding of the business and their own needs can better ensure that the application solution delivered matches their needs. This close involvement has the added benefit of addressing human factor needs (useability) earlier in the application development cycle.

- *CICS Transaction Server for VSE/ESA REXX facilitates systems management*

  One of the major uses of REXX is as a Procedures (Scripting) Language. CICS Transaction Server for VSE/ESA REXX can be used to automate sequences of CICS system and application systems management activities, providing greater productivity and reliability.

  Since CICS Transaction Server for VSE/ESA REXX supports application development (and testing) directly under CICS, systems management can be simplified.

- *CICS Transaction Server for VSE/ESA REXX supports six languages*

  CICS Transaction Server for VSE/ESA REXX provides REXX messages in six languages, in addition to U.S. English:

  - Canadian French
  - French
  - German
  - Japanese Kanji
  - Spanish
  - Simplified Chinese.

  Only the REXX messages are translated. The text editor and file list utility are not translated.

  The translated messages files are included on the product tape and are not separate features.

F.2 Product Positioning

The IBM CICS computing environment is one of the largest concentrations of customer production applications and data in the world. There has been tremendous customer investment in CICS-based mainframe systems, CICS-based application development, data collection for CICS-based systems, and employee education relating to the use and support of CICS-based systems. CICS Transaction Server for VSE/ESA REXX helps preserve and enhance the usefulness of this investment.

Not only does CICS Transaction Server for VSE/ESA REXX enhances the delivery of traditional CICS-based production applications, it makes the CICS environment suitable for a broader range of information processing activities. With CICS Transaction Server for VSE/ESA REXX, it is now practical to also perform end-user computing, prototyping, and application development directly within the CICS environment, either in separate or common regions.

Appendix G. System Definition/Customization/Administration

This appendix discusses the system definition, customization, and administration of REXX/CICS.

G.1 Authorized REXX/CICS Commands/Authorized Command Options

Several REXX/CICS commands, or command options, are identified as being authorized. An authorized REXX/CICS command can only be executed if:

- The user ID issuing the exec is an authorized REXX/CICS user. Authorized users are defined by the AUTHUSER command.

- The exec was loaded from a REXX/CICS authorized sublibrary. An authorized sublibrary is a VSE Librarian sublibrary specified on a SETSYS AUTHCLIB or a SETSYS AUTHELIB command.

This rule applies regardless of whether or not the exec attempting the command was issued by an authorized exec.

G.2 System Profile Exec

A system profile exec, named CICSTART, is issued before the first user exec is run after a CICS system restart. Usually, the system profile exec contains system customization commands, authorized sublibrary definitions, authorized user definitions, and authorized command definitions that must reside in an authorized REXX/CICS sublibrary.

The CICSTART exec is loaded from the first occurrence of the member CICSTART.PROC in the LIBDEF PROC search chain for the CICS partition.

Note: The sublibrary in which the member CICSTART.PROC is found is treated as the initial authorized REXX/CICS sublibrary. This can then be changed using the SETSYS AUTHCLIB and SETSYS AUTHELIB commands.

G.3 Authorized REXX/CICS VSE Librarian sublibraries

The VSE Librarian sublibraries specified on the SETSYS AUTHCLIB and SETSYS AUTHELIB commands are considered REXX/CICS authorized sublibraries. If multiple sublibraries are specified, they are searched in the order specified in the command.

The sublibrary in the LIBDEF PROC search chain for the CICS partition containing the member CICSTART.PROC used to initialize REXX/CICS is treated as the initial authorized sublibrary. This can then be changed by using the SETSYS AUTHCLIB and SETSYS AUTHELIB commands.

Notes:

1. Users can cause dynamic allocation of their own VSE Librarian sublibraries by using the REXX/CICS PATH command.

2. Any user can run an exec from a sublibrary specified in the SETSYS AUTHELIB command and any exec loaded from one of these sublibraries can use REXX/CICS authorized commands.

G.4 Defining Authorized Users

Users may be specified as authorized by the AUTHUSER command. It is recommended that all AUTHUSER commands be placed in the CICSTART exec, or in an exec issued from within the CICSTART exec.

G.5 Setting System Options

System options are specified by using the REXX/CICS SETSYS command. It is recommended that system-wide SETSYS commands be placed in the CICSTART exec.

G.6 Defining and Initializing a REXX File System (RFS) File Pool

Use the FILEPOOL DEFINE command to define a RFS file pool. Use the FILEPOOL FORMAT command to initialize the first file in each file pool.

G.7 Adding Files to a REXX File System (RFS) File Pool

Use the FILEPOOL ADD command to add a VSAM file to a REXX File System file pool.

G.8 RFS File Sharing Authorization

Use the RFS AUTH command to specify file sharing permission. Normally, you can allow sharing of resources that you own. As an authorized REXX/CICS user you can specify permission for the sharing of any RFS directories that you have created.

G.9 Creating a PLT Entry for CICSTART

The CICSTART exec can be issued immediately after CICS system initialization by creating a CICS program load table (PLT) entry to invoke the CICREXD program. Otherwise, the first REXX/CICS user after region startup will cause the CICSTART exec to be run.

G.10 Security Exit

This section describes replaceable security exit CICSECX2. IBM provides a sample assembler CICSECX2 exit for customers to customize or replace.

Note: This exit must reside in the same region as REXX/CICS (for example: the use of distributed program link is not allowed).

G.10.1 CICSECX2

CICSECX2 is a REXX File System access security exit.

G.10.1.1 Parameters

The COMMAREA contains the following on input to the exit.

Parameter	Number of Bytes	Datatype	Description
1	4	fullword	Return code
2	8	character	CICS sign on ID
3	1	character	Function requested
4	3		Reserved for IBM use
5	4	fullword	Address of fully qualified RFS file ID string
6	4	fullword	Length of RFS file ID string

G.10.1.2 Return Codes

0   Function request allowed

¬0   Not authorized

G.10.1.3 Function IDs

A   Alter

R   Read

U   Update

Appendix H. Security

REXX/CICS can be viewed as a more sophisticated version of the CICS-supplied Command Level Interpreter Transaction (CECI). The REXX transaction (used to issue REXX execs), much like the CECI transaction, can be controlled using CICS transaction security. The REXX transaction might be made widely available, or might be limited to a few individuals, depending upon the nature of the CICS region it is running in.

Note: The REXX transaction is not required to execute existing REXX execs, but is required if users or programmers want the ability to create or modify REXX execs, and then test them.

H.1 REXX/CICS Supports Multiple Transaction Identifiers

REXX/CICS supports the ability to associate transaction identifiers (TRANIDs), other than REXX, with the REXX/CICS support program. In this case, the name of the REXX exec that is issued is determined by a previous DEFTRNID command. This gives you the ability to still use transaction security with REXX on an exec by exec basis.

H.2 REXX/CICS File Security

Access at the RFS directory level is controlled with the RFS AUTH command and the RFS replaceable security exit.

H.3 ESA/VSE Command Level Security

In some situations, current software practices limit the effectiveness of relying on CICS resource security alone. For additional security control, REXX/CICS was designed with the concept of command level security. Because most facilities under REXX/CICS are accessed as commands, command level security can be used to control access to CICS (and other product or system) facilities. For example, VSAM file access is accomplished through the READ, WRITE, and REWRITE commands.

REXX/command level security is controlled by the DEFSCMD and DEFCMD AUTH parameter and by the provision of authorized REXX/CICS library support.

Command execution security controls the use of certain REXX/CICS commands, or command keywords. In general, this is accomplished by the designation of certain commands (or command options) as authorized. Such command designation is accomplished by the DEFCMD and DEFSCMD commands. For authorized commands to execute properly, they must either be:

1. Executed from an exec loaded from a VSE Librarian sublibrary specified on a SETSYS AUTHCLIB or SETSYS AUTHELIB command.

2. Executed by an authorized user. A user can be authorized by the AUTHUSER command.

H.4 REXX/CICS Authorized Command Support

Any REXX/CICS command can be identified as authorized by a REXX/CICS Systems Administrator. Authorized commands can only be successfully executed in an exec that is issued by an authorized REXX/CICS user or that was loaded from an authorized REXX/CICS sublibrary. Only authorized REXX/CICS users have access to the commands and execs in the "authorized command" sublibraries specified on the SETSYS AUTHCLIB command. All users have the ability to run execs in the "authorized exec" sublibraries specified on the SETSYS AUTHELIB command. All users can run execs in sublibraries specified in the LIBDEF PROC search chain for the CICS partition. Authorized users can be defined by any existing authorized user or in an authorized exec. The REXX/CICS CICSTART exec that is called at REXX/CICS initialization (at the first REXX/CICS transaction after a CICS restart) is automatically authorized. This is the logical place to define authorized users and libraries. The sublibrary containing the CICSTART exec is treated as the initial "authorized command" and "authorized exec" sublibrary.

Because access to REXX/CICS libraries can easily be controlled, this is the logical counterpart to controlling access to CICS production program libraries. Any commands that a site feels are sensitive (such as READ, WRITE, and DELETE) could be defined as authorized in the production region. This would mean that only authorized users could create execs that issue authorized commands and decide whether all users could invoke these execs that contain authorized commands or only other authorized users.

Note: You can control the ability of REXX/CICS execs to access external APIs by redefining the CICS START, LINK, and XCTL commands as REXX/CICS authorized commands.

H.5 Security Definitions

This section discusses the security definitions for REXX/CICS such as: general users, authorized users, authorized commands, authorized exec, and system libraries.

H.5.1 REXX/CICS General Users

REXX/CICS users that are not defined as authorized by the AUTHUSER command cannot use REXX/CICS authorized commands. However, these users can define, write, alter, and use user commands (defined using the DEFCMD command) and execs. Users can also use (but not define, create, or alter) REXX/CICS authorized execs that reside in the CICEXEC library.

H.5.2 REXX/CICS Authorized Users

Authorized users are defined by the AUTHUSER command, that are allowed to use authorized REXX/CICS commands (commands defined using the DEFCMD or DEFSCMD command with the AUTH option specified).

H.5.3 REXX/CICS Authorized Commands

Authorized commands are REXX/CICS commands that can only be used by authorized users or from authorized execs. Authorized commands are defined using the DEFCMD or DEFSCMD command with the AUTH option specified.

H.5.4 REXX/CICS Authorized Execs

Authorized execs are programs (execs) that were loaded from sublibraries that were specified on the SETSYS AUTHCLIB or SETSYS AUTHELIB commands and are considered authorized. That is, these programs are allowed to use authorized REXX/CICS commands. All REXX/CICS users have access to execs loaded from the sublibraries specified on the SETSYS AUTHELIB command, but only authorized users have access to commands and execs loaded from the sublibraries specified on the SETSYS AUTHCLIB command.

H.5.5 REXX/CICS System Sublibraries

All authorized commands written in the REXX language must be loaded from a VSE Librarian sublibrary specified on the SETSYS AUTHCLIB command. These may be both IBM and customer (or vendor) supplied.

All authorized execs must be loaded from a VSE Librarian sublibrary specified on either the SETSYS AUTHCLIB or SETSYS AUTHELIB commands. These may be both IBM and customer (or vendor) supplied.

User execs that are not authorized but are being shared by all REXX/CICS users can be placed in a VSE Librarian sublibrary specified in the LIBDEF PROC search chain for the CICS partition.

Notes:

 1. The AUTH option of the DEFCMD or DEFSCMD is itself an authorized command option. That is, AUTH may only be used if the user issuing it is an authorized user or if it was issued from an exec loaded from an authorized sublibrary.

 2. The EXECLOAD and EXECDROP commands are authorized. Therefore, only an authorized user or exec can EXECLOAD an exec from an authorized sublibrary.

Security

Appendix I. Performance Considerations

Because of the production nature of CICS, emphasis is placed on performance. Many design choices can affect performance. These include:

- How REXX environments are defined
- How the REXX File System structure is implemented
- How security interfaces are implemented
- How much virtual storage is given to an exec at invocation.

REXX uses sophisticated techniques, such as look-aside tables and tree balancing, for good performance. Although REXX execs are interpreted, most of the actual processing for the typical application is spent executing the REXX commands that do most of the actual work. These commands can be (and usually are) written in Assembler, or a compiled language, when performance is an important consideration.

Client/server support is a REXX feature that provides a substantial performance advantage. With this facility, a server REXX exec can often be used instead of a nested REXX exec to provide application function. The performance characteristics of such a server can be better managed. The advantage of a server exec over a nested exec is that a server exec can be started and can process multiple client requests before ending. This has a shorter path-length, provides better response time, and often uses less system resource.

An EXEC CICS SUSPEND is automatically issued after every 1024 clauses executed in an exec to help prevent a REXX/CICS exec from monopolizing processor resources.

Usually, for the majority of small- to medium-scale CICS applications, the productivity benefits of using REXX far outweighs the performance penalty.

REXX/CICS execs may reside in either VSAM-based REXX File System files or in VSE Librarian sublibraries.

485

Performance Considerations

Appendix J. Basic Mapping Support Example

This appendix has a list of steps that you must follow so you can use the CICS basic mapping support (BMS) within the REXX/CICS environment. The steps include:

1. BMS maps must be assembled and linked into a CICS library. This library must be in the LIBDEF in the CICS region startup JCL.

2. If you are going to use the REXX/CICS CONVTMAP command to generate a file structure, the BMS map must be assembled to produce the map DSECT.

3. BMS maps must be defined to CICS using Resource Definition Online (RDO).

4. If you want to read from or send data to the screen, you need to GETMAIN CICS storage for the length of the map input/output areas. The storage must be initialized to nulls.

5. If a field within a file structure is represented by more than one label, then the last label to reference that field in the structure must be used when referencing the field.

The example in this appendix is using BMS map PANELG. The following is the map definition.

```
         TITLE 'PANEL GROUP FOR REXX/CICS        '                       00000010
         PRINT ON,NOGEN                                                  00000020
PANELG   DFHMSD TYPE=MAP,LANG=ASM,MODE=INOUT,STORAGE=AUTO,SUFFIX=        00000030
         TITLE 'TEST PANEL FOR REXX/CICS         '                       00000040
DPANEL1  DFHMDI SIZE=(24,80),CTRL=(FREEKB),MAPATTS=(COLOR,HILIGHT),     *00000050
              DSATTS=(COLOR,HILIGHT),COLUMN=1,LINE=1,DATA=FIELD,        *00000060
              TIOAPFX=YES,OBFMT=NO                                       00000070
         DFHMDF POS=(1,1),LENGTH=1,ATTRB=(PROT,BRT)                      00000080
         DFHMDF POS=(5,27),LENGTH=22,INITIAL='REXXCICS HEADER PANEL1',  *00000090
              ATTRB=(PROT,NORM)                                          00000100
         DFHMDF POS=(5,73),LENGTH=6,INITIAL='PANEL1',ATTRB=(PROT,NORM)   00000110
         DFHMDF POS=(9,6),LENGTH=25,                                    *00000120
              INITIAL='PLEASE ENTER YOUR USERID:',ATTRB=(PROT,NORM)      00000130
* DUSERID                                                                00000140
DUSERID  DFHMDF POS=(9,32),LENGTH=8,ATTRB=(UNPROT,BRT,IC,FSET)           00000150
         DFHMDF POS=(9,41),LENGTH=1,ATTRB=(PROT,NORM)                    00000160
         DFHMDF POS=(14,6),LENGTH=4,INITIAL='MSG:',ATTRB=(PROT,NORM)     00000170
* DMSG                                                                   00000180
DMSG     DFHMDF POS=(14,11),LENGTH=29,ATTRB=(UNPROT,NORM,FSET)           00000190
         DFHMDF POS=(14,41),LENGTH=1,ATTRB=(PROT,NORM)                   00000200
         DFHMSD TYPE=FINAL                                               00000210
         END                                                            00000220
```

487

The map DSECT follows.

```
* TEST PANEL FOR REXX/CICS                                       00000010
PANEL1S EQU   *                     START OF DEFINITION          00000020
        SPACE                                                    00000030
        DS    CL12                  TIOA PREFIX                  00000040
DUSERIDL DS   CL2                   INPUT DATA FIELD LENGTH      00000050
DUSERIDF DS   0C                    DATA FIELD FLAG              00000060
DUSERIDA DS   C                     DATA FIELD 3270 ATTRIBUTE    00000070
DUSERIDC DS   C                     COLOR ATTRIBUTE             00000080
DUSERIDH DS   C                     HIGHLIGHTING ATTRIBUTE       00000090
DUSERIDI DS   0CL8                  INPUT DATA FIELD             00000100
DUSERIDO DS   CL8                   OUTPUT DATA FIELD            00000110
        SPACE                                                    00000120
DMSGL   DS    CL2                   INPUT DATA FIELD LENGTH      00000130
DMSGF   DS    0C                    DATA FIELD FLAG              00000140
DMSGA   DS    C                     DATA FIELD 3270 ATTRIBUTE    00000150
DMSGC   DS    C                     COLOR ATTRIBUTE             00000160
DMSGH   DS    C                     HIGHLIGHTING ATTRIBUTE       00000170
DMSGI   DS    0CL29                 INPUT DATA FIELD             00000180
DMSGO   DS    CL29                  OUTPUT DATA FIELD            00000190
        SPACE                                                    00000200
PANEL1E EQU   *                                                 00000210
        ORG   PANEL1S               ADDRESS START               00000220
* CALCULATE MAPLENGTH, ASSIGNING A VALUE OF ONE WHERE LENGTH=ZERO 00000230
PANEL1L EQU   PANEL1E-PANEL1S                                   00000240
PANEL1I DS    0CL(PANEL1L+1-(PANEL1L/PANEL1L))                  00000250
PANEL1O DS    0CL(PANEL1L+1-(PANEL1L/PANEL1L))                  00000260
        ORG                                                      00000270
* * * END OF DEFINITION * * *                                   00000280
        SPACE 3                                                  00000290
        ORG                                                      00000300
PANELGT EQU   *                     * END OF MAP SET            00000310
* * * END OF MAP SET DEFINITION * * *                           00000320
        SPACE 3                                                  00000330
```

The CONVTMAP command is used to take the DSECT and create a file structure stored in the RFS. The command is entered as follows:

```
'CONVTMAP USER.TEST(PANELG) POOL1:\USERS\USER1\PANELG.DATA'
```

The following is the file structure created by CONVTMAP.

```
00000 **************************** TOP OF DATA *************************
00001 DUSERIDL 13 2   C
00002 DUSERIDF 15 1   C
00003 DUSERIDA 15 1   C
00004 DUSERIDC 16 1   C
00005 DUSERIDH 17 1   C
00006 DUSERIDI 18 8   C
00007 DUSERIDO 18 8   C
00008 DMSGL 26 2    C
00009 DMSGF 28 1    C
00010 DMSGA 28 1    C
00011 DMSGC 29 1    C
00012 DMSGH 30 1    C
00013 DMSGI 31 29   C
00014 DMSGO 31 29   C
00015 **************************** BOTTOM OF DATA************************
```

The following example is exec BMSMAP1. It creates a simple panel that asks for a user ID.

```
/* This EXEC uses CICS SEND and RECEIVE commands            */
/* The panel has two fields USERID and a message            */
/* field.  The panel is initially displayed with            */
/* a message - "USERID must be 8 characters"                */

/* GETMAIN storage to be used for data mapping              */
/* and initialize                                           */
'PSEUDO OFF'
ZEROES = '00'x
'CICS GETMAIN SET(WORKPTR) LENGTH(90) INITIMG(ZEROES)'

VAR1 = 'USERID must be 8 characters'

/* Copy the REXX variable VAR1 to the GETMAINed storage     */
'COPYR2S VAR1 WORKPTR 30'

/* Copy the storage area to REXX variable                   */
'COPYS2R WORKPTR X 0 90'

'CICS SEND MAP(PANELG) FREEKB ERASE FROM(X)'
'CICS RECEIVE MAP(PANELG) INTO(Y)'

/* Copy Y into the GETMAINED storage area and then copy the data */
/* to REXX variables using the file structure generated     */
/* previously by the CONVTMAP command                       */
'COPYR2S Y WORKPTR 0 90'
'COPYS2R WORKPTR * POOL1:\USERS\USER1\PANELG.DATA'

/* loop until the user enters a USERID exactly 8 characters in  */
/* length                                                   */
do forever
MUSERID = STRIP(DUSERIDO)
if LENGTH(MUSERID) < 8  then
  do
   DMSGO = 'Please enter 8 char USERID'
   'COPYR2S * WORKPTR POOL1:\USERS\USER1\PANELG.DATA'
   'COPYS2R WORKPTR X 0 90'
   'CICS SEND MAP(PANELG) FREEKB ERASE FROM(X)'
   'CICS RECEIVE MAP(PANELG) INTO(Z)'
   'COPYR2S Z WORKPTR 0 90'
   'COPYS2R WORKPTR * POOL1:\USERS\USER1\PANELG.DATA'
  END
 ELSE LEAVE
END
'SENDE'
say ' '
say 'Hello' DUSERIDO', Welcome to REXX/CICS !!'
exit
```

The BMSMAP1 exec created the following panels.

```
                    REXX/CICS HEADER PANEL1                    PANEL1

PLEASE ENTER YOUR USERID:

MSG: USERID must be 8 characters
```

```
                    REXX/CICS HEADER PANEL1                    PANEL1

PLEASE ENTER YOUR USERID: TEST

MSG: Please enter 8 character USERID
```

Appendix K. Post-Installation Configuration

This appendix has a list of the steps to configure the REXX support.

K.1 Create the RFS Filepools

The REXX Filing System uses two or more filepools to store data. These are implemented as sets of VSAM clusters. There is a skeleton sample job called CICVSAM.J in PRD1.BASE to define two filepools for REXX. You should change this job to reflect your own environment as, for example, the dataset names. When you have completed the changes, run the job to define the filepools.

K.2 Install Resource Definitions

Definitions for Profile, Program, Transaction and File resources for REXX, are in group CICREXX, which is in the supplied or upgraded CSD.

You may make some changes to the resource definitions, for example you may want to change the name of one or more of the provided transactions, and you probably need to change the dataset names specified in the four file definitions to conform to those used when creating the RFS filepools.

Create a new group by copying CICREXX, and then make the changes to the new group.

When you have completed any changes, use CEDA to INSTALL the new group and add it to the appropriate GRPLISTs ready for the next cold start.

K.3 Update LSRPOOL Definitions

Because the RFS files have a maximum key length of 252, and the supplied VSAM definitions use a control interval size of 18K, it is important to check that the LSRPOOL used for the RFS supports these values.

If you fail to do this, you will get an OPEN error on the system console with the message:

```
4228I FILE RFSxxxx OPEN ERROR X'DC' (220)
```

K.4 Rename supplied Procedures

Supply the following input to LIBR to copy and rename the supplied EXECs (PROCs). Replace user.sublib with a suitable sublibrary (for example, PRD2.CONFIG) in the LIBDEF PROC or LIBDEF * search chain for the CICS initialization JCL.

```
CONNECT S=PRD1.BASE : user.sublib
COPY CICAUSER.Z:=.PROC R=Y
COPY CICCD.Z:=.PROC R=Y
COPY CICCLD.Z:=.PROC R=Y
COPY CICDEFT.Z:=.PROC R=Y
COPY CICDIR.Z:=.PROC R=Y
COPY CICEDIT.Z:=.PROC R=Y
COPY CICEMAP.Z:=.PROC R=Y
COPY CICEPROF.Z:=.PROC R=Y
COPY CICESVR.Z:=.PROC R=Y
COPY CICEXIO.Z:=.PROC R=Y
COPY CICFLST.Z:=.PROC R=Y
COPY CICFPOOL.Z:=.PROC R=Y
COPY CICFPROF.Z:=.PROC R=Y
COPY CICFSVR.Z:=.PROC R=Y
COPY CICGVER.Z:=.PROC R=Y
COPY CICGVERD.Z:=.PROC R=Y
COPY CICHELP.Z:=.PROC R=Y
COPY CICHPREP.Z:=.PROC R=Y
COPY CICIVP1.Z:=.PROC R=Y
COPY CICIVP2.Z:=.PROC R=Y
COPY CICIVP3.Z:=.PROC R=Y
COPY CICLCLIB.Z:=.PROC R=Y
COPY CICLELIB.Z:=.PROC R=Y
COPY CICLISTC.Z:=.PROC R=Y
COPY CICLISTP.Z:=.PROC R=Y
COPY CICLISTT.Z:=.PROC R=Y
COPY CICOVSIB.Z:=.PROC R=Y
COPY CICPATH.Z:=.PROC R=Y
COPY CICPSAMP.Z:=.PROC R=Y
COPY CICRXTRY.Z:=.PROC R=Y
COPY CICSET.Z:=.PROC R=Y
COPY CICSETS.Z:=.PROC R=Y
COPY CICSPROF.Z:=.PROC R=Y
COPY CICSTART.Z:=.PROC R=Y
COPY CICTRMID.Z:=.PROC R=Y
COPY CICXPROF.Z:=.PROC R=Y
```

K.5 Update CICSTART.PROC

Edit CICSTART.PROC from your chosen sublibrary (for example PRD2.CONFIG) using DITTO or another editor to update the list of authorized user ids. If you fail to do this, the filepool formatting function gives the message:

Subcommand return code = -4

Either change the existing 'AUTHUSER RCUSER' line or add additional lines specifying the userid that will log on to format the pools, for example 'AUTHUSER CICSUSER' if no security is active, or 'AUTHUSER SYSA' if userid SYSA is used. Use single apostrophes to delimit the command.

Two filepools are defined in the CICSTART.PROC. Change them to match changes to the supplied CICREXX group. You can add additional filepools at a later date:

```
'FILEPOOL DEFINE POOL1 RFSDIR1 RFSPOL1 (USER'
  IF RC ¬= 0 THEN EXIT RC
'FILEPOOL DEFINE POOL2 RFSDIR2 RFSPOL2 (USER'
  IF RC ¬= 0 THEN EXIT RC
```

K.6 Update CICS Initialization JCL

REXX must be able to access sublibrary-resident EXECs through the search chain of either a LIBDEF PROC or a LIBDEF * statement. Either add a LIBDEF statement, or modify an existing one to include the sublibrary that contains the PROCs copied in the earlier step. You may now add the sublibraries that contain the user EXECs, for example,

// LIBDEF PROC,SEARCH=(PRD2.CONFIG)

If you fail to make your chosen sublibraries available to resolve requests for executing the sublibrary-resident execs, any REXX transaction will give the message:

CICREX490E Error 48 running CICSTART EXEC: Failure in system service

If the DSNAME and CATNAME parameters are removed from the supplied CSD file definitions for the RFS, appropriate DLBL statements are required in the initialization JCL.

K.7 Format the RFS Filepools

Ensure that all required configuration tasks have been performed, and if necessary re-start CICS. Sign on with a userid defined as an authorized user in CICSTART.PROC. Enter REXX (which is the default transaction id associated with the CICRXTRY exec). You should see the following line at the top of the screen:

```
Enter a REXX command or EXIT to quit
```

and a **READ** in the lower right hand corner. The cursor is in the lower left hand corner. You have now entered the supplied exec which allows the execution of REXX and REXX/CICS commands interactively.

Now prepare the filepools for use by entering a command for every filepool that is defined in CICSTART:

'FILEPOOL FORMAT poolname

where **poolname** is substituted by the filepool name you specified in the CICSTART exec (the defaults are POOL1 and POOL2).

Note: We recommend you to use either single or double apostrophes to delimit the REXX command.

The interactive environment echoes each command at the next available line on the screen and displays any requested output.

The FILEPOOL FORMAT command does not display any information to indicate that it has successfully completed, but the word **READ** appears in the lower right hand corner of the screen.

To determine whether the FILEPOOL FORMAT command worked successfully, enter SAY RC (without apostrophes). If a **0** is displayed on the next available line, the FILEPOOL FORMAT command was successful.

Attempting to re-format a filepool gives the message:

Subcommand return code = 1836

This is also the return code displayed by SAY RC.

Continue this process until all RFS filepools are formatted. You only format the filepool when a new filepool is defined, or if you delete and redefine the clusters for an existing filepool.

If you fill the screen while formatting the filepools or interactively executing REXX or REXX/CICS commands and instructions, a **MORE** indicator appears at the bottom right hand corner. To clear the screen, press the ENTER key. You may press the CLEAR key any time you want to clear the screen of data. Press the PF3 key to exit from the interactive environment, this simulates entering the EXIT instruction. You may also enter the EXIT instruction yourself (without apostrophes).

The interactive environment also allows recalling previously entered commands, by pressing the RETRIEVE key. The system has a default setting for this key of PF12. Pressing the RETRIEVE key causes the previously entered line to be re-displayed at the input location. You may then modify this area and execute the instruction again by pressing ENTER. Pressing the RETRIEVE key multiple times continues to bring the next previously entered command to the input area.

K.8 Create the Help Files

To create the help files, rename some modules using the following as sample LIBR input. Choose an appropriate sublibrary (for example PRD2.CONFIG) to contain the renamed members:

```
CONNECT S=PRD1.BASE : user.sublib
COPY CICR3270.Y : CICR3270.BOOK
COPY CICINDEX.N : CICINDEX.PANSRC
COPY CICSNDX.N : CICSNDX.PANSRC
COPY CICCHAP.N : CICCHAP.PANSRC
```

Start CICS with at least EDSALIM=25M. Approximately 10M of free EDSA is needed to execute this procedure, therefore allow this much extra over your normal configuration. CEMT I DSAS displays the current usage, and can increase EDSALIM as long as partition getvis storage is available. The following example shows EDSALIM=30M, and the four extended DSAs currently using 7MB, therefore there is 23MB free EDSA available.

```
 I DSAS
 STATUS:  RESULTS - OVERTYPE TO MODIFY
   Sosstatus(Notsos)

 Dsalimit( 06291456 )
  Cdsasize(00524288)
  Rdsasize(00524288)
  Sdsasize(00262144)
  Udsasize(00000000)

 Edsalimit( 0031457280 )
  Ecdsasize(0002097152)
  Erdsasize(0007340032)
  Esdsasize(0001048576)
  Eudsasize(0001048576)
```

Start the REXX transaction, then enter EXEC CICHPREP (without apostrophes) to run the exec that creates the HELP text.

In response to the following messages, enter user.sublib where this is the sublibrary you chose (for example, PRD2.CONFIG):

```
Please enter the Librarian sublibrary containing the REXX/CICS LIST3270.
(example: LIB.SUBLIB )
user.sublib
Please enter the Librarian sublibrary containing the help panels.
(example: LIB.SUBLIB )
user.sublib
```

You should see the following messages:

```
  Building INDEX and CHAPTER files.  Please be patient.
  INDEX file built.
  Building Chapter files
  Chapter files are built
  Building Appendix files.
  Appendix files are built
  34 Files built, all done !!
```

K.9 Verify the Installation

To verify that the installation has been successful, an exec has been supplied.

Execute the REXX transactionX environment used to format the filepools, enter CALL CICIVP1. The exec indicates what should occur. Remember to press ENTER when **MORE** appears on the lower right hand corner of the screen.

Sample output:

```
Enter a REXX command or EXIT to quit
CALL CICIVP1
***-----------------------------------------------------------***
*** This is a test REXX program running under CICS/VSE   ***
*** It was loaded from PROCLIB-user.sublib(CICIVP1)
***-----------------------------------------------------------***

What is your name?

<type name and press ENTER>

Welcome to REXX/CICS for CICS/VSE  , xxxxx

Invoking nested exec CICIVP2 (which has tracing on)
    11 *-* say 'You entered CICIVP2 exec'
       >>>   "You entered CICIVP2 exec"
You entered CICIVP2 exec
    12 *-* call CICIVP3
You entered CICIVP3 exec which has tracing off
    13 *-* exit
Back to CICIVP1 exec

          This is fullscreen output to terminal xxxx
          Now input some data and press ENTER or a PF key

<type data and press ENTER>

The AID key that was pressed = ENTER
The cursor was at (Row Col): 24 5
The data that was entered (Row Col Data): 24 1 xxxxx

<press ENTER>

Example of more than one screen
1000 assignment statements have been executed
2000 assignment statements have been executed
3000 assignment statements have been executed
4000 assignment statements have been executed
5000 assignment statements have been executed
6000 assignment statements have been executed
7000 assignment statements have been executed
8000 assignment statements have been executed
9000 assignment statements have been executed
10000 assignment statements have been executed
11000 assignment statements have been executed
12000 assignment statements have been executed
13000 assignment statements have been executed
14000 assignment statements have been executed
15000 assignment statements have been executed
16000 assignment statements have been executed
17000 assignment statements have been executed
18000 assignment statements have been executed
19000 assignment statements have been executed
20000 assignment statements have been executed
```

```
<press ENTER to clear the "MORE">

 Today's date is dd mmm yyyy
 The time is hh.mm.ss
 REXX/CICS CICIVP1 is now finished
```

K.10 Configure the REXX DB2 Interface

This step is only required if the REXX EXECSQL command environment is enabled for DB2 support. DB2 must be fully installed before this step can be performed.

A package for the CICSQL program is loaded into the DB2 database under the SQLDBA user. This is supplied as member CICSQL.A.

If you fail to do this, REXX gives return codes such as -805 whenever an "ADDRESS EXECSQL command" is issued.

Adapt the following JCL and run DB2 in single user mode, substituting the correct **password** for user SQLDBA. The default is SQLDBAPW but this is normally changed as part of the DB2 installation process.

```
// JOB RELOAD CICSQL PACKAGE
// LIBDEF *,SEARCH=(PRD2.DB2710,PRD1.BASE)
// EXEC PROC=ARIS71DB       *-- STARTER DATABASE IDENTIFICATION
// EXEC PROC=ARISDBSD       *-- RUN DB2 IN SINGLE-USER MODE
CONNECT SQLDBA IDENTIFIED BY password;
RELOAD PROGRAM (SQLDBA.CICSQL) REPLACE KEEP INFILE(SYSIPT BLKSZ(80)
PDEV(DASD));
READ MEMBER CICSQL
/*
COMMIT WORK;
/*
/&
```

Issue GRANT EXECUTE ON SQLDBA.CICSQL, as required, to enable users to run the package.

Bibliography

CICS Transaction Server for VSE/ESA Release 1 library

Evaluation and planning	
Enhancements Guide	GC34-5763
Release Guide	GC33-1645
Migration Guide	GC33-1646
Report Controller Planning Guide	SC33-1941

General	
Master Index	SC33-1648
Trace Entries	SX33-6108
User's Handbook	SX33-6101
Glossary (softcopy only)	GC33-1649

Administration	
System Definition Guide	SC33-1651
Customization Guide	SC33-1652
Resource Definition Guide	SC33-1653
Operations and Utilities Guide	SC33-1654
CICS-Supplied Transactions	SC33-1655

Programming	
Application Programming Guide	SC33-1657
Application Programming Reference	SC33-1658
Sample Applications Guide	SC33-1713
Application Migration Aid Guide	SC33-1943
System Programming Reference	SC33-1659
Distributed Transaction Programming Guide	SC33-1661
Front End Programming Interface User's Guide	SC33-1662
REXX Guide	SC34-5764

Diagnosis	
Problem Determination Guide	GC33-1663
Messages and Codes Vol 3 (softcopy only)	SC33-6799
Diagnosis Reference	LY33-6085
Data Areas	LY33-6086
Supplementary Data Areas	LY33-6087

Communication	
Intercommunication Guide	SC33-1665
Internet Guide	SC34-5765
CICS Family: Interproduct Communication	SC33-0824
CICS Family: Communicating from CICS on System/390	SC33-1697

Special topics	
Recovery and Restart Guide	SC33-1666
Performance Guide	SC33-1667
Shared Data Tables Guide	SC33-1668
Security Guide	SC33-1942
External Interfaces Guide	SC33-1669
XRF Guide	SC33-1671
Report Controller User's Guide	SC34-5688

CICS Clients	
CICS Clients: Administration	SC33-1792
CICS Universal Clients Version 3 for OS/2: Administration	SC34-5450
CICS Universal Clients Version 3 for Windows: Administration	SC34-5449
CICS Universal Clients Version 3 for AIX: Administration	SC34-5348
CICS Universal Clients Version 3 for Solaris: Administration	SC34-5451
CICS Family: OO programming in C++ for CICS Clients	SC33-1923
CICS Family: OO programming in BASIC for CICS Clients	SC33-1671
CICS Family: Client/Server Programming	SC33-1435
CICS Transaction Gateway Version 3: Administration	SC34-5448

Where to Find More Information

You can find more information about CICS TS for VSE/ESA and REXX in the following publications:

- The *CICS/TS for VSE/ESA Application Programming Reference*, SC33-1688-03, is helpful to programmers using CICS TS for VSE/ESA as it contains information about the application programming commands.

- The *CICS/TS for VSE/ESA System Programming Reference*, SC33-1689-03, is helpful to programmers using CICS TS for VSE/ESA as it contains information about the system programming commands.

- *The REXX Language, A Practical Approach to Programming*, by M. F. Cowlishaw (IBM* Order number: ZB35-5100 and available in book stores) offers general information about the REXX language.

- The *SAA* CPI Common Programming Interface REXX Level 2 Reference*, SC24-5549, may be useful to more experienced REXX users who may wish to code portable programs.

- *CICS/TS for VSE/ESA CICS-Supplied Transactions*, SC33-1686-03, contains information about CEMT and CEDA.

- *MVS/ESA SPV4 Assembler Programming Reference*, GC28-1642-05

Books from VSE/ESA 2.5 base program libraries

VSE/ESA Version 2 Release 5

Book title	Order number
Administration	SC33-6705
Diagnosis Tools	SC33-6614
Extended Addressability	SC33-6621
Guide for Solving Problems	SC33-6710
Guide to System Functions	SC33-6711
Installation	SC33-6704
Licensed Program Specification	GC33-6700
Messages and Codes Volume 1	SC33-6796
Messages and Codes Volume 2	SC33-6798
Messages and Codes Volume 3	SC33-6799
Networking Support	SC33-6708
Operation	SC33-6706
Planning	SC33-6703
Programming and Workstation Guide	SC33-6709
System Control Statements	SC33-6713
System Macro Reference	SC33-6716
System Macro User's Guide	SC33-6715
System Upgrade and Service	SC33-6702
System Utilities	SC33-6717
TCP/IP User's Guide	SC33-6601
Turbo Dispatcher Guide and Reference	SC33-6797
Unattended Node Support	SC33-6712

High-Level Assembler Language (HLASM)

Book title	Order number
General Information	GC26-8261
Installation and Customization Guide	SC26-8263
Language Reference	SC26-8265
Programmer's Guide	SC26-8264

Language Environment for VSE/ESA (LE/VSE)

Book title	Order number
C Run-Time Library Reference	SC33-6689
C Run-Time Programming Guide	SC33-6688
Concepts Guide	GC33-6680
Debug Tool for VSE/ESA Fact Sheet	GC26-8925
Debug Tool for VSE/ESA Installation and Customization Guide	SC26-8798
Debug Tool for VSE/ESA User's Guide and Reference	SC26-8797
Debugging Guide and Run-Time Messages	SC33-6681
Diagnosis Guide	SC26-8060
Fact Sheet	GC33-6679
Installation and Customization Guide	SC33-6682
LE/VSE Enhancements	SC33-6778
Licensed Program Specification	GC33-6683
Programming Guide	SC33-6684
Programming Reference	SC33-6685
Run-Time Migration Guide	SC33-6687
Writing Interlanguage Communication Applications	SC33-6686

VSE/ICCF

Book title	Order number
Adminstration and Operations	SC33-6738
User's Guide	SC33-6739

VSE/POWER

Book title	Order number
Administration and Operation	SC33-6733
Application Programming	SC33-6736
Networking Guide	SC33-6735
Remote Job Entry User's Guide	SC33-6734

VSE/VSAM

Book title	Order number
Commands	SC33-6731
User's Guide and Application Programming	SC33-6732

VTAM for VSE/ESA

Book title	Order number
Customization	LY43-0063
Diagnosis	LY43-0065
Data Areas	LY43-0104
Messages and Codes	SC31-6493
Migration Guide	GC31-8072
Network Implementation Guide	SC31-6494
Operation	SC31-6495
Overview	GC31-8114
Programming	SC31-6496
Programming for LU6.2	SC31-6497
Release Guide	GC31-8090
Resource Definition Reference	SC31-6498

Books from VSE/ESA 2.5 optional program libraries

C for VSE/ESA (C/VSE)

Book title	Order number
C Run-Time Library Reference	SC33-6689
C Run-Time Programming Guide	SC33-6688
Diagnosis Guide	GC09-2426
Installation and Customization Guide	GC09-2422
Language Reference	SC09-2425
Licensed Program Specification	GC09-2421
Migration Guide	SC09-2423
User's Guide	SC09-2424

COBOL for VSE/ESA (COBOL/VSE)

Book title	Order number
Debug Tool for VSE/ESA Fact Sheet	GC26-8925
Debug Tool for VSE/ESA Installation and Customization Guide	SC26-8798
Debug Tool for VSE/ESA User's Guide and Reference	SC26-8797
Diagnosis Guide	SC26-8528
General Information	GC26-8068
Installation and Customization Guide	SC26-8071
Language Reference	SC26-8073
Licensed Program Specifications	GC26-8069
Migration Guide	GC26-8070
Migrating VSE Applications To Advanced COBOL	GC26-8349
Programming Guide	SC26-8072

DB2 Server for VSE

Book title	Order number
Application Programming	SC09-2393
Database Administration	GC09-2389
Installation	GC09-2391
Interactive SQL Guide and Reference	SC09-2410
Operation	SC09-2401
Overview	GC08-2386
System Administration	GC09-2406

DL/I VSE

Book title	Order number
Application and Database Design	SH24-5022
Application Programming: CALL and RQDLI Interface	SH12-5411
Application Programming: High-Level Programming Interface	SH24-5009
Database Administration	SH24-5011
Diagnostic Guide	SH24-5002
General Information	GH20-1246
Guide for New Users	SH24-5001
Interactive Resource Definition and Utilities	SH24-5029
Library Guide and Master Index	GH24-5008
Licensed Program Specifications	GH24-5031
Low-level Code and Continuity Check Feature	SH20-9046
Library Guide and Master Index	GH24-5008
Messages and Codes	SH12-5414
Recovery and Restart Guide	SH24-5030
Reference Summary: CALL Program Interface	SX24-5103
Reference Summary: System Programming	SX24-5104
Reference Summary: HLPI Interface	SX24-5120
Release Guide	SC33-6211

PL/I for VSE/ESA (PL/I VSE)

Book title	Order number
Compile Time Messages and Codes	SC26-8059
Debug Tool For VSE/ESA User's Guide and Reference	SC26-8797
Diagnosis Guide	SC26-8058
Installation and Customization Guide	SC26-8057
Language Reference	SC26-8054
Licensed Program Specifications	GC26-8055
Migration Guide	SC26-8056
Programming Guide	SC26-8053
Reference Summary	SX26-3836

Screen Definition Facility II (SDF II)

Book title	Order number
VSE Administrator's Guide	SH12-6311
VSE General Introduction	SH12-6315
VSE Primer for CICS/BMS Programs	SH12-6313
VSE Run-Time Services	SH12-6312

Notices

This information was developed for products and services offered in the U.S.A. IBM may not offer the products, services, or features discussed in this document in other countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, NY 10504-1785
U.S.A.

For license inquiries regarding double-byte (DBCS) information, contact the IBM Intellectual Property Department in your country or send inquiries, in writing, to:

IBM World Trade Asia Corporation
Licensing
2-31 Roppongi 3-chome, Minato-ku
Tokyo 106, Japan

The following paragraph does not apply in the United Kingdom or any other country where such provisions are inconsistent with local law:

INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY, OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact IBM United Kingdom Laboratories, MP151, Hursley Park, Winchester, Hampshire, England, SO21 2JN. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Programming License Agreement, or any equivalent agreement between us.

Trademarks

The following terms are trademarks of International Business Machines Corporation in the United States, or other countries, or both:

ACF/VTAM	AFP
AIX	AT
C/370	CICS
CICS OS/2	CICS/ESA
CICS/MVS	CICS/VSE
CICSPLex	DATABASE 2
DB2	Hiperbatch
IBM	IBMLink
IMS/ESA	MVS/ESA
OfficeVision/VM	Operating System/2
OS/2	RACF
System/390	VSE/ESA

Index

Special Characters

! prefix on TRACE option   196
? prefix on TRACE option   196
/ (division operator)   26, 138, 257
// (remainder operator)   26, 138, 260
/= (not equal operator)   139
/== (strictly not equal operator)   138, 139
.DEFINE verb   344, 345
.PANEL verb   347, 348
* (multiplication operator)   26, 138, 257
- tracing flag   197
** (power operator)   26, 138, 259
\ (NOT operator)   29, 31, 139
\< (not less than operator)   29, 139
\<< (strictly not less than operator)   139
\= (not equal operator)   29, 139
\== (strictly not equal operator)   29, 138
\> (not greater than operator)   29, 139
\>> (strictly not greater than operator)   139
& (AND logical operator)   31, 139
&& (exclusive OR operator)   31, 140
% (integer division operator)   26, 138, 260
+ (addition operator)   26, 138, 257
+++ tracing flag   197
< (less than operator)   29, 139
<< (strictly less than operator)   138, 139
<<= (strictly less than or equal operator)   139
<= (less than or equal operator)   29, 139
<> (less than or greater than operator)   139
= (equal sign)   29
== (strictly equal operator)   29, 138, 139, 257
> (greater than operator)   29, 139
>.> tracing flag   197
>< (greater than or less than operator)   29, 139
>= (greater than or equal operator)   29, 139
>> (strictly greater than operator)   138, 139
>>= (strictly greater than or equal operator)   139
>>> tracing flag   36, 197
>C> tracing flag   198
>F> tracing flag   198
>L> tracing flag   35, 198
>O> tracing flag   35, 198
>P> tracing flag   198
>V> tracing flag   35, 198
| (inclusive OR operator)   31, 140
|| (concatenation operator)   33, 137

A

abuttal   33, 137

action taken when a condition is not trapped   266
action taken when a condition is trapped   266
active loops   173
additional operator examples   260
ADDRESS instruction   97, 331
address setting   157, 162
advanced topics in parsing   249
algebraic precedence   140
alphabetic character word options in TRACE   195
alphanumeric checking with DATATYPE   213
AND, logical operator   139
ANDing character strings together   208
ARBCHAR command   274
ARG instruction   19, 76, 86
ARG option of PARSE instruction   180
ARGS command   275
arguments   21
associative storage   144
assumption, XEDIT   14
AUTH command   302, 478
AUTHUSER command   370, 477, 478, 481, 482
automatic server initiation (ASI)   337

B

backslash, use of   135, 139
BACKWARD command   275
Base option of DATE function   214
basic mapping support (BMS)   152, 377
basic operator examples   259
BASSM assembler instruction   326
bits checked using DATATYPE   213
blank line   13
block prefix commands   273
BMS example   487
BMSMAP1 exec   488
BOTTOM command   276
bottom of program reached during execution   169
BSM assembler instruction   326
business solutions   473
BY phrase of DO instruction   163

C

C2S command   326, 337, 382
CALL instruction   56, 69
CANCEL command   276, 291
CASE command   277, 284
CATMOUSE EXEC   105
CD command   150, 271, 300, 371, 391
CEDA command   125, 373

CEMT command   125, 374
Century option of DATE function   214
CHANGE command   277
checking arguments with ARG function   207
CICEPROF macro   282
CICGETV routine   328
CICPARMS control block   327
CICREX program   478
CICS   97
CICS commands   95
CICSECX2 security exit   479
CICSLINK option on DEFCMD command   327
CICSLOAD option on DEFCMD command   326
CICSPROF exec   126, 482
CICSTART exec   126, 477, 478
CICXPROF macro   282
CKDIR command   303, 319
CKFILE command   303
CLD command   318, 375
client exec example   338
client/server   485
CMDLINE command   278
code page   130
coding style   111
collating sequence using XRANGE   235
collections of variables   232
command execution security   302
command level interpreter transaction (CECI)   481
CON EXEC (exercise)   104
conceptual overview of parsing   250
Configure the REXX DB2 Interface   497
 Configuring REXX support   491
constant   24
constant symbols   143
content addressable storage   144
control variable   46, 164
controlled loops   164
CONVTMAP command   377
COPY command   304
copying a string using COPIES   211
COPYR2S command   378
COPYS2R command   380
correcting your program   110
Create the Help Files   494
Create the RFS Filepools   491
CTLCHAR command   278
CURLINE command   279
current directory   300, 318
current terminal line width   222

D

DATABASE 2 (DB2) Interface   331
date and version of the language processor   182
DBCJUSTIFY function   461

DEFCMD command   150, 326, 337, 383, 481, 482
DEFSCMD command   326, 386, 482
DEFTRNID command   389
DELETE command   304, 319
derived names of variables   144
destination prefix commands   273
DIGITS option of NUMERIC instruction   176, 256
DIR command   13, 391
directory ID   299, 317
DISKR command   304
DISKW command   299, 305
DISPLAY command   280
DO FOREVER loop   47
DO Loop   54
DO...END instruction   45
DOWN command   280
DPATH command   301

E

EDIT command   271, 281, 392
EDITSVR   97
engineering notation   262
equality, testing of   138
ERROR condition of SIGNAL and CALL
  instructions   269
ETMODE   178
European option of DATE function   214
evaluation of expressions   137
exception conditions saved during subroutine calls   162
exclusive OR operator   31, 140
exclusive-ORing character strings together   209
EXEC command   282, 393
exec identifier   13
EXECDB2 command environment   331
EXECDROP command   394
EXECIO command   396
EXECLOAD command   126, 398
EXECMAP command   400
EXECSQL command environment   331
EXECUTE command   316
EXIT instruction   56, 69
EXPORT command   401
EXPOSE option of PROCEDURE instruction   183
exposed variable   183
expressions   24
EXTERNAL option of PARSE instruction   180

F

FAILURE condition of SIGNAL and CALL
  instructions   265, 269
failure, definition   147
FIFO (first-in/first-out) stacking   188
file access security   301

N

nested exec   485
nesting of control structures   162
nibbles   132
NOETMODE   178
NOEXMODE   178
Normal option of DATE function   215
not equal operator   139
not greater than operator   139
not less than operator   139
NOT operator   135, 140
NOTYPING flag cleared before error messages   429
NULLS command   289
NUMBERS command   289

O

OfficeVision/VM   485
Ordered option of DATE function   215
ORing character strings together   208
overflow, arithmetic   264
overlaying a string onto another   223

P

¬ (NOT operator)   29, 140
¬< (not less than operator)   139
¬<< (strictly not less than operator)   139
¬= (not equal operator)   139
¬== (strictly not equal operator)   138, 139
¬> (not greater than operator)   139
¬>> (strictly not greater than operator)   139
packing a string with X2C   236
pad character, definition   205
page, code   130
PANEL command   350, 351
parentheses   95
PARSE ARG instruction   87
PARSE instruction   150
PARSE PULL instruction   86
PARSE UPPER ARG instruction   87
PARSE UPPER PULL instruction   86
PARSE UPPER VALUE instruction   87
PARSE UPPER VAR instruction   87
PARSE VALUE...WITH instruction   87
PARSE VAR instruction   87
PATH command   150, 301, 372, 413, 478
permanent command destination change   156
PFKEY command   290
PFKLINE command   290
placeholder in parsing   20, 88
Post-Installation Configuration   491
powers of ten in numbers   134
precedence of operators   140

precision of arithmetic   256
predefined variables   331, 333
preface   ix
presumed command destinations   156
PROCEDURE instruction   73, 75
program identifier   7, 129
program load table (PLT)   478
protecting variables   183
prototyping development methodology   473
PSEUDO command   415
pseudo random number function of RANDOM   224
PULL instruction   15, 18, 86, 150
PULL option of PARSE instruction   181

Q

QQUIT command   291, 293
QUERY command   291
querying TRACE setting   230
QUIT command   292

R

random number function of RANDOM   224
RDIR command   307
READ command   322
recursive call   161
reference   123
relative numeric pattern in parsing   90
relative positional patterns   243
RENAME command   307
Rename supplied Procedures   491
reordering data with TRANSLATE function   231
repeating a string with COPIES   211
repetitive loops   45
reservation of keywords   467
RESERVED command   293, 308
RESET command   293
resetting command environment   369
resource definition online (RDO)   487
resource definition online transaction (CEDA)   5
restoring variables   168
RETURN instruction   56, 69
REXX program identifier   7, 13, 129
REXX/CICS Command Definition Facility   325—329
REXX/CICS commands   369
REXX/CICS File System (RFS)   150, 299
REXX/CICS List System (RLS)   317
REXX/CICS Panel Facility   341
REXX/CICS Text Editor   150, 271
REXXCICS   97
RFS   97
RFS command   299, 416, 478
RIGHT command   294
RLS   97

Sending your comments to IBM

CICS® Transaction Server for VSE/ESA™

REXX Guide

SC34-5764-00

If you want to send to IBM any comments you have about this book, please use one of the methods listed below. Feel free to comment on anything you regard as a specific error or omission in the subject matter, and on the clarity, organization or completeness of the book itself.

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.

You can send your comments to IBM in any of the following ways:

- By mail:

    IBM UK Laboratories
    Information Development
    Mail Point 095
    Hursley Park
    Winchester, SO21 2JN
    England

- By fax:

    – From outside the U.K., after your international access code use 44 1962 870229
    – From within the U.K., use 01962 870229

- Electronically, use the appropriate network ID:

    – IBM Mail Exchange: GBIBM2Q9 at IBMMAIL
    – IBMLink: HURSLEY(IDRCF)
    – Email: idrcf@hursley.ibm.com

Whichever method you use, ensure that you include:

- The publication number and title
- The page number or topic to which your comment applies
- Your name and address/telephone number/fax number/network ID.

IBM ®

Program Number: 5648-054

SC34-5764-00

IBM

CICS TS for VSE/ESA

REXX Guide