z/VM

IBM

REXX/VM User's Guide

version 6 release 1

SC24-6222-00

z/VM

REXX/VM User's Guide

version 6 release 1

SC24-6222-00

Note:

Before using this information and the product it supports, read the information in "Notices" on page 197.

This edition applies to version 6, release 1, modification 0 of IBM z/VM (product number 5741-A07) and to all subsequent releases and modifications until otherwise indicated in new editions.

This edition replaces SC24-6114-01.

Contents

Figures

Tables

About This Document

If you would like to be able to write programs, this document is for you. You will need a terminal with access to IBM® z/VM®, and you should be reasonably familiar with z/VM, but you need not have had any previous programming experience.

The programming language described by this document is called the REstructured eXtended eXecutor language (sometimes abbreviated REXX). The document also describes how the z/VM REXX/VM language processor (shortened, hereafter, to the language processor) processes or *interprets* the REstructured eXtended eXecutor language.

You will learn about:

- Contents of a REXX program, rules of syntax and substitution, and the use of variables
- How to write expressions, use conversations, enter CMS and CP commands, control your program, and construct and design your REXX programs
- Examples of REXX programs, and tailoring XEDIT through REXX programs.

Intended Audience

You should read this document if:

- You want to learn how to write programs but do not have any previous programming experience
- You are familiar with other programming languages but want to learn how to use REXX
- You have had some experience with the REXX language but want to gain more knowledge of practical examples.

As you can see, this document is not intended for any particular user possessing any particular title. REXX is a very powerful, yet adaptable language suited to fit many varying programming needs.

Before reading this document, it is important for you to consider the following items:

- If you are not familiar with CMS or SFS, read *z/VM: CMS Primer* first.
- You will need a VM user ID and logon password.
- If you are using REXX in the GCS environment, see the *z/VM: REXX/VM Reference*.

How to Read Syntax Diagrams

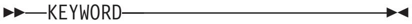
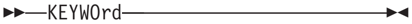
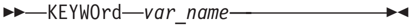
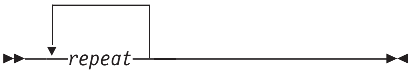
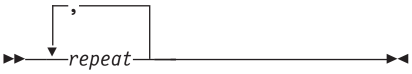
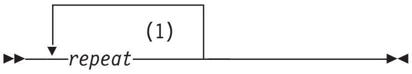
This document uses diagrams (often called *railroad tracks*) to show the syntax of external interfaces.

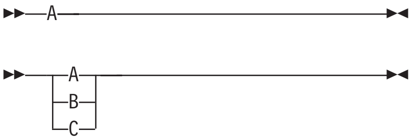
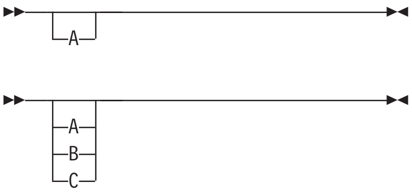
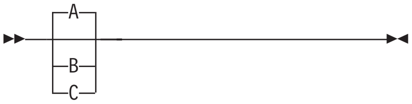
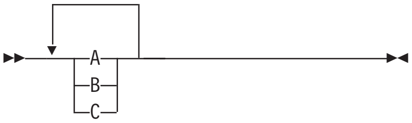
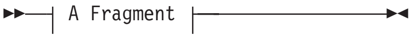
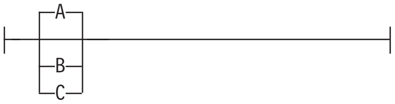
To read a syntax diagram, follow the path of the line. Read from left to right and top to bottom.

- The ►►── symbol indicates the beginning of the syntax diagram.
- The ──► symbol, at the end of a line, indicates that the syntax diagram is continued on the next line.
- The ►── symbol, at the beginning of a line, indicates that the syntax diagram is continued from the previous line.

- The ───►◄ symbol indicates the end of the syntax diagram.

Within the syntax diagram, items on the line are required, items below the line are optional, and items above the line are defaults. See the following examples.

Syntax Diagram Convention	Example
Keywords and Constants: A keyword or constant appears in uppercase letters. In this example, you must specify the item KEYWORD as shown. In most cases, you can specify a keyword or constant in uppercase letters, lowercase letters, or any combination. However, some applications may have additional conventions for using all-uppercase or all-lowercase.	
Abbreviations: Uppercase letters denote the shortest acceptable abbreviation of an item, and lowercase letters denote the part that can be omitted. If an item appears entirely in uppercase letters, it cannot be abbreviated. In this example, you can specify KEYWO, KEYWOR, or KEYWORD.	
Symbols: You must specify these symbols exactly as they appear in the syntax diagram.	* Asterisk : Colon , Comma = Equal Sign - Hyphen () Parentheses . Period
Variables: A variable appears in highlighted lowercase, usually italics. In this example, *var_name* represents a variable that you must specify following KEYWORD.	
Repetitions: An arrow returning to the left means that the item can be repeated. A character within the arrow means that you must separate each repetition of the item with that character. A number (1) by the arrow references a syntax note at the bottom of the diagram. The syntax note tells you how many times the item can be repeated. Syntax notes may also be used to explain other special aspects of the syntax.	   **Notes:** 1 Specify *repeat* up to 5 times.

Syntax Diagram Convention	Example
Required Item or Choice: When an item is on the line, it is required. In this example, you must specify A. When two or more items are in a stack and one of them is on the line, you must specify one item. In this example, you must choose A, B, or C.	
Optional Item or Choice: When an item is below the line, it is optional. In this example, you can choose A or nothing at all. When two or more items are in a stack below the line, all of them are optional. In this example, you can choose A, B, C, or nothing at all.	
Defaults: When an item is above the line, it is the default. The system will use the default unless you override it. You can override the default by specifying an option from the stack below the line. In this example, A is the default. You can override A by choosing B or C.	
Repeatable Choice: A stack of items followed by an arrow returning to the left means that you can select more than one item or, in some cases, repeat a single item. In this example, you can choose any combination of A, B, or C.	
Syntax Fragment: Some diagrams, because of their length, must fragment the syntax. The fragment name appears between vertical bars in the diagram. The expanded fragment appears in the diagram after a heading with the same fragment name. In this example, the fragment is named "A Fragment."	 **A Fragment:** 

Message and Response Notation

This document might include examples of messages or responses. Although most examples are shown exactly as they would appear, some content may depend on the specific situation. The following notation is used to show variable, optional, or alternative content:

xxx Highlighted text (usually italics) indicates a variable that represents the data that will be displayed.

[ ] Brackets enclose an optional item that might be displayed.

{ } Braces enclose alternative items, one of which will be displayed.

| The vertical bar separates items within brackets or braces.

... The ellipsis indicates that the preceding item might be repeated. A vertical ellipsis indicates that the preceding line, or a variation of that line, might be repeated.

Where to Find More Information

You can find more information about VM and REXX in the publications listed in the back of this book. See "Bibliography" on page 203.

Links to Other Online Documents

If you are viewing the Adobe® Portable Document Format (PDF) version of this document, it might contain links to other documents. A link to another document is based on the name of the requested PDF file. The name of the PDF file for an IBM document is unique and identifies the edition. The links provided in this document are for the editions (PDF names) that were current when the PDF file for this document was generated. However, newer editions of some documents (with different PDF names) might exist. A link from this document to another document works only when both documents reside in the same directory.

How to Send Your Comments to IBM

We appreciate your input on this publication. Feel free to comment on the clarity, accuracy, and completeness of the information or give us any other feedback that you might have.

Use one of the following methods to send us your comments:

1. Send an e-mail to mhvrcfs@us.ibm.com
2. Visit the z/VM reader's comments Web page at www.ibm.com/systems/z/os/zvm/zvmforms/webqs.html
3. Mail the comments to the following address:

   IBM Corporation
   Attention: MHVRCFS Reader Comments
   Department H6MA, Mail Station P181
   2455 South Road
   Poughkeepsie, NY 12601-5400
   U.S.A.
4. Fax the comments to us as follows:

   From the United States and Canada: 1+845+432-9405
   From all other countries: Your international access code +1+845+432-9405

Include the following information:

- Your name and address
- Your e-mail address
- Your telephone or fax number
- The publication title and order number:

  z/VM V6R1 REXX/VM User's Guide
  SC24-6222-00
- The topic and page number related to your comment
- The text of your comment

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.

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

If You Have a Technical Problem

Do not use the feedback methods listed above. Instead, do one of the following:

- Contact your IBM service representative.
- Contact IBM technical support.
- Visit the z/VM support Web page at www.vm.ibm.com/service/
- Visit the IBM mainframes support Web page at www.ibm.com/systems/support/z/

Chapter 1. Introduction

We'll begin each chapter with a brief description of its contents.

In this chapter:
- What is REXX?
- Features of REXX
- REXX and z/VM
- How to use the reading plan.

What is REXX?

The REstructured eXtended eXecutor language, or REXX language, is a versatile, easy to use structured programming language that is an integral part of z/VM. Its simplicity and free format make it a good first language for beginners. For more experienced users and computer professionals, REXX offers powerful functions, extensive mathematical capabilities, and the ability to send commands to multiple environments.

REXX is an adaptation of the CMS (Conversational Monitor System) EXEC 2 language; however, REXX instructions are quite different and easier to use. If you are a newcomer to programming, you will find that it is fairly easy to learn and write programs in REXX.

On the other hand, if you are an experienced programmer, you will find that REXX somewhat resembles PL/I. There are a number of differences, but the main difference is that a REXX program is interpreted (the language processor operates on the program directly as it runs). In PL/I, the program is compiled (translated into machine language) first, then run.

Using the REXX Compiler (which runs under CMS on z/VM), you can improve performance, maintain code security, and improve your program's documentation. The REXX Compiler translates REXX source programs into compiled programs, which run faster because they do not have to be translated while running. For additional information on the benefits of using the REXX Compiler, see the *CMS REXX Compiler General Information* manual.

Features of REXX

Ease of use: The REXX language is easy to learn and use because many instructions are meaningful English words. Unlike some programming languages that use abbreviations, REXX instructions are common words, such as SAY, PULL, IF...THEN...ELSE, DO...END, and EXIT.

Free format: REXX has few rules about format. A single instruction might span many lines or multiple instructions may be entered on a single line. Instructions need not begin in a particular column; you can skip spaces in a line or skip entire lines. You can type instructions in upper, lower, or mixed case. And there is no line numbering.

Interpreted: When a REXX program runs, its language processor reads each language statement from the source file and runs it, one statement at a time. Languages that are not interpreted must be compiled into machine language (in separate files) before they can be run.

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.

Parsing capabilities: REXX includes extensive capabilities for manipulating character strings. This lets your programs read and separate characters, numbers, and mixed input.

Debugging: When a REXX program contains an error, messages with meaningful explanations are displayed on the screen. In addition, the TRACE instruction provides a powerful debugging tool.

REXX and z/VM

By far, the most vital role REXX plays is as a procedural language for z/VM. That means a REXX program can be a kind of script for z/VM to follow. By using REXX, you can reduce long or complex or repetitious tasks to a single command or program that can be run from CMS.

REXX is a built-in feature of z/VM, so there is no installation process or separate environment. Any REXX program can call CMS and CP commands.

Note: In an XA or XC virtual machine, REXX execs and XEDIT macros can reside in storage above the 16MB line.

About Programming

Think of a program as a list of directions, like a recipe.

- First of all, the directions have a basic sequence: you cannot mix an omelet until you have broken the eggs.
- In a recipe, there are some instructions that indicate actions: chopping and mixing, for example.
- Other directions simply specify the ingredients and their proportions or measurements: a pound of almonds, two cups of flour.
- Then there are directions to tell you how to carry out other directions.
  - Some are iterative; that is, they specify repetitious actions, like stirring and kneading.
  - Some are conditional; they indicate when an action should begin or end: “bake for 30 minutes or until brown.”

And that is all a program is: a list of directions and some directions about directions.

Now, you may think of programming as a skill practiced only by computer experts, but that is not true. You need not know how a computer works to write a program any more than you have to know chemistry to bake a souffle—although even a little knowledge helps when you are troubleshooting.

You will have to take care to be very precise—in your typing as well as your thinking—because computers are extremely literal. They simply cannot overlook minor errors the way people can. Even so, by solving a program’s errors, you are sure to learn more about the job you want your program to perform. And that is useful, too.

Anyone can write a program, and anyone who uses a computer eventually finds a good reason to do so. With even a little programming know-how, you can reduce a long or repetitious series of commands into a single command. Or you can *customize* z/VM and other programs to work more the way you want them to.

You will see that programming helps you let the computer do the work it does best. That is what REXX was meant for, and this book should make REXX itself that much easier.

The Reading Plan

To assist beginners and less-experienced programmers, each subject is dealt with at three levels: Reading 1, Reading 2, and Reading 3.

Reading 1
The first reading introduces you to all the basic concepts of REXX. You will learn these concepts by writing programs suggested in the text. We expect you will also write some programs for your own use.

Reading 2
The second reading expands your knowledge of the first reading's information and teaches you the main body of the REXX language. You will also write, copy, and modify more programs.

Reading 3
The third reading contains information on features that are not often used or that are specific for special kinds of programs.

To guide you through these readings, there are headings (like the one following) at the top of each page that tell you what reading level you are on.

Reading 1

In addition, there are reminders in reverse printing at the beginning of each reading. These reminders will tell you where a particular reading begins. Following is an example.

Reading 1

There are also **bold type** reminders at the end of each reading. These reminders will tell you where the reading ends and where you should go next.

The three-level reading scheme should help maintain your interest while you build up your knowledge and skill.

If You Have Never Written a Computer Program...

If you are a newcomer to programming, you will find it fairly easy to learn and write programs in REXX. Start by reading just the basics of each chapter in sequence.

When you have read all of the basics, go back and read the remainder of each chapter to learn more about specific topics.

If You Are Already Familiar with Another Language...

Even if you are already an expert programmer, you might want to skim the basics just to get an overview of the REXX language. Or, you may prefer to read about individual topics, one at a time. Here are some areas you might want to investigate:

- If you are skilled in BASIC, you will want to note in particular the ways that REXX differs from BASIC:

  - There is no line numbering
  - There are no GOSUB or GOTO statements; use CALL and SIGNAL instead
  - REXX variables have no data type.
- If you are familiar with development languages like C and Pascal, you will find REXX somewhat similar. Again, the main difference is that a REXX program is interpreted; that is, the source code of the program is processed line by line. There is no compiling process (unless you purchase the compiler).

Exercises and Examples

As with any other language, you do not learn a programming language just by reading about it. You learn it by using it, by trying it out. That is why this book will devote a good deal of space to hands-on exercises and examples. To get the most out of this book, set it down next to your computer and:

- Test yourself with the exercises as you read.
- Examine the sample programs in the text. Type them in just as you find them here.
- Try out your own variations of each program. See if you can find a different—or better—way to do what the sample program does.

The REXX Reference

The *z/VM: REXX/VM Reference* contains the most complete description of the *grammar* of the REXX language. You will need to have your own copy of this book on hand, so you can look up any instruction or function not completely defined here.

Think of the Reference as your dictionary for REXX and this User's Guide as a kind of cookbook of simple (and a few fairly sophisticated) recipes and ideas.

Chapter 2. Starting Out with REXX

In this chapter:

Reading 1 immediately following, describes:
- How a program works
- Conversations
- Typing in a program
- Running a program
- Stopping a program
- What goes into a program
- Comments
- Keyword instructions
- Strings (in quotation marks)
- Lowercase characters (a...z)
- Blanks
- Clauses
- Syntax error.

Reading 2 on page 14, describes:
- Substitution rules.

Reading 3 on page 15, describes:
- Repeated substitution using
  - The VALUE( ) function
  - Compound Symbols
  - The INTERPRET instruction.

How a Program Works

Reading 1

We have described a REXX program as a list of instructions to your computer, something like a recipe. The program itself is simply a text file that you create with a word processor or text editor.

Sometimes the computer runs a program with no guidance. Other times it may need additional information from the operator to do its work. One way that a computer can communicate with its user is to ask questions and then compute results based on the answers typed in. As part of the *recipe*, then, the programmer (you) can include instructions that let the computer converse with whomever is using it.

Conversations

One way that a computer can communicate with a user is to ask questions and then compute results based on the answers typed in. In other words, the user has a conversation with the computer. You can easily write a list of REXX instructions that will conduct a conversation. We call such a list of instructions a program. Figure 1 on page 6 shows a sample REXX program. What it does is ask for the user to give his or her name. Then the program greets the user by the name given.

For instance, if the user types in the name `Jean`, the program replies `Hello JEAN` . Or else, if the user does not type anything in, the reply `Hello stranger!` is displayed instead.

First, you will look closely at how this program works; then you can try it for yourself.

```
/* HELLO EXEC  - A conversation    */

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

Figure 1. HELLO EXEC

This sample program consists of six statements, one to each line, called *clauses*. Briefly, the various pieces of the program are:

/* ... */
The first clause is a *comment* explaining what the program is about. All REXX programs must begin with a begin-comment delimiter (/*). Apart from this, comments are ignored.

say The second clause is a *keyword instruction*, `say`, that displays text on screen.

"Hello!..."
Anything in quotation marks after `say` is displayed just as it is. This is called a *literal string*.

pull This keyword instruction reads and stores the response entered by the program's user. This is the third clause.

who A *variable*: a name given to the place in storage where the user's response is stored.

if The fourth clause begins with the `if` instruction; it tests a given condition.

who = ""
The condition to be tested: whether the variable `who` is empty.

then Tells REXX to process the instruction that follows, if the tested condition is true.

say "Hello stranger!"
Displays `Hello stranger!` on the screen (but only if the condition is true).

else This final clause gives an alternative direction: process the instruction that follows, if the tested condition is *not* true.

say "Hello" who
Displays `Hello`, followed by whatever data is stored in `who` (if the tested condition is not true).

That is what the program does.

Typing in a Program

To type in the following program, use the same editor as you use for other work; any editor will do. This discussion will assume that you use XEDIT, the z/VM editor.

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

1. Log on to z/VM and type the command:

   ```
   xedit  hello  exec
   ```

2. Type in the program, exactly as it is shown in Figure 1, beginning with `/* HELLO EXEC  - A conversation    */`. Then file it using the XEDIT command:

   ```
   ====>  file
   ```

   The system will reply with the ready message:

   ```
   Ready;
   ```

Now your program is ready to run.

Running a Program

If you want to run a program that has a file type of EXEC, you just type in its file name. In this case, type `hello` on the command line and press Enter. Try it!

Suppose your name is Fred. Type `fred` and press Enter. `Hello FRED` is displayed.

```
Ready;
hello
Hello! What is your name?
fred
Hello FRED
Ready;
```

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 `fred` on the command line and then press Enter.
4. The PULL instruction puts the word FRED 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 FRED equal to nothing?

   ```
   "FRED" = ""
   ```

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 FRED
   ```

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

```
hello
Hello! What is your name?

Hello stranger!
Ready;
```

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.

Stopping a Program

Most of the programs we use in this book run pretty fast. But if you ever need to stop a program from running further, just enter the CMS immediate command to halt interpretation:

```
HI
```

REXX then stops running the program and returns to the CMS prompt.

Test Yourself...

Did you get your version of HELLO EXEC to run on your z/VM system? If not, check that you have correctly typed it in. If it still does not work and you cannot understand the error messages, ask for help. Usually, experienced users are happy to help a beginner. At some installations the System Support people will give help over the telephone.

Do not worry if you did not fully understand how you could use the SAY, PULL, and IF instructions. This will be explained again later.

What Goes into a Program

You can write a program in any accessed SFS directory for which you have write authority or on any minidisk accessed read/write.

Use the same editor as you use for other work; any editor will do. In this book, we shall assume that you use XEDIT, the z/VM editor.

In order to explain what goes on when you run a REXX program, we have introduced a lot of terms. There will be more, so before we go on, we will define the ones we have used so far.

Comments in Programs

When you write a program, remember that you will almost certainly want to read it over later (before improving it, for example). Other readers of your program also need to know what the program is for, what kind of input it can handle, what kind of output it produces, and so on. You may also want to write remarks about individual instructions themselves. All these things, words that are to be read by humans but are not to be interpreted, are called *comments*.

To indicate which things are comments, use:

/* to mark the start of a comment
***/** to mark the end of a comment.

The /* causes the language processor to stop interpreting; interpreting starts again only after a */ is found, which may be a few words or several lines later. For example,

```
/* This is a comment. */
say ...  /* This is on the same line as the instruction */
/* Comments may
   occupy more
   than one line. */
```

Comments with Special Meaning to CMS

The first line of a REXX program **must** start with a comment. Why?

Historically, there are three languages that can be used for writing execs for z/VM. The oldest is called CMS EXEC; the next is EXEC 2; and the latest is REXX. For technical reasons, they all have a file type of EXEC. Because each type of exec requires its own special processing, CMS must be able to distinguish one type from another. It does this by looking at the first line of the exec file. So, to tell CMS that your program is written in REXX, the first line of the file must start with a comment.

```
/* This is a REXX program.  */
```

Although /* */ is sufficient, a better use for this space is to provide a brief description of your program. You can even do it this way:

```
/*************************************
*  HELLO EXEC written by Denise B.   *
*          May 12, 1994              *
* A program to greet a user by name. *
*************************************/
```

Keyword Instructions

Words like PULL, IF, and SAY are part of the REXX language called *instructions*. The words themselves are referred to as *keywords*. You will notice that they are usually (though not always) verbs. They are the directions that tell REXX what to do with this or that information at a certain point in the program:

Say (display on screen) “hello”.
Pull (accept and store) information from the user.
If this situation true, then perform this action.

When you list these instructions in the order you want REXX to carry them out, you have a program.

Clauses

In a more formal sense, we say that a REXX program is made up of *clauses*—that is, a complete instruction, including the information it works on and any options that may be used. REXX reads each individual clause and then processes it before going on to the next. That is why we say that REXX is an *interpreted* language.

In the sample program just given, each line of text corresponds to a single clause. REXX allows exceptions to this (they are discussed in detail on page 11). For clarity’s sake, we will follow the convention of one clause to a line. This is the case for all the examples in this book, except where explicitly noted.

Literal Strings

When REXX finds a quotation mark (either " or ') it stops processing and looks ahead for the matching quotation mark. The string of characters inside the matching quotation marks is used just as it is. Hence, the name *literal string*. Examples of literal strings are:

- 'Hello'
- "Final result:"

If you want to use a quotation mark within a literal string, use quotation marks of the other kind to delimit string as a whole.

- "Don't panic"
- 'He said, "Bother"'

There is another way. Within a literal string, a pair of quotation marks (the same kind that delimits the string) is interpreted as one of that kind.

- 'Don''t panic' (same as "Don't panic")
- "He said, ""Bother""" (same as 'He said, "Bother"')

Uppercase Translation

When a clause is processed, any letters that are *not* in quotation marks are translated to uppercase. In other words, the letters

a, b, c, ... z

get changed to

A, B, C, ... Z

REXX also ignores some of the blanks that you may have written into your program, keeping only one blank between words. If this is not what you want, you should use quotation marks. Figure 2 shows an example.

```
/* SHAGGY EXEC               */

/* Example: cases and spaces */
say  a        long         story   /* Result if "a," "long," and  */
                                   /* "story" have not been       */
                                   /* assigned a value.  See      */
                                   /* Example: Setting Variables  */
                                   /* on page 20                 */

say "A        long         story"  /* Quotation marks mean to     */
                                   /* print exactly as entered.   */

say about"    "a dog
```

Figure 2. SHAGGY EXEC

When you run the SHAGGY program, here is what appears on your screen:

```
shaggy
A LONG STORY
A        long         story
ABOUT    A DOG
Ready;
```

One more point: Remember in the sample program how the user's input `fred` got changed to `FRED`? That had nothing to do with the process we just described. Rather, that particular translation is a feature of the `pull` instruction, which always converts user input to uppercase. The practical value of this is that the user can type in any combination of uppercase and lowercase letters.

Variables

When we need to work with changeable information (such as the user's name in HELLO EXEC), we can reserve a place in storage. That memory niche is called a *variable*.

When REXX processes a clause containing a variable, it substitutes the variable name with the stored data. That is how the stored entry `FRED` took the place of the variable name `who` in our first example.

We will cover variables in more depth in Chapter 3, “Variables,” on page 17.

Clauses

Your REXX program consists of a number of clauses. A clause can be:

1. An *instruction* that tells the language processor to do something; for example,

   ```
   say "the word"
   ```

   In this case, the language processor will display `the word` on the user’s screen.
2. An *assignment*; for example,

   ```
   Message = 'Take care!'
   ```

   This means that the string `Take care!` is to be put into a place called `MESSAGE` in the computer’s storage.

   Because `MESSAGE` can be given different values in different parts of the program, it is called a *variable* (discussed in Chapter 3, “Variables,” on page 17).
3. A *label*, which is a name followed by a colon; for example,

   ```
   MYSUB:
   ```

   (Labels are discussed in “The CALL Instruction” on page 151 and “The SIGNAL Instruction” on page 159).
4. A *null clause*, such as a completely blank line, or

   ```
   ;
   ```

   Note: Anything that is not one of these (an instruction, an assignment, a label, or a null clause) is taken to be:
5. A *command*; for example,

   ```
   erase hello exec
   ```

   Commands are passed to CMS (or other environments; discussed in “Issuing Commands to CMS and CP” on page 99).

When Does a Clause End?

It is sometimes useful to be able to write more than one clause on a line, or to extend a clause over many lines. The rules are:

- Usually, each clause occupies one line.
- If you want to put more than one clause on a line you must use a semicolon (;) to separate the clauses.
- If you want a clause to span more than one line you must put a comma (,) at the end of the line to indicate that the clause continues on the next line. The comma cannot, however, be used in the middle of a string or it will be interpreted as part of the string itself. The same situation holds true for comments.

What will you see on the screen when this exec is run?

```
/* RAH EXEC                                      */

/* Example: there are six clauses in this program */
say "Everybody cheer!"
say "2"; say "4"; say "6"; say "8";
say "Who do we",
"appreciate?"
```

Figure 3. RAH EXEC

(If you are not sure, use XEDIT to create a file called RAH EXEC and try out the program.)

Syntax Errors

The rules governing the arrangement of words and punctuation marks in a language are called its *syntax*. The actions we have been describing are part of the syntax for the REXX language. If REXX encounters something that does not make sense according to its syntax, it stops running your program and returns to CMS. REXX then displays the incorrect instruction line and an *error message* saying what is wrong.

We will go back to our sample program. Suppose we alter it to read so:

```
/* HELLO2 EXEC    */

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

Figure 4. HELLO2 EXEC with a syntax error

There is a syntax error here. We have forgotten to put a */ at the end of the third comment. When we run the program, what appears on the screen is:

```
hello
Hello! What is your name?
     3 +++ pull who               /* Get the answer!if who = "" then say "Hello
stranger"else say "Hello" who
DMSREX453E Error 6 running HELLO EXEC, line3: Unmatched "/*" or quotation mark
Ready(20006);
```

Here is what the *error message* means:

- `3 +++` means the language processor was interpreting the clause that started on line 3. (The clause itself is displayed following the +++.)
- `Error 6` gives the REXX error number.

  The error message gives you a good idea what went wrong. If you need more information, look up `Error 6` in the list of error messages in the back of your *z/VM: REXX/VM Reference*.
- `Ready(20006);` is the return code that the language processor returns to CMS.

Leaving out a final quotation mark at the end of a literal string causes REXX to issue a similar error message.

Test Yourself...

1. Read the following program carefully. Take a pencil and write down what each word is and what REXX will do with it, depending on how the user responds.

```
/* WHOAMI EXEC    */

/* Who Am I? game */
say "What is my name?"
pull guess
if guess = "REXX" then say "You win!"
else say no but guess "is a good guess."
```

   Now create a file called WHOAMI EXEC and try out the program. Did everything happen as you expected? If not, read this chapter again and then study the explanation below.

2. This next program has an error in it. Type the program in and run it.

```
/* TROUBLE EXEC            */

/* Example: a syntax error */
say Unfortunately, there is an error here
```

   Use the error number to look up the cause of the error in your *z/VM: REXX/VM Reference*. Correct the error and test the program again.

Answers:

1. The syntax of the program WHOAMI EXEC is:
   - `/* WHOAMI EXEC    */` is a *comment* describing the program. (The first line of a REXX program must start with a comment.)
   - The `say` instruction displays, `What is my name?`
   - `pull` is another instruction. The variable `guess` gets the value entered by the user.
   - The `if` instruction checks to see if the user entered `REXX`.

     Note: Because `pull` translates the entry to uppercase, the user can type it in any combination of uppercase and lowercase letters (`rexx`, `Rexx`, `rExX`, and so on).
   - If so (if `GUESS = 'REXX'`), then `You win!` is displayed.
   - If the user enters something other than `REXX`, then the clause beginning with `else say` is interpreted and the result is displayed.
     – `no but` is changed to uppercase. It is a string, but it is not in quotation marks. It is displayed as: `NO BUT`
     – `guess` is the name of a variable. The user's entry, translated to uppercase, is substituted.
     – `"is a good guess."` is a literal string. It is displayed just as it is, even though `GUESS` is also the name of a variable.

   Here is what actually appears on the screen if the user guesses right:

```
whoami
What is my name?
rexx
You win!
Ready;
```

But if the user guesses wrong:

```
whoami
What is my name?
spot
NO BUT SPOT is a good guess.
Ready;
```

Now, what happens if the user types nothing, but just presses Enter?

```
whoami
What is my name?

NO BUT  is a good guess.
Ready;
```

The variable `GUESS` was empty, so the `say` instruction displayed nothing. Only two blanks remain—the ones before and after the variable in the program.

That last response does not make much sense. See if you can think of a way to fix WHOAMI EXEC so that it does. (A hint: take another look at HELLO EXEC).

2. The error number for the program TROUBLE EXEC is 37. The error message reads `Unexpected ","or ")"`.

   Obviously, REXX found a comma where it did not belong. What may not be obvious is what to do about it. When you get a message like this, turn to the *z/VM: REXX/VM Reference*. In the back of the book, you will find a list of error messages and explanation of their causes.

   In this case, the comma has a special meaning for REXX when it is used outside of a literal string (this is described on page 11). For a comma to be used as it is intended here, it would have to be enclosed in matching quotation marks.

```
/* TROUBLE2 EXEC                 */

/* Example: a syntax error fixed */
say "Unfortunately, there is an error here"
```

Reading 1 continues in Chapter 3, “Variables,” on page 17.

Reading 2 begins here.

If you would like to review Reading 1 of this section, read Chapter 2, “Starting Out with REXX,” on page 5.

If you wish to start Reading 2, continue on.

Substitution Rules

When replacing the names of variables with their values, the language processor does *not* look at the words it substitutes to see if they are also the names of variables.

For example:

```
food = meat
meat = steak
steak = sirloin
say "Buy us some" food       /* says "Buy us some MEAT" */
```

This rule applies to simple symbols. Compound symbols, discussed on page 15 (and in more detail on page 24), provide a further level of substitution.

Reading 2 continues in Chapter 3, “Variables,” on page 17.

Repeated Substitution

Reading 3 begins here.

For repeated substitution, you can use

- The VALUE( ) function
- Compound symbols
- The INTERPRET instruction.

The VALUE( ) Function

To specify a computed value as the name of a variable, use the VALUE( ) function. The example on page 14 could be redesigned like this:

```
/* ERRAND EXEC                                    */

/* Example: the name of the name of ...           */
food = meat
meat = steak
steak = "sirloin"
say "Buy us some"  value(food)       ||,
    "; I mean some"  value(value(food))"."

 /* says "Buy us some STEAK; I mean some sirloin." */
```

Figure 5. ERRAND EXEC

Compound Symbols

Many programmers who use REXX are familiar with compound symbols, but only a few have ever used the VALUE( ) function. Therefore, when you find a program that can be coded using either method, choose compound symbols.

```
/* VENTS EXEC                                           */

/* Part of a ventilation monitor. The user can query    */
/* settings of certain ventilators.                     */

vent.front.door = open; vent.back.door = shut
vent.front.window = open; vent.back.window = open

do until noun ¬= ""
   say "Enter command"
   pull verb adjective noun      /* user enters         */
end                              /*  "query front door" */

if abbrev("QUERY",verb,1) then
say adjective noun "is" vent.adjective.noun

           /* says "FRONT DOOR is OPEN" */
```

Figure 6. VENTS EXEC

The same example could have been coded:

```
frontdoor = open; ...

say adjective noun "is" value(adjective||noun)
```

This is less familiar, though still readable.

The INTERPRET Instruction

To use a computed value as though it were a line in an exec file, use the INTERPRET instruction.

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

The specified expression is evaluated and the result is interpreted. (For a complete description, see the *z/VM: REXX/VM Reference*.)

Here is an example:

```
/* MATH EXEC           */

/*  Simple calculator  */
say "Please enter an expression to be evaluated."
say "Enter a null line to end:"
do forever
   parse pull expr
   if expr='' then leave
   interpret "Say" expr
end
```

Figure 7. MATH EXEC

To avoid confusing anyone reading your programs, it is better not to use INTERPRET in situations where a simple VALUE( ) or a CALL would do instead.

Reading 3 continues in Chapter 3, “Variables,” on page 17.

Chapter 3. Variables

Variables are a means of handling changeable information by representing it in terms of *symbols*. In this chapter, we will see why that is important when writing programs; then we will describe the basic rules for using variables.

In this chapter:

Reading 1 immediately following, describes:

- What a variable is and how to assign values to them.

Reading 2 on page 23, describes:

- How to use variables as symbols
- How to use compound symbols to build arrays
- How to avoid duplicate names.

Reading 3 on page 30, describes:

- How to limit the scope of variable names with the PROCEDURE instruction
- How to find out whether a particular symbol is the name of a variable
- How to DROP a variable
- How to build arrays with more than one dimension.

What Are Variables?

Reading 1

One basic requirement of any program is that it must work with unknown information—unknown, that is, when the program is written.

For example, you could write a program that simply totals a fixed list of numbers, like this:

```
/* TWOPLUS3 EXEC            */
/* the sum of two and three */
say "2 + 3 equals" 2 + 3
```

Figure 8. TWOPLUS3 EXEC

and you would get the result 5 every time you ran it.

But that is all you would get: a reliable program, but not a very useful one. More useful is a program that can process different information each time it is run. You do this by using variables to stand in for values to be processed. A *variable* is a symbol (one or more characters) that represents a value.

Take as an example this program, the simplest calculator you will ever see:

```
/* ADD2NUM EXEC           */
/* the sum of two numbers */
say "Enter a number:"
pull first             /* waits for entry */
say "Enter another number:"
pull second            /* waits for entry */
say "The sum is" first + second
```

Figure 9. ADD2NUM EXEC

Here is what it looks like when you run it:

```
add2num
Enter a number:
25
Enter another number:
32
The sum is 57
Ready;
```

We used two PULL instructions to let the user enter the two numbers to be added and then *assign* (store) them in the variables *first* and *second*. The SAY instruction displays the sum of the two.

You can see for yourself what this program does if the user enters only one number. For this program to be anywhere as reliable as TWOPLUS3 EXEC, it will have to make certain that the user has entered the input numbers properly. That is an important topic in itself, but one we will leave for later chapters.

Right now, we will look at how you can use variables to manipulate information.

Names and Values

The information stored in a variable is called its *value*. The value might be one or more words of text, or it might be a number. It might be nothing at all.

A variable's value can change any time you want it to. It can be different each time the program is run, or it may change many times in a single run.

But no matter how the value of a variable changes, the variable's *name* stays the same. We made up the names we chose for the variables, *first* and *second*. The names need only be meaningful to the programmer—you.

You can think of a variable, then, as simply the name for the kind of values you want it to hold.

Assignments

An instruction that stores a value in a variable or changes its value is called an *assignment*.

The simplest form of assignment is the equals sign, a REXX clause of the form

```
name = value
```

where:

name is the name you give the variable

value is the value it will hold.

In more formal terms, the syntax of an assignment looks like this:

symbol = expression

where:

symbol is a valid variable name

expression
is the information to be stored: a number, a string, or some calculation that you want REXX to perform.

We will cover expressions in more detail in the next chapter. For the time being, all you need to know is that REXX first *evaluates* (computes) the `expression` and then puts the result of that evaluation into the variable called `symbol`. In plain English, the assignment instruction says:

> “Evaluate the `expression` and store the result as `symbol`”.

In an assignment, then, you name a variable and give it a value. Here are some examples:

- To give the variable called `TOTAL` the value `0`, use this kind of assignment:

  ```
  total = 0
  ```

- To give another variable, called `PRICE`, the same value as `TOTAL`, assign the value this way:

  ```
  price = total
  ```

- To give the variable called `TOTAL` a new value, namely the old value of `TOTAL` plus the value of `SOMETHING`, use the assignment:

  ```
  total = total + something
  ```

- Here is a different kind of assignment, one we have already used:

  ```
  pull something
  ```

  This PULL instruction gives the variable `SOMETHING` a value that the user enters while the program is running.

Displaying a Variable’s Value

To display a variable’s value at any given point in a program, use the SAY instruction.

```
/* ASSIGN EXEC      */
/* some assignments */
amount = 100                    /* assigns 100 to AMOUNT         */
money = "dollars"               /* assigns "dollars" to MONEY    */
say amount money                /* displays "100 dollars"        */
amount = amount + 25            /* adds 25 to AMOUNT             */
say amount money                /* displays "125 dollars"        */

/* Now get some input from the user */

say "Type a line, then press Enter" /* prompts the user to type      */
pull anything                   /* waits for user to press Enter */
say "You typed:" anything       /* displays the input on screen  */
```

Figure 10. ASSIGN EXEC

What if you SAY a variable that has not yet been assigned a value? In some languages, you would get an error. In REXX, the default value of a variable is its own name, converted to uppercase letters.

```
/* NOASSIGN EXEC                     */
/* display unassigned variables      */
say amount      /* displays "AMOUNT" */
say first       /* displays "FIRST"  */
say price       /* displays "PRICE"  */
say who         /* displays "WHO"    */
```

Figure 11. NOASSIGN EXEC

Note: There is another way to peek at the value of a variable while a program is running—the TRACE instruction, used for correcting programs. We will look at it in Chapter 4, “Expressions,” on page 35.

Choosing Names for Variables

You can choose any symbol as the name of a variable, with these restrictions:

1. The first character must be one of:

   A-Z a-z @ # $ ¢ ! ? _

   Note: The language processor translates lowercase characters to uppercase before using them.
2. The rest of the characters may be any of the following:

   A-Z a-z @ # $ ¢ ! ? _ . or 0-9

   But you should not use a period unless you understand the rules for “Compound Symbols” on page 24, described in Reading 2 of this chapter.

Example: Setting Variables

To make your program easy to understand, use ordinary English words for the names of variables, as in Figure 12.

```
/* MCDONALD EXEC                       */
/* Example:  farmyard noises explained */
say "What animal?"
pull beast                /* user enters name of animal */
select
   when beast = "LAMB"   then noise = "Baah! Baah! Baah!"
   when beast = "DONKEY" then noise = "Eeyore!"
   when beast = "PIG"    then noise = "Grunt! Grunt!"
   otherwise                  noise = "I don't exist"
end
say 'The' beast 'says' noise
```

Figure 12. MCDONALD EXEC

Use XEDIT to create this file called MCDONALD EXEC and try it out. Did it work? If not, study the error messages and make sure you copied everything correctly.

In the MCDONALD EXEC `BEAST` and `NOISE` were the names of variables.

say displays a string on the screen.

pull causes the program to pause. The user may now type something in. When the user presses Enter, whatever the user typed in is put into the variable `BEAST` and the program continues.

select
chooses one of four assignment instructions, according to the value of the variable BEAST. The chosen instruction sets the variable NOISE.

```
noise =  ...
```

(We shall discuss how to use select, when, then and otherwise later, in "The SELECT Instruction" on page 134.)

end indicates that this is the end of the select. (To make the program easier to read, the instructions between the select and the end are indented three spaces to the right.)

say uses the symbols BEAST and NOISE to obtain the values of these variables and to display them on the screen.

When the language processor finds a symbol (a word that is not in quotation marks) it looks to see if the symbol is the name of a variable; that is, whether it has been given a value. If so, the language processor substitutes that value for the symbol. If not, it translates the symbol to uppercase and uses that.

The idea of a variable (such as NOISE in Figure 12) is very important in computing. However, before we can make much more use of it we shall have to find out how *expressions* are handled. This is the topic of the next chapter.

Test Yourself...

Which of the following could be used as the name of a REXX variable?

1. DOG
2. K9
3. 9T
4. nine_to_five
5. #7

Answers:

1. OK
2. OK
3. Incorrect, because the first character is a numeric digit.
4. OK, same as NINE_TO_FIVE
5. OK

Other Assignments

You can also use variables to store unknown information—unknown, that is, while you are writing the program.

Assigning User Input

One such use for variables that we have already encountered is as a holding place for information supplied by the user. Here are two keyword instructions commonly used for this purpose.

The PULL Instruction: This instruction pauses the running of a program to let the user type one or more items of data which are then assigned to variables. For example we used PULL in Figure 9 on page 18 to get two numbers to add:

```
say "Enter a number:"
pull first            /* waits for entry */
say "Enter another number:"
pull second           /* waits for entry */
```

Each PULL instruction pauses the program so the user can type a number and press Enter. It then assigns the entry to the variable named in the instruction.

You can also use PULL to collect more than one item in an entry, so long as the items are separated by spaces. We could replace the four lines above with:

```
say "Type two numbers (leave a space between) and press Enter"
pull first second
```

Here too, PULL pauses the program so the user then can then type the two numbers to add. When the user presses Enter, PULL reads the two numbers and assigns them, in the order they were typed, to the list of variables (`first` and `second`). This process of reading and breaking up information is called *parsing*, and we will devote much discussion to that in later chapters.

The ARG Instruction: Another way to assign data from the user is with ARG. It works in the same manner as PULL, except that items are entered at the command prompt along with the program name. Our mini-calculator in Figure 9 on page 18 could also work this way:

```
/* ADD EXEC                            */
/* the sum of two numbers, this time    */
/* entered at the command prompt        */
arg first second    /* collects entries */
say "The sum is" first + second
```

Figure 13. ADD EXEC

Here is how it looks when you run it:

```
add 20 33
The sum is 53
Ready;
```

Notice that with ARG, there is no pause because the numbers are entered along with the command that starts the program.

Assigning an Expression Result

Take another look at the program ASSIGN EXEC in Figure 10 on page 19. The instruction `amount = amount + 25` demonstrates how variables can represent another kind of unknown information: data that must be calculated or otherwise manipulated. You can simply assign to a variable the result of a calculation or *expression*. Here is another example:

```
/* AREAS EXEC                      */
/* area of a 3 by 5 in. rectangle */
area = 3 * 5
say area "sq. in."                   /* displays "15 sq. in."      */

/* area of a 5 in. circle         */
diameter = 5
radius = diameter/2
area = 3.14 * radius * radius
say area "sq. in."                   /* displays "19.6250 sq. in." */
```

Figure 14. AREAS EXEC

Simple enough. But REXX expressions can have very complex forms as well, and they can work with all kinds of information. They are our topic for the next chapter.

Reading 1 continues in Chapter 4, “Expressions,” on page 35.

Variables as Symbols

Reading 2

Variables are part of a class of REXX language elements called *symbols*. These include:

- REXX keywords and instructions
- Labels used to call internal subroutines (see the discussion of the CALL instruction on page 150)
- Constants
- Variables.

REXX uses a symbol’s context to determine if it is to be taken as a keyword or a label or a variable. For each symbol it encounters, REXX takes the following steps to determine how it will be handled:

1. Is the symbol the very first token in a clause? If so...
   a. If it is followed by an equal sign (=), then the clause is an assignment instruction. The symbol is a variable, and is assigned the expression that follows the equal sign.
   b. If it is followed by a colon (:), then it is a label, signaling the beginning of a subroutine.
   c. Is the symbol among the list of REXX keyword instructions?
2. Is the symbol a keyword used in a control structure? (such as WHILE or THEN; see Chapter 8, “Control,” on page 129). If so, REXX interprets the keyword accordingly.
3. Is it a constant (an unchangeable value)?

If none of these steps determine how the symbol is to be handled, REXX evaluates it as a variable and substitutes its stored value for the variable name.

Constants and Variables

Symbols that begin with a digit (0-9), a period, or a sign (+ or -) are *constants*. They cannot be assigned new values and therefore cannot be used as variables. Here are some examples of constants:

77	a valid number
.0004	begins with a period (decimal point)
1.2e6	Scientific notation (equal to 1,200,000)
42nd	Not a valid number; its value is always `42ND`

Note that:

- A symbol that begins with a number cannot be assigned a different value; it cannot be a variable.
- The default value for a symbol is its own name, translated into uppercase letters. A variable that has not been assigned a value contains this default value.
- All valid numbers are constants, but not all constants are valid numbers. The symbol `3girls` is not a valid number, but neither can it be used as a variable name; its value is always `3GIRLS`.

There is a special class of symbols in which variables and constants are combined to create groups of variables for easy processing. These are called *compound symbols*.

Compound Symbols

A variable containing a period is treated as a *compound symbol*. Here are examples of compound symbols:

```
fred.3
row.column
array.I.J.
gift.day
```

Stems and Tails

The *stem* of a compound symbol is the portion up to and including the first period. That is, it is a valid variable name that ends with a period.

The stem is followed by a *tail* comprising one or more valid symbols (constants or variables), separated by periods.

Derived Names

You can use compound symbols to create an array of variables that can be processed by their *derived names*. Take for example this collection:

```
gift.1 = "A partridge in a pear tree"
gift.2 = "Two turtle doves"
gift.3 = "Three French hens"
gift.4 = "Four calling birds"
...
```

Now, if we know what day it is, we know what gift will be given. Suppose, we also assign a variable called `DAY` a value of 3.

```
day = 3
```

Then this instruction:

```
say gift.day
```

displays `Three French hens` on the screen. Sounds a bit tricky, but here is what happens:

- REXX recognizes the symbol `gift.day` as compound because it contains a period.
- REXX checks to see if the characters following the period form the name of a variable; in this case, it is the variable name `day`.
- The value of `day` is substituted for its name, producing a *derived name* of `GIFT.3`.
- And the value of the variable `GIFT.3` is the literal string `Three French hens`.

But note: If `day` had never been given a value, its value would have been its own name, `DAY`, and the derived name of the compound symbol `gift.day` would have been `GIFT.DAY`.

A collection of consecutively numbered variables like this is sometimes called an *array*. Figure 15 on page 25 is an example of our gift-giver's array in action.

```
/* TWELVDAY EXEC                         */
/* What my true love sent ...            */

/* First, assign the gifts to the days   */
gift.1  = 'A partridge in a pear tree'
gift.2  = 'Two turtle doves'
gift.3  = 'Three French hens'
gift.4  = 'Four calling birds'
gift.5  = 'Five golden rings'
gift.6  = 'Six geese a-laying'
gift.7  = 'Seven swans a-swimming'
gift.8  = 'Eight maids a-milking'
gift.9  = 'Nine ladies dancing'
gift.10 = 'Ten lords a-leaping'
gift.11 = 'Eleven pipers piping'
gift.12 = 'Twelve drummers drumming'

/* list all gifts from the 12th day to   */
/* the 1st day Rrefer to the discussion  */
/* of loops on page 53. */
do day=12 to 1 by -1
 say gift.day
end

/* now display the gift for a chosen day */
say "Enter a number from 1 to 12."
pull day

/* check for proper input */
/* See page 57. */
if ¬datatype(day,"n") then      /* if the entry is not a number */
  exit                          /* then exit the program        */

if day < 1 | day > 12 then      /* same if it is out of range   */
 exit

say gift.day
```

Figure 15. TWELVDAY EXEC

Creating an Array

You can refer to all the variables in an array by using its stem. It is often convenient to set all variables in an array to zero using their stem.

The example in Figure 16 on page 26 shows how compound symbols can collect and process data. In the first part of the program, the first player’s score is entered into `SCORE.1`, the second player’s into `SCORE.2`, and so on. Thus, using compound symbols, the array of `SCORE`s is processed to give the result in the required form.

```
/* GAME EXEC                                            */
/* This is a scoreboard for a game.  Any number of      */
/* players can play.  The rules for scoring are these:  */
/*                                                      */
/* Each player has one turn and can score any number of */
/* points;  fractions of a point are not allowed.  The  */
/* scores are entered into the computer and the program */
/* replies with                                         */
/*                                                      */
/*      the average score (to the nearest hundredth of  */
/*                           a point)                   */
/*      the highest score                               */
/*      the winner        (or, in the case of a tie,    */
/*                           the winners)               */
/*------------------------------------------------------*/
/* Obtain scores from players                           */
/*------------------------------------------------------*/
say "Enter the score for each player in turn.  When all"
say "have been entered, enter a blank line!"
say
n=1
do forever
  say "Please enter the score for player "n
  pull score.n
  select
    when datatype(score.n,"whole") then n=n+1
    when score.n="" then leave
    otherwise  say "The score must be a whole number."
  end
end
n = n - 1                  /* now n = number of players */
if n = 0 then exit
/*------------------------------------------------------*/
/* compute average score                                */
/*------------------------------------------------------*/
total = 0
do player = 1 to n
   total = total + score.player
end
                                     /* continued ...   */
```

Figure 16. GAME EXEC (Part 1 of 2)

```
say "Average score is",
    format(total/n,,2,0) /* format "total/n" with       */
                         /*   no leading blanks,        */
                         /*   round to 2 decimal places,*/
                         /*   do not use exponential    */
                         /*     notation                */
/*------------------------------------------------------*/
/* compute highest score                                */
/*------------------------------------------------------*/
highest = 0
do player = 1 to n
   highest = max(highest,score.player)
end
say "Highest score is" highest
/*------------------------------------------------------*/
/* Now compute:                                         */
/*  * W, the total number of players that have a score  */
/*    equal to HIGHEST                                  */
/*  * WINNER.1, WINNER.2 ... WINNER.W, the id-numbers   */
/*    of these players                                  */
/*------------------------------------------------------*/
w = 0                      /* number of winners         */
do player = 1 to n
   if score.player = highest then do
      w = w + 1
      winner.w = player
   end
end
/*------------------------------------------------------*/
/* announce winners                                     */
/*------------------------------------------------------*/
if w = 1
   then say "The winner is Player #"winner.1
else do
   say "There is a draw for top place.  The winners are"
   do p = 1 to w
      say "     Player #"winner.p
   end
end
exit
```

Figure 16. GAME EXEC (Part 2 of 2)

Test Yourself...

1. Write a program to say the days of the week repeatedly, as:
   Sunday
   Monday
   Tuesday
   Wednesday
   Thursday
   Friday
   Saturday
   Sunday
   Monday
   ...

   You can use the CMS command, HI or HX, to stop it.
2. Extend this program to say the days of the month, as:
   Sunday 1st January
   Monday 2nd January

...

Answers:

1. The simplest solution is:

```
/* DAYS1 EXEC */

/* to say the days of the week indefinitely */
do forever
   say "Sunday"
   say "Monday"
   say "Tuesday"
   say "Wednesday"
   say "Thursday"
   say "Friday"
   say "Saturday"
end
```

   Note: To stop this exec, type `HX`. This is the immediate command to halt execution.

   But, in view of the next question, consider a solution that uses compound variables, like this:

```
/* DAYS2 EXEC */

/* to say the days of the week indefinitely */
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"
j=0
do forever
   j = j + 1
   say day.j
   if j = 7 then j = 0
end
```

2. This idea can be extended, like this:

```
/* MONTH1 EXEC */

/* to say the days of the month for January */
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"
do dayofmonth = 1 to 31
   dayofweek = (dayofmonth+6)//7 + 1
   select
      when dayofmonth = 1 then th = "st"
      when dayofmonth = 2 then th = "nd"
      when dayofmonth = 3 then th = "rd"
      when dayofmonth = 21 then th = "st"
      when dayofmonth = 22 then th = "nd"
      when dayofmonth = 23 then th = "rd"
      when dayofmonth = 31 then th = "st"
      otherwise th = "th"
   end
   say day.dayofweek dayofmonth||th "January"
end
```

Avoiding Duplicate Names

In any program, it is important not to use a symbol in more than one way. Here is an extreme example. The SAY expressions show how the values of the variables `LINE` and `DATA` change, during execution.

```
/* MESSY EXEC         */
/* NOT a good example */
do line = 1 to 10
   say line
   say Enter a line of data
   pull line
   say line
   data = data line
   say data
   line = length(data)
   say line
end line
say Done
```

Figure 17. MESSY EXEC

Looking at some sample input to this exec will help in understanding why you should not use a symbol in more than one way. If you enter

```
melvin
```

as input to this exec, the following will be displayed:

```
messy
1
ENTER A 1 OF DATA
melvin
MELVIN
```

```
DATA MELVIN
11
DONE
Ready;
```

Notice how the values of the variables `LINE` and `DATA` change. Try running the exec again but with different input.

From this horrid mess you can learn that:

- It is safer and neater to put what you want to SAY in quotation marks.

  A good example of this can be seen in the result from MESSY EXEC. Because the expression

  ```
  Enter a line of data
  ```

  is not enclosed in quotation marks, the symbol, `LINE`, is evaluated and its value is displayed instead. For example,

  ```
  Enter a 1 of data
  ```
- Each symbol should be used for only one purpose.

  In the MESSY EXEC, the language processor cannot keep track of all the different uses of the symbols `LINE` and `DATA`. Thus, the program does not run correctly.

  For small programs it is fairly easy to limit the use of a symbol to one purpose; however, it is more difficult to do this for large programs. We shall return to this subject in the next reading of this chapter.

Reading 2 continues in Chapter 4, “Expressions,” on page 35.

How Much Should You Tell Your Subroutine?

Reading 3

When you are writing a subroutine, you may not be aware of the names of all the variables in the main program. Of course, you could check by reading through the whole program every time you wanted to invent a new name. But this is tedious and prone to error.

The PROCEDURE Instruction

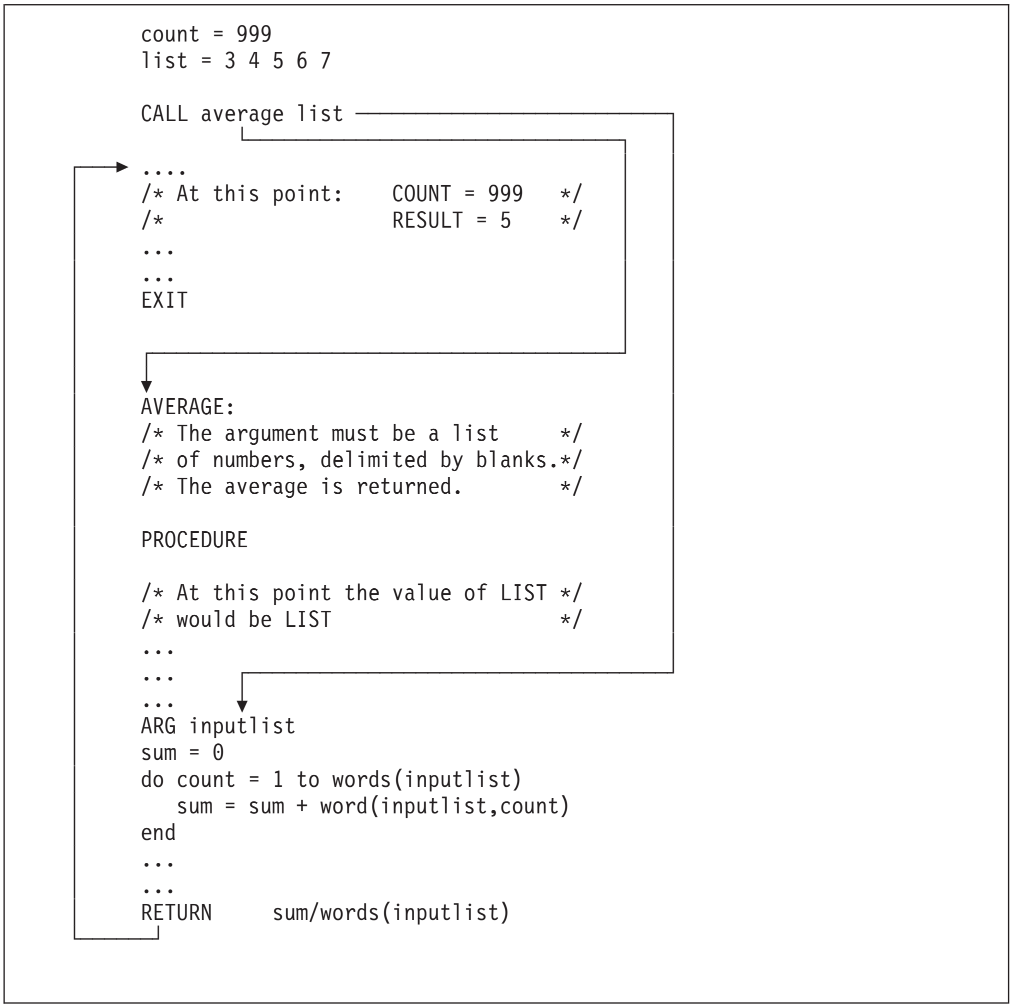
To make the language processor forget, for the time being, all the variables it knows, use the PROCEDURE instruction.

After this instruction has been run, new variables can be created that will be regarded as *different*, even if some of them have the same names as variables that existed before the PROCEDURE instruction was run.

When a RETURN instruction is executed, the new variables are forgotten and the original variables are remembered.

A PROCEDURE instruction can only be used within an internal routine; within that routine, it can only be used one time. If the PROCEDURE instruction is used in an internal routine, it *must* be the first instruction in the routine. For further details on the PROCEDURE instruction, see the *z/VM: REXX/VM Reference*.

In this next example, `COUNT` is used for two separate purposes.

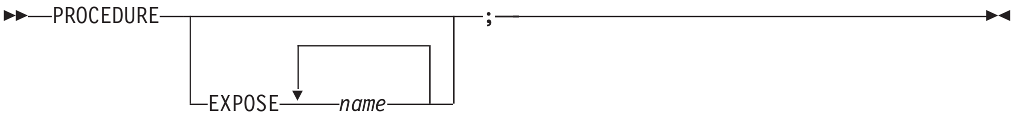


```
count = 999
list = 3 4 5 6 7

CALL average list

....
/* At this point:    COUNT = 999   */
/*                   RESULT = 5    */
...
...
EXIT

AVERAGE:
/* The argument must be a list     */
/* of numbers, delimited by blanks.*/
/* The average is returned.        */

PROCEDURE

/* At this point the value of LIST */
/* would be LIST                   */
...
...
...
ARG inputlist
sum = 0
do count = 1 to words(inputlist)
   sum = sum + word(inputlist,count)
end
...
...
RETURN     sum/words(inputlist)
```

Figure 18. COUNT Used for Two Different Purposes

The PROCEDURE EXPOSE Instruction

To share a limited set of variables between the main routine and the subroutine (leaving all the other variables protected) use:



where:

name is the name of a variable to be shared. For further details, see the PROCEDURE instruction in your *z/VM: REXX/VM Reference*.

For more information about sharing variables, see the GLOBALV command in the *z/VM: CMS Commands and Utilities Reference*.

The Existence of Variable Names

You can find out if a symbol already exists with the SYMBOL( ) function or unassign a variable with the DROP instruction.

The SYMBOL( ) Function

It is sometimes useful to know whether a symbol has already been used as a name of a variable. The SYMBOL( ) function returns:

BAD if the argument is not a valid symbol

VAR if the variable exists

LIT if the variable does not exist, or if the argument is a constant symbol, such as 3D.

This example shows how to make sure that `payment` is never added to an empty string, which would cause a syntax error.

```
if symbol("CASH") = "LIT" then cash = 0
cash = cash + payment
```

Notice what happens if the argument of SYMBOL( ) is not in quotation marks.

```
cash = 100
say symbol(CASH)        /* says "LIT", because 100 is   */
                        /* a literal                    */
say symbol("CASH")      /* says "VAR", because CASH is  */
                        /* the name of a variable       */
```

Without the enclosing quotation marks `CASH` is treated as a variable, and its value is substituted before the function is performed.

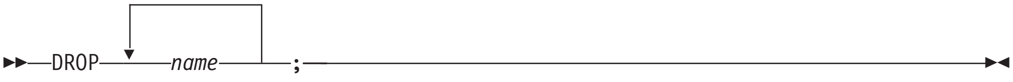
Finally, an example:

```
/* TICKETS EXEC                      */
/* Example:  the SYMBOL( ) function */
firstclass = 120
secondclass = 80
do until symbol(ans||class) = "VAR"
   say "What class?  First or second."
   pull ans
end
say "That will be" value(ans||class) "dollars, please."
```

Figure 19. TICKETS EXEC

The DROP Instruction

Usually, the place where you want the language processor to temporarily hide variables is at the beginning of subroutines. For this you can use the PROCEDURE instruction (described earlier). But in other situations, you may want the language processor to forget about a variable altogether. In this case, use the DROP instruction.



where:

name is name of a variable to be dropped.

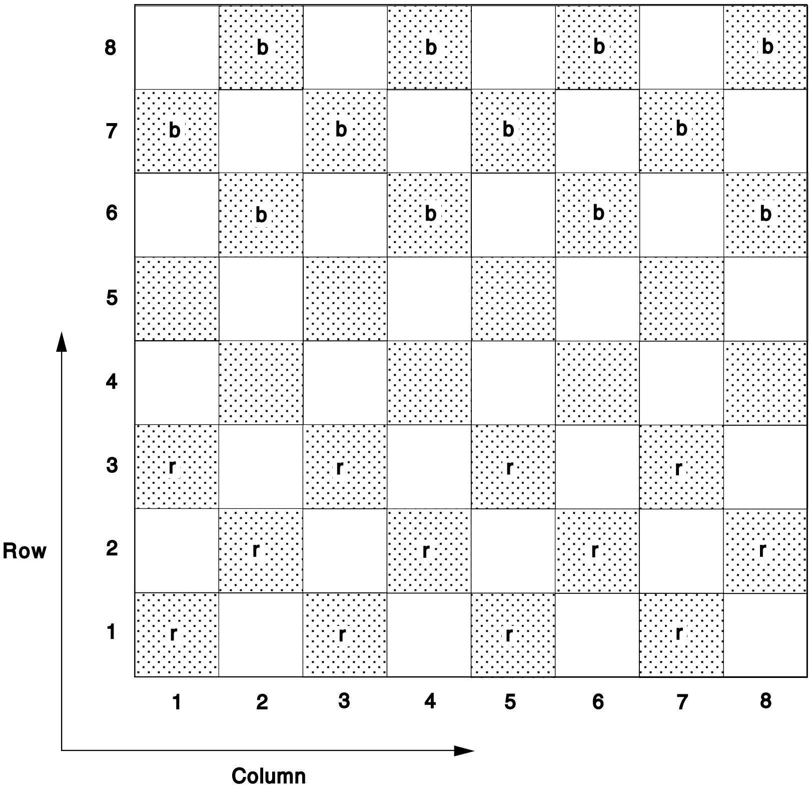
You can drop more than one variable using a single DROP instruction. You can also drop all the elements of an array by specifying the stem of the array. For example:

```
DROP player.
```

Once dropped in this way, the old values of the variables cannot be *remembered*.

Arrays with More Than One Dimension

You can have more than one period in a compound symbol. For example, here is the beginning of a program that sets up a board for playing checkers. BOARD is a 2-dimensional array, 8 squares by 8 squares. The squares on the board are called BOARD.ROW.COL and there are 64 of them altogether. The picture shows how the “men” are set out at the start of the game.



8		b		b		b		b
7	b		b		b		b	
6		b		b		b		b
5								
4								
3	r		r		r		r	
2		r		r		r		r
1	r		r		r		r	
	1	2	3	4	5	6	7	8

```
/* CHECKERS EXEC                                        */
/* This program segment sets up a board on which the    */
/* game of checkers can be played.                      */
/* In the internal representation, Red's "men" are      */
/* represented by the character "r" and red's "kings"   */
/* by the character "R".  Similarly, Black's "men" and  */
/* "kings" are represented by "b" and "B".              */
/*------------------------------------------------------*/
/* Clear the board                                      */
/*------------------------------------------------------*/
board. = " "
/*------------------------------------------------------*/
/* Set out the men                                      */
/*------------------------------------------------------*/
do col = 1 by 2 to 7
   board.1.col = "r"
   board.3.col = "r"
   board.7.col = "b"
end
do col = 2 by 2 to 8
   board.2.col = "r"
   board.6.col = "b"
   board.8.col = "b"
end                         /* Now the board is set up. */
```

Figure 20. CHECKERS EXEC

Reading 3 continues in Chapter 4, “Expressions,” on page 35.

Chapter 4. Expressions

An *expression* is something that can be computed. In your *z/VM: REXX/VM Reference*, you will find model instructions like:

```
symbol = expression
SAY expression
IF expression THEN ...
```

When you are writing instructions in one of your programs, you can replace the word `expression` with any expression that can be evaluated. Here are some expressions:

```
2 + 2  /* Its value is "4"          */

"A" "B" "C"
       /* Its value is "A B C"      */

5 < 7  /* Its value is "1", because */
       /*  the comparison is true   */
```

In this chapter we discuss how to write expressions that the language processor can compute. The rules that the language processor uses for evaluating an expression (that is, finding its value) will be explained. The chapter is divided into sections, namely:

- Operators
- True and False
- Functions
- Loops (see note below)
- Arithmetic
- Groups (see note below)
- Text
- Comparisons
- Conversion and translation.

Each section has its own introduction describing what is in it and advising you what to leave until Reading 2 or Reading 3.

Note: This chapter includes brief discussions on “Loops” and “Groups of Instructions”. These topics are included here so that you will be able to understand some of the examples given later in this chapter. There are further discussions on both topics later in the book (“Loops” on page 138, Groups of Instructions in “Selection” on page 129).

Operators

In this section:

Reading 1 immediately following, describes:

- Operators and Terms
- Order of evaluation
- Parentheses.

Reading 2 on page 37, describes:

- Using the TRACE instruction to see how expressions are being evaluated
- Data types
- Prefix operators
- Priority of operators
- Using parentheses.

Reading 3 skips this section.

- Continue **Reading 3** in “Functions” on page 45.

Operators and Terms

Reading 1

An expression can include *operators* that operate on the adjacent *terms*. Here are some operators:

+	Add
*****	Multiply
II	Concatenate (join together).

In this example, the operators act on the terms 4 and 3.

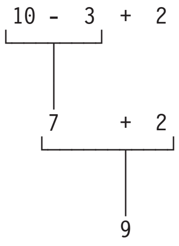
```
say 4 + 3                     /* says "7"             */
say 4 * 3                      /* says "12"            */
say 4 || 3                     /* says "43"            */
```

The *terms* that the operators work on can be numbers, strings in quotation marks, variables, the results obtained from a function call, or the result that has been obtained by evaluating the expression so far.

Order of Evaluation

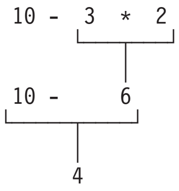
Expressions are usually evaluated from left to right.

For example,



In other words, the value of: 10 - 3 + 2 is: 9.

But some operations are given priority over others. The rules are generally the same as in ordinary algebra. For example, multiply (*) has a higher priority than subtract (-).



In other words, the value of: 10 - 3 * 2 is: 4.

We shall discuss the rules of priority again in Reading 2 on page 40.

Parentheses

When the language processor finds an expression in parentheses, it evaluates the value of the expression inside the parentheses first.

For example:
  The value of 10 * (3 + 4) is: 70
  The value of 10 * (3 || 4) is: 340.

Note, however, that if there is a symbol or a string immediately to the left of the left parenthesis, this denotes a *function*. This concept is discussed later in “Functions” on page 45.

Test Yourself...

You probably remember that if the name of a variable is found in an expression, the value of that variable will be substituted for its name.

For example:

```
/* After the instructions */
something = "mice"
a = 7
say "Cats chase" something    /* says "Cats chase mice" */
say a + 3                     /* says "10"              */
```

1. What will this program display on the screen?

```
/* PERSONS EXEC */

/* Example: simple arithmetic using variables */
pa = 1
ma = 1
kids = 3
say "There are" pa+ma+kids "people in this family"
```

2. What will this program display on the screen?

```
/* COUNTING EXEC */

/* Example: simple arithmetic using variables */
thumbs = 1
fingers = 4
hands = 2
say "It's easy to count up to",
     hands * (thumbs + fingers)
```

Answers:

1. `There are 5 people in this family`
2. `It's easy to count up to 10`

Reading 1 continues in “True and False” on page 41.

Tracing

Reading 2

To find out how the language processor will evaluate an expression, use the TRACE instruction. Some useful forms of this instruction are:

TRACE Intermediates
As each expression is evaluated, the result of each operation (that is, Intermediate results) is displayed on the screen.

TRACE Results
When each expression has been evaluated, the final result is displayed on the screen.

TRACE Normal
Only commands that are rejected by the environment are displayed on the screen.

When a TRACE instruction is being interpreted, the first letter of the second word determines what type of tracing will be switched on, and the rest of the word is ignored.

For example, to trace intermediate results for an expression, you could write:

```
TRACE I
... expression
TRACE N
```

Here is a practical example:

```
/* TTRACE EXEC                                           */
/* Example: to show how an expression is evaluated,     */
/* operation by operation                                */
x = 9
y = 2
trace I
if x + 1 > 5 * y
then say "x is big enough"
trace N
```

Figure 21. TTRACE EXEC

This would cause the following to be displayed on your screen:

```
ttrace
     6 *-* if x + 1 > 5 * y
       >V>   "9"
       >L>   "1"
       >O>   "10"
       >L>   "5"
       >V>   "2"
       >O>   "10"
       >O>   "0"
     8 *-* trace N
Ready;
```

where:

- This is the instruction being traced. The number on the left is the line number in your program.

>V> Value of a Variable.

>L> Value of a Literal.

>O> Result of an Operation.

For Figure 21 on page 38, you can see that the final result is `0` (false). And because the IF expression is false, the THEN clause is not executed.

To display only the final results use TRACE Results:

```
TRACE R
...
TRACE N
```

For example:

```
/* RTRACE EXEC                                          */
/* Example: to show how an expression is evaluated,     */
/* operation by operation using TRACE R                 */
x = 9
y = 2
trace R
if x + 1 > 5 * y
then say "x is big enough"
trace N
```

Figure 22. RTRACE EXEC

When used in the same program, this would give:

```
ttrace
     6 *-* if x + 1 > 5 * y
       >>>   "0"
     8 *-* trace N
Ready;
```

where:

>>> This is the final result.

Again, you can see that the final result is `0` (false). And because the IF expression is false, the THEN clause is not executed.

Here is a suggested order for tracing your programs that will make it easier for you to find errors:

1. TRACE SCAN—shows unmatched DO/ENDs, quotation marks, missing commas, and so on.
2. TRACE !RESULTS—(use only if there are host commands)—separates host command errors from REXX instruction errors.
3. TRACE RESULTS—checks host and REXX commands.
4. TRACE INTERMEDIATES—looks at each step.

Data Types

The values of REXX variables and expressions are always character strings.

So it is possible to write, for example:

```
dollars = 5
cents = 95
...
if cents < 10 then price = dollars".0"cents
else price = dollars"."cents
say "Price =" price             /* says "Price = 5.95"  */
```

A string of digits is like any other character string but, when an arithmetical operation is performed on a string, the result is rounded. (The default is to round to nine significant digits.)

```
/* DICEY EXEC                                         */
/* Example: an arithmetical operation on a string of    */
/* digits results in a number (rounded if necessary)    */
dicey = 123456.123456    /* Assigns the 13-character    */
                         /* string to DICEY             */
say dicey                /* Says "123456.123456"        */
say dicey + 0            /* The expression is evaluated */
                         /* with an accuracy of 9       */
                         /* significant digits (The     */
                         /* default).  The result is    */
                         /* "123456.123"; and this is   */
                         /* what is displayed.          */
```

Figure 23. DICEY EXEC

Prefix Operators

Most operators work on the terms of the expression on both sides of the operator. If you omit either term, an error occurs. However, three operators work only on the term that follows them:

+	Take (a number) as is
–	Negate (a number)
\ ¬	Logical NOT; negates, 1 becomes 0 and 0 becomes 1.

These three operators are called *prefix operators*. (Notice that the characters "+" and "–" can represent both ordinary operators and prefix operators.)

Priority of Operators

When evaluating an expression, the language processor usually works from left to right. But some operators are given a higher priority than others.

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

\ ¬ – +	(prefix operators)
**	(exponentiation)
*** / % //**	(multiply and divide)
+ –	(add and subtract)
" " || abuttal	(concatenation, with/without blank)
== = \\== ¬== **/== \\= ¬= /=** **> < >> << ><** **<> >= \\< ¬<** **>>= \\<< ¬<<** **<= \\> ¬> <<=** **\\>> ¬>>**	(comparison operators)
&	(and)

| **&&** (or, exclusive or).

For any expression, you can discover the sequence that will be used from the preceding list of priorities. For example:

```
Say 3 + 2*5                /* says "13"              */
```

Because multiply (*) has a higher priority than add (+), the multiply operation is done before the operation on its left.

Similarly, because add (+) has a higher priority than concatenate (blank),

```
Say 3 2+2 5                /* says "3 4 5"           */
```

For full details see the *z/VM: REXX/VM Reference*.

Using Parentheses

You can use parentheses to force evaluation in a different order, because expressions inside parentheses are evaluated first. For example:

The value of 6 – 4 + 1 is 3.
The value of 6 –(4 + 1) is 1.
The value of 3 + 2||2 + 3 is 55.
The value of 3 +(2||2)+ 3 is 28.

For full details on the use and priority of operators, see the *z/VM: REXX/VM Reference*.

Test Yourself...

What is the value of:

1. 4 + 20 “tailors”
2. 24 = 4 + 20
3. “eggs” = “eggs” & 2*2 = 4
4. 3 / 2*5
5. 3 || 7+7
6. 3(2+2)
7. (2+2)3.

Answers:

1. 24 tailors (add before concatenate)
2. 1 (add before comparison)
3. 1 (comparison before AND, multiply before AND, comparison before AND)
4. 7.5 (operators that have the same priority are processed left to right)
5. 314 (add before concatenate)
6. calls the function `3` with the argument `4` (or gives a syntax error if `3` does not exist)
7. 43 (evaluate expression in parentheses first; then abut).

Reading 2 continues in “True and False.”

True and False

In this section:

Reading 1 immediately following, describes:

- Comparisons
- Using TRUE and FALSE
- The Equal Sign
- The AND operator
- The OR operator.

Reading 2 on page 44, describes:

- The logical operators: NOT, AND and OR.

Comparisons

Reading 1

Comparisons are performed using the operators

>	Greater than
=	Equal
<	Less than.

These operators can be combined with each other and with the not character (\ or ¬). The result of these comparisons is either TRUE or FALSE. For more information see the *z/VM: REXX/VM Reference*.

Using True and False

If the expression is:

TRUE, the computed result is `1`
FALSE, the computed result is `0`.

For example:

```
say 4 < 7                /* says "1", meaning TRUE     */
say "Chalk" = "Cheese"   /* says "0", meaning FALSE    */
```

Instructions like:

```
IF expression THEN ... ...
```

must be given an expression that computes to `0` or `1`.

The following two fragments will give the same result.

```
ready = "YES"
...
if ready = "YES" then ...
...
```

or

```
ready = 1
...
if ready then ...
...
```

You can use whichever form you prefer.

The Equal Sign (=)

Notice that the equal sign (=) can have two meanings in REXX depending on its position in a clause.

For example:

```
amount = 5      /* The variable AMOUNT gets the value 5 */
say amount = 5  /* Compare the value of AMOUNT with 5   */
                /* If they are the same, says "1"       */
                /* Otherwise, says "0"                  */
```

The rule is, a clause beginning

```
symbol = ...
```

is an *assignment*. An equal sign appearing anywhere else in a clause stands for the comparison operator. (In a comment or a string, the equal sign is simply a character; it is not an operator.)

The AND (&) Operator

To write an expression that is only true when every one of a set of comparisons is true, use the AND (&) operator:

```
If ready = "YES" & steady = "RIGHT"
then say "GO"
```

This means “If `READY` has a value of YES **and** `STEADY` has a value of RIGHT, then say GO. Otherwise, do nothing”.

The OR (|) Operator

To write an expression that is true when any one of a set of comparisons is true, use the inclusive OR (|) operator:

```
If ready = "YES" | steady = "RIGHT"
then say "GO"
```

This means “If either `READY` has a value of YES **or** `STEADY` has a value of RIGHT, then say GO. Otherwise, do nothing”.

Test Yourself...

1. What appears on the screen when the following program is run?

```
/* FAIR EXEC         */

/* A fair comparison */
say "Apples" = "Apples"
```

2. What appears on the screen when the following program is run?

```
/* MEASURES EXEC              */

/* Example: comparing numbers */
dozen = 12
score = 20
say score = dozen + 8
/* Using the AND operator     */
say dozen = 12 & score = 21
```

Answers:

1. What is displayed is:

```
fair
1
Ready;
```

This is because `Apples` is equal to `Apples`, so the result is `1` (true).

2. What is displayed is:

```
measures
1
0
Ready;
```

The last line of output may need some explanation. The first comparison (`dozen = 12`) gives `1` (true); but the second comparison (`score = 21`) gives `0` (false). So the result is `0` (false).

Remember, the AND operation gives a result of `1` (true) *only* if both operands are `1`.

Reading 1 continues in “Functions” on page 45.

Logical Operators

Reading 2

The three most frequently used logical operators are:

¬	NOT
& (ampersand)	AND
| (vertical bar)	OR

(There is also an Exclusive OR operator (&&), but it is not often used.)

Logical operators can only process the values `1` or `0`.

The NOT (¬, \) Operator

The *not* operator (¬, \), is placed in front of a term and changes its value from *true* to *false* or from *false* to *true*.

```
say ¬ 0                /* says "1"               */
say ¬ 1                 /* says "0"               */
say ¬ 2                 /* gives a syntax error    */
say \ (3 = 3)           /* says "0"               */
```

The AND (&) Operator

The *and* operator (&), is placed between two terms. It gives a value of *true* only if both terms are *true*.

```
say (3 = 3) & (5 = 5)    /* says "1"               */
say (3 = 4) & (5 = 5)    /* says "0"               */
say (3 = 3) & (4 = 5)    /* says "0"               */
say (3 = 4) & (4 = 5)    /* says "0"               */
```

The OR (|) Operator

The *or* operator (|), is placed between two terms. It gives a value of *true* unless both terms are *false*.

```
say (3 = 3) | (5 = 5)    /* says "1"                */
say (3 = 4) | (5 = 5)    /* says "1"                */
say (3 = 3) | (4 = 5)    /* says "1"                */
say (3 = 4) | (4 = 5)    /* says "0"                */
```

Test Yourself...

1. Suggest suitable values for X and Y in this program fragment:
   a. `if month = "DECEMBER" & day of month = 25 then say X`
   b. `if command = "STOP" | message = "WATCH OUT" then color of flag = Y`
2. In the preceding program fragment, what happens if:
   a. `month = JUNE` but `day of month = 25`?
   b. `command = GO` but `message = WATCH OUT`?
3. Suitors may be TALL (or not), DARK (or not), HANDSOME (or not), and RICH (or not). A certain princess specifies:

   ```
   If TALL & DARK | HANDSOME & RICH
   then say "I will marry him"
   ```

   A certain prince has the following attributes:
   TALL—yes
   DARK—yes
   HANDSOME—no
   RICH—no.

   If he asks for her hand (and half the kingdom, of course) what will she say? You may need to review "Priority of Operators" on page 40.

Answers:

1. The answers are:
   a. `X` could be `Merry Christmas`.
   b. `Y` could be `RED`.
2. If so,
   a. Nothing is said
   b. `COLOR OF FLAG` is set to the value of `Y`.
3. `I will marry him`

   The AND operator (&) has priority over the OR operator (|). In other words, REXX computes the expression as

   ```
   (TALL & DARK) | (HANDSOME & RICH)
   ```

Reading 2 continues in "Functions."

Functions

A *function call* can be written anywhere in an expression. It performs the computation named by the function and returns a result, which is then used in the expression in place of the function call.

In this section:

Reading 1 immediately following, describes:
- The idea of a function
- REXX built-in functions
- User-written functions.

Reading 2 on page 48, describes:
- Writing your own functions
  - The ARG instruction and the ARG function
  - The RETURN instruction.

Reading 3 on page 51, describes:

- Including your own functions in the exec file of the program that uses them
- Functions written in Assembler Language.

The Idea of a Function

Reading 1

To help explain the idea of a function, think about the fictitious function:

```
HALF(   )
```

For example:
The value of HALF(6) is 3.
The value of HALF(3+5) is 4.
The value of 7 + HALF(5-3) is 8.

(The full specification and code for the HALF( ) function will be discussed later, in Figure 25 on page 50.)

Generally, if the language processor finds

```
symbol(expression ... )
```

in an expression, with no space between the last character of the symbol and the left parenthesis, it assumes that `symbol` is the name of a function and that this is a call to the function `symbol( )`.

The value of a function call depends on what is inside the parentheses. (It is an error to leave out the right parenthesis). When the value of the function has been calculated, the result is put back into the expression in place of the function call.

For example:

```
say 7 + HALF(6)        /* becomes 7 + 3 which says "10" */
x = HALF(4 + 6) - 1     /* becomes x = 5 - 1             */
say x                   /* says "4"                      */
```

The expression inside the parentheses is called an *argument*. As you can see, an argument can itself be an expression; the language processor computes the value of this expression before passing it to the function.

If a function requires more than one argument, they must be separated by commas. For instance, to obtain the greatest of a set of numbers you can use the REXX function:

```
         ┌─,─────┐
         ▼       │
►►──MAX(───number─┴──)──────────────────────────────────────────►◄
```

For example:
The value of MAX(2,3,7,4) is 7.
The value of MAX(-9,3+4,5) is 7.

Remember that a function call, like any other expression, does not usually appear in a clause by itself.

```
x = 12
y = half(x)                /* makes y equal to half(x) */
half(x)                    /* calls "6 EXEC" if it     */
```

```
                            /* exists!                   */
                            /* See page Chapter 6, "Commands," on page 99. */
x = half(x)                 /* halves x                  */
```

Built-in Functions

Over 50 functions (like the MAX( ) function, shown previously), are *built-in* to REXX. In this book, they will be introduced where you are most likely to want to use them. For example, arithmetical functions like FORMAT( ) and TRUNC( ) appear in the section on arithmetic. You will find a dictionary of built-in functions in your *z/VM: REXX/VM Reference*. From now on, if we refer to a function without saying where to find it, assume that it is a REXX built-in function.

User-Written Functions

You can also write your own functions. And you can use functions written by other people in your organization.

If a function is in the same file as the program that uses it, it is called an *internal* function. If it is in a separate file it is called an *external* function. Later, we shall see that HALF( ) is an external function.

Test Yourself...

1. What is the value of:
   a. `HALF( HALF(26)  +  HALF(6)  )`
   b. `MAX( 3, HALF(8) )`
   c. `HALF(100)`
   d. `HALF (100)`
2. The RANDOM( ) function can be used for games and for statistical models. For example, to obtain a number, chosen at random from the range 1 through 6, you could write:

   ```
   random(1,6)
   ```

   Write a program called TOSS that will display either the word `Heads` or (just as likely) the word `Tails`. Run your program a number of times. Are the results like those you could obtain by tossing a coin?

Answers:

1. If used as an expression (for example, as part of a SAY instruction) the result would be:
   a. 8
   b. 4
   c. 50
   d. HALF 100 (Not a function, because there is no name immediately to the left of the left parenthesis.)
2. A simple solution would be:

```
/* TOSS EXEC                */

/* Simulates tossing a coin */
if random(1,2) = 1
then say "Heads"
else say "Tails"
```

If you needed to make a lot of two-way decisions, you might make use of this program. The CP command

```
set pf6 immed toss
```

would let you reach a decision quickly, just by pressing the Program Function key.

Reading 1 continues in "Loops" on page 53.

Writing Your Own Functions

Reading 2

If you find you need a function that is not provided by REXX, you can easily write one of your own. You will need:

- The ARG instruction (or the PARSE ARG instruction, or the ARG( ) function) to obtain the arguments
- The RETURN instruction to return the result.

ARG Instruction

To obtain the arguments (that is, the computed values of the expression or expressions inside the parentheses of the function call), use:

```
               ┌──────────────┐
               ▼              │
►►──ARG────────────myarg──────┴──;─────────────────────────────────────►◄
```

where:

myarg are the names you choose for the variables that will be given the values of the arguments.

These values will be translated to uppercase. If you want to assign them without translating them to uppercase, use

```
                     ┌──────────────┐
                     ▼              │
►►──PARSE ARG──────────────myarg────┴──;───────────────────────────────►◄
```

The ARG( ) Function

If you do not want to give names to the arguments, you can use the function:

```
►►──ARG(──┬──────────────────┬──)──────────────────────────────────────►◄
          └─n─┬───────────┬──┘
              └─,option───┘
```

In this way you can refer to the *n*th argument.

RETURN Instruction

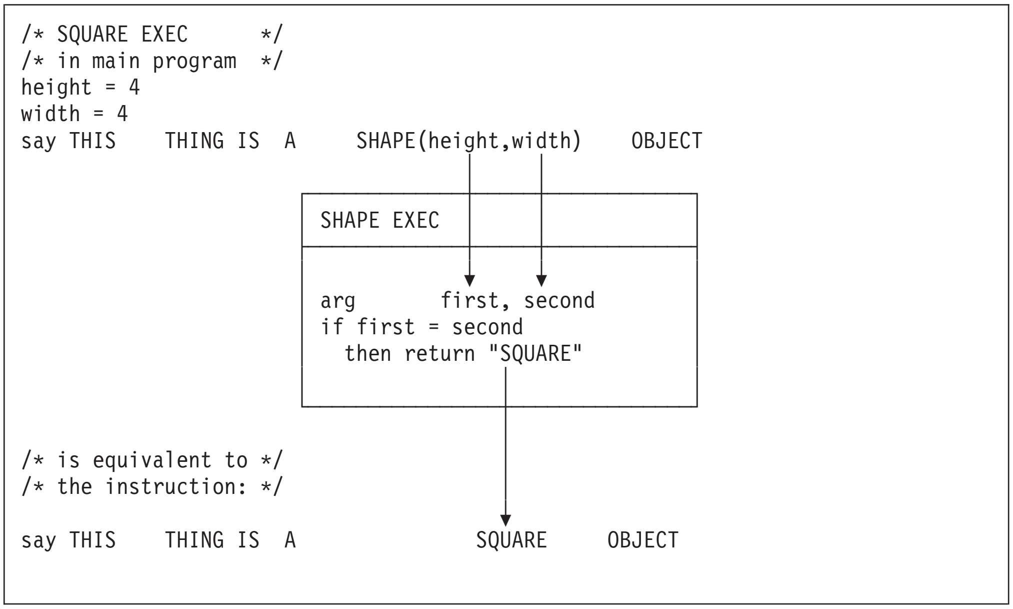
To use the result from a function call, the data must be returned from the function call to the main program. To return the result, use the following instruction:

```
►►──RETURN──┬────────────┬──;─────────────────────────────────────────────►◄
            └─expression─┘
```

The language processor computes the value of `expression` and returns the value to the main program.

A function *must* return some data.

In this next example, the expression in the main program is a string of words. One of the words is computed by a function.



```
/* SQUARE EXEC      */
/* in main program  */
height = 4
width = 4
say THIS    THING IS  A     SHAPE(height,width)    OBJECT

                         SHAPE EXEC

                         arg       first, second
                         if first = second
                           then return "SQUARE"

/* is equivalent to */
/* the instruction: */

say THIS    THING IS  A               SQUARE     OBJECT
```

Figure 24. SQUARE EXEC

The RETURN instruction *must* specify some data when returning from a function. If the RETURN instruction does not do so, you will receive a syntax error. You can intentionally leave out the data on the RETURN instruction if you want to warn the user that the input arguments, if any, are incorrect.

For example, you can write:

```
return /* error message */
```

When the function is called with incorrect arguments, the RETURN instruction, including the comment, is displayed on the screen (Error 45) followed by the line containing the function call (Error 40).

It might be wise to check that the right number of arguments has been submitted. This can be done using the ARG( ) function.

```
if arg() ¬= 1
then return  /* wrong number of arguments */
```

See the ARG( ) function in your *z/VM: REXX/VM Reference* for other ways of using this function.

Test Yourself...

Here is the specification and code for the HALF( ) function that we discussed on page 46.

```
/* HALF EXEC                                        */

/*       HALF(number)                               */
/*                                                  */
/* This function returns half of "number".  If "number" */
/* is not even, the "big half" is returned.  That is,   */
/* integer division by 2 is performed and, if there is  */
/* a remainder, it is added to the result.          */
/*                                                  */
/*    The value of   HALF(6)    is   3              */
/*    The value of   HALF(7)    is   4              */
/*                                                  */
/* If "number" is not a whole number, nothing is    */
/* returned.  This will cause a syntax error to be  */
/* raised in this program and in the calling program.   */
/*                                                  */
arg number
if datatype(number,whole)
then return number%2 + number//2
else return     /* first argument is not a whole number */
```

Figure 25. HALF EXEC

1. Use XEDIT to create a file containing the last five lines of HALF EXEC. Write an exec called TESTHALF that uses HALF and displays the result of:
   a. Half(3) Half(4) Half(5)
   b. Half(4.5)
2. Alter HALF EXEC so that it signals an error if more than one argument is supplied. Alter TESTHALF so that it contains:

   ```
   say "Testing" HALF(5,7)
   ```

   Write an exec that will give you a simple set of error messages.

Answers:

1. A possible answer is:

```
/* TESTHALF EXEC            */

/* Test cases for HALF EXEC */
say "Case 1(a)"
say half(3) half(4) half(5)
say
say "Case 1(b)"
say half(4.5)
```

When run, the TESTHALF EXEC gives the result:

```
testhalf
Case 1(a)
2 2 3
Case 1(b)
    18 +++  return     /* first argument is not a whole number */
DMSREX480E Error 45 running HALF EXEC, line 18: No data specified on function RE
```

```
TURN
     6 +++ say half(4.5)
DMSREX475E Error 40 running TESTHALF EXEC, line 6: Incorrect call to routine
Ready(20040);
```

2. A possible answer is:

```
/* TESTHAL2 EXEC */

/* Test case for modified HALF EXEC (See Question 2) */
say "Testing" half2(5,7)
```

The TESTHAL2 EXEC calls a modified version of HALF EXEC, named HALF2 EXEC.

```
/* HALF2 EXEC */

/*                                                   */
if arg() ¬= 1
then return            /* wrong number of arguments */
arg number
if datatype(number,whole)
then return number%2 + number//2
else return /* first argument is not a whole number */
```

When run, the HALF2 EXEC results in:

```
testhal2
    3 +++  return /* wrong number of arguments */
DMSREX480E Error 45 running HALF2 EXEC, line 3: No data specified on function RE
TURN
     2 +++ say "Testing" half2(5,7)
DMSREX475E Error 40 running TESTHAL2 EXEC, line 2: Incorrect call to routine
Ready(20040);
```

Reading 2 continues in “Arithmetic” on page 56.

A Square Root Function

Reading 3

This is an example of a function that you could code for yourself.

```
/* SQRT EXEC                                                */
/* The SQUARE ROOT function.                                   */
/*                                                             */
/* A function to calculate the square root of a number         */
/* using the Newton-Raphson method.                            */
/*                                                             */
/*                SQRT(number)                                 */
/*                                                             */
/* where "number" is a nonnegative REXX number,                */
/* returns the square root of "number".  If the number  */
/* is negative or not a decimal number, then this function will */
/* return a null character and report the error.                */
arg num                             /* get the number          */
null = ''

if ¬datatype(num,'Number')          /* valid number?           */
  then do
    say 'Invalid input argument:' Num'.  Must be a positive decimal number.'
    return null
  end

if num < 0                          /* check for negative      */
  then do
    say 'Invalid input argument:' Num'.  Must be a positive decimal number.'
    return null
  end
  else if num = 0 then
    return 0                        /* check for 0             */

xnew = num                          /* initialize answer       */

                                    /* calculate maximum       */
eps = 0.5 * 10**(1+fuzz()-digits()) /* accuracy                */

/* Loop until a sufficiently accurate answer is obtained.      */

do until abs(xold-xnew) < (eps*xnew)
  xold = xnew                       /* save the old value      */
  xnew = 0.5 * (xold + num / xold)  /* calculate the new       */
end

xnew = xnew / 1                     /* strip unnecessary zeros */

return xnew
```

Figure 26. SQRT EXEC

Internal Functions

Instead of writing a function as a separate file, you may prefer to include it in your main program. If the function is called many times by your main program, there will be a perceptible improvement in performance.

Begin your function with a label. To avoid problems with duplicate names, use the PROCEDURE instruction (see “The PROCEDURE Instruction” on page 30).

```
/* ROOTS EXEC                                          */
/* This program tabulates the square roots of the      */
/* whole numbers in the range 1 to 100.                */
/*                                                     */
/* The output is stored in the file ROOTS TABLE A.     */
/* The previous version of that file, if any, is       */
/* overwritten.                                        */
"ERASE ROOTS TABLE A"
do j = 1 to 100 until rc ¬= 0
   "EXECIO 1 DISKW ROOTS TABLE A (STRING",
                        format(j,3,0) format(sqrt(j),3,8)
end
if rc ¬= 0
then say "Unexpected return code" rc,
         "from EXECIO 1 DISKW command in ROOTS EXEC"
exit
/*-----------------------------------------------------*/
/* square root function                                */
/*-----------------------------------------------------*/
SQRT: procedure
...
           /* From here on, the code  */
           /* is the same as that shown in */
           /* SQRT EXEC on page 52. */
```

Figure 27. ROOTS EXEC

Functions Written in Assembler Language

A further improvement in performance can be obtained by writing your function in assembler language. However, this is only likely to be worthwhile for a function used very frequently, and by many programs.

Consult your System Support specialist or *z/VM: REXX/VM Reference* for more information.

Reading 3 continues in “Arithmetic” on page 56.

Loops

Reading 1

This whole section, “Loops” is covered in Reading 1.

A *loop* is a part of a program in which the same sequence of instructions are executed repeatedly. This is a good point to interrupt our discussion on expressions and take a look at one or two things about loops:

- How to write a loop that keeps asking for input until a valid answer is keyed in
- How to stop a program that is in an endless loop.

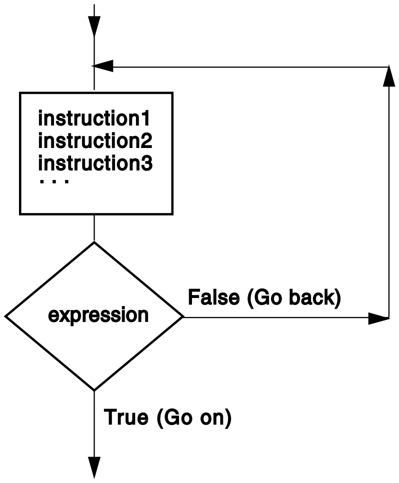
The DO Instruction

To build loops, you should use the REXX instruction DO. This is described fully in a later section, “Loops” on page 138.

A DO UNTIL Loop

There is one particular kind of loop that we shall need to use in our examples in the next two sections. It is the one where, when all the instructions inside the loop have been executed, a decision is made either to go on or to go back and repeat the instruction again.

The diagram shows why this is called a loop. The diamond represents a decision about which way to go.



In REXX programs, this should be written:

```
DO UNTIL expression
   instruction1
   instruction2
   instruction3
    ...                /* and so on */
END
```

where:

`expression` is any expression that evaluates to give `1` (true) or `0` (false). The value of `expression` is computed every time the language processor reaches the keyword `END`; if the result is `0`, the language processor loops back to `instruction1`. Otherwise, execution continues with the instruction following the `END` instruction.

For example, the program in Figure 28 will go on asking the same question until the user answers `12`.

```
/* DOZEN EXEC       */
/* Just testing you */
DO UNTIL answer = 12
   say "What is three times four?"
   pull answer
END
```

Figure 28. DOZEN EXEC

Getting Out of Loops

This program will never finish.

```
/* NEVER EXEC              */
/* This program never ends */
DO UNTIL moon = blue
   say "We are still waiting"
   moon = silver
END
```

Figure 29. NEVER EXEC

You can recognize this situation because, when you type in another command, CMS does not run it. If by any chance you find that you are running such a program and your screen fills with "We are still waiting", enter the CMS immediate command to halt interpretation:

```
HI
```

Sooner or later, you will return to CMS.

On the other hand, the program in Figure 30 is nearly impossible to get out of if you do not know what the answer is.

```
/* ABRACADA EXEC              */
/* Guess the secret password! */
DO UNTIL answer = "I QUIT"
   say "What is your answer"
   pull answer
END
```

Figure 30. ABRACADA EXEC

You can recognize this situation because, whatever you do, the words `VM READ` continue to appear in the bottom right hand corner of your screen. And typing in `HI` is no good. It just gets compared with `I QUIT`.

If you do not know the answer, the simplest way out is to enter CP mode and re-IPL CMS. Enter:

```
#cp i cms
```

This will cause CP to take over and issue an IPL CMS command.

Test Yourself...

1. Write a program called WHATDAY EXEC that keeps on asking what day of the week it is. Your program should finish as soon as the user gives the right answer. You can use the function DATE(WEEKDAY) to find out what the date really is.
2. Write a program called TESTS EXEC that keeps on asking simple arithmetical questions until the user has given five correct answers. You can use the RANDOM() function to generate some numbers at random, and ask the user to add them together.

   For example:

   ```
   RANDOM(1,9)
   ```

   Gives a whole number in the range 1 through 9.

Answers:

1. A possible answer is:

```
/* WHATDAY EXEC */

/* Example: to make the user say what day of the    */
/* week it is today.                                */
do until reply = date(weekday)
   say "What day of the week is it?"
   say "(The first letter of your response should be in"
   say "uppercase, the rest of the word should be in"
   say "lowercase.)"
   parse pull reply
   if reply ¬= date(weekday)
   then say "No, it is" date(weekday)
end
say "Correct!"
```

Figure 31. WHATDAY EXEC

2. A possible answer is:

```
/* TESTS EXEC        */

/* Arithmetical test */
credits = 0
do until credits = 5
   a = random(1,9)     /* Choose a whole number     */
                       /* between 1 and 9.  Choose  */
                       /* at random.                */
   b = random(1,9)
   say "What is" a "+" b "?"
   pull answer
   if answer = a + b
   then credits = credits + 1
   else say a "+" b "is" a+b
end
```

That is enough about loops for now. Let us return to the subject of expressions by discussing Arithmetic.

Reading 1 continues in “Arithmetic.”

Arithmetic

In this section:

Reading 1 immediately following, describes:
- Numbers
- Checking your input
- Addition, subtraction, multiplication
- Division
- Range of numbers allowed
- Exponential notation.

Reading 2 on page 61, describes:
- Formatting numeric output
- Specifying conventional and exponential notation.

Reading 3 on page 64, describes:

- Using the ** operator to compute the *n*th power of a number
- Using the NUMERIC DIGITS instruction
- Using the SIGN( ) function
- Rounding and truncation.

Numbers

Reading 1

We begin this section with some examples of numbers:

12 This is a *whole number* or *integer*.

0.5 This is a *decimal fraction* or *decimal* (one half).

3.5E6 This is a *floating point* number (three and a half million). It uses *exponential notation*. The portion that follows the E says how many places the decimal point must be moved to the right to make it into an ordinary number.

This notation is useful when dealing with very large or very small numbers.

–5 This is a *signed* number (minus five).

Checking Your Input

Before attempting to do arithmetic on data entered from the keyboard, you should check that the data is valid. You can do this using the DATATYPE( ) function.

In its simplest form, this function returns the word, NUM, if the argument (the expression inside the parentheses) would be accepted by the language processor as a number that could be used in arithmetical operations. Otherwise, it returns the word, CHAR.

The value of datatype(49) is NUM.
The value of datatype(5.5) is NUM.
The value of datatype(5.5.5) is CHAR.
The value of datatype(5,000) is CHAR.
The value of datatype(5 4 3 2) is CHAR.

So, if you want the user to keep trying until entering a valid number you could write:

```
/* VALNUM EXEC                       */
/* Example requiring numeric input */
do until datatype(howmuch) = "NUM"
   say "Enter a number"
   pull howmuch
   if datatype(howmuch) = "CHAR"
   then say "That was not a number.  Try again!"
end
say "The number you entered was" howmuch
```

Figure 32. VALNUM EXEC

If you were interested only in whole numbers you could use the alternative form of the DATATYPE( ) function. This form requires two arguments:

1. The data to be tested
2. The type of data to be tested for, for example, a whole number.

   Only the first character is inspected. Thus, to test for whole numbers it would be sufficient to write *W* or *w*. But in this book we shall write *whole* to remind you of the meaning of this argument.

This form of the function:

```
DATATYPE(number,"whole")
```

returns `1` (true) if `number` is a whole number, `0` (false) otherwise.

For example:

```
do until datatype(howmany,"whole")
    ...
   pull howmany
    ...
end
```

And if you also wanted to restrict the input to numbers greater than zero you could write:

```
do until datatype(howmany,"whole") & howmany > 0
    ...
   pull howmany
    ...
end
```

(The & is the AND operator. See "The AND (&) Operator" on page 43.)

By the way, the DATATYPE( ) function can test for other types of data, as well. See the DATATYPE function in your *z/VM: REXX/VM Reference* for further details.

Addition, Subtraction, Multiplication

These operations are performed in the usual way. You can use both whole numbers and decimal fractions.

Operand	Operation	Example
+ (plus sign)	Add	`Say 7 + 2   /* says "9"  */`
– (minus sign)	Subtract	`Say 7 - 2   /* says "5"  */`
* (asterisk)	Multiply	`Say .7 * .2   /* says ".14" */`

Division

When it comes to division, you can say whether or not you want the answer expressed as a whole number (integer). The operators you can use are:

% (percent sign)
Integer divide. The result will be a whole number. Any remainder is ignored.

For example:

```
Say 7 % 2   /* says "3"   */
```

// (two slashes)
Remainder after integer division.

For example:

```
Say 7 // 2  /* says "1"   */
```

/ (one slash)
Divide.

For example:

```
Say 7 / 2   /* says "3.5" */
```

Notice which of these operators is used here:

```
/* SHARE EXEC                                           */
/* This program works out how to share zero or more     */
/* sweets between one or more children, assuming that   */
/* a single sweet cannot be split.                      */
/*------------------------------------------------------*/
/* Get input from user                                  */
/*------------------------------------------------------*/
do until datatype(sweets,"whole") & sweets >= 0
   say "How many sweets"
   pull sweets
end
do until datatype(children,"whole") & children > 0
   say "How many children"
   pull children
end
/*------------------------------------------------------*/
/* Compute result                                       */
/*------------------------------------------------------*/
say "Each child will get" sweets%children  "sweets",
    "and there will be"   sweets//children  "left over."
```

Figure 33. SHARE EXEC

You should be careful not to divide by zero. If you do, a syntax error will result. That is why in Figure 33 the user was not allowed to answer `0` to the question “How many children.”

Because apples and oranges can be cut into pieces, you can use the other kind of division operator.

```
children = 5;  apples = 7;
say "Each child gets" apples/children "apples."
/* says "Each child gets 1.4 apples." */
```

Fractions are usually computed with an accuracy of nine significant digits:

```
children = 3;  oranges = 7;
say "Each child gets" oranges/children "oranges."
/* says "Each child gets 2.33333333 oranges." */
```

To summarize:

- The result of a `%` operation is always a whole number. There may be a remainder; to compute the remainder, write out the expression again, using the // operator.
- The result of a / operation can be a decimal.

Range of Numbers

Like a good quality hand-held calculator, the language processor works out the result correct to nine digits if necessary. This means nine significant digits, not counting the zeros that come just after the decimal point in very small decimal fractions.

```
say 1*2*3*4*5*6*7*8*9*10*11*12      /* says "479001600" */
say 7/30000000000       /* says: ".000000000233333333" */
```

The accuracy of computed results can be changed using the NUMERIC DIGITS instruction. This instruction is described in “The NUMERIC DIGITS Instruction” on page 65.

Exponential Notation

Numbers much bigger or smaller than these are difficult to read and write, because it is easy to make a mistake counting the zeros. It is simpler to use *exponential notation*. Very big numbers can be written as an ordinary (fixed point) number, followed by a letter E, followed by a whole number. The whole number says how many places to the right the decimal point of the fixed point number would have to be moved to obtain the same value as an ordinary number. So:

4.5E6 is the same as 4500000 (four and a half million).
23E6 is the same as 23000000 (twenty-three million).
1E12 is the same as 1000000000000 (a million million).

The number to the right of the E is called the *exponent*. If the exponent is negative, this means that the decimal point is to be shifted to the left, instead of to the right. So:

4.5E–3 is the same as 0.0045 (four and a half thousandths).
1E–6 is the same as 0.000001 (one millionth).

You can write numbers like this in expressions, and also when entering numeric data requested by REXX programs. The language processor will use this notation when displaying results that are too big or too small to be expressed conveniently as ordinary numbers or decimals. When the language processor uses this notation, the part of the number that comes before the E (the *mantissa*) will usually be a number between 1 and 9.99999999.

For example:

```
j = 1
do until j > 1e12
   say j                 /* says "1"                */
   j = j * 11            /*      "11"               */
end                      /*      "121"              */
                         /*      "1331"             */
                         /*      "14641"            */
                         /*      "161051"           */
                         /*      "1771561"          */
                         /*      "19487171"         */
                         /*      "214358881"        */
                         /*      "2.35794769E+9"    */
                         /*      "2.59374246E+10"   */
                         /*      "2.85311671E+11"   */
```

Numbers written in exponential notation (for example, 1.5e9) are sometimes called *floating point* numbers. Conversely, ordinary numbers (for example, 3.14) are sometimes called *fixed point* numbers.

Test Yourself...

What is displayed on the screen when this program is run?

```
/* ARITHOPS EXEC */

/* Example: arithmetical operations */
quarter = 25
deuce = 2
say quarter+deuce
say quarter-deuce
say quarter*deuce
say quarter/deuce
say quarter%deuce
say quarter//deuce
x = quarter"E"deuce
say x + 0
```

Answer:

The following is displayed:

```
arithops
27
23
50
12.5
12
1
2500
Ready;
```

The last two lines of the program require some explanation. First, `x` gets the value `25E2`. This is the same as `25.00` with the decimal point moved two places to the right (in other words, `2500`). When `x` is used in the arithmetical expression, the number `25E2` is added to zero, giving a result of `2500`.

Reading 1 continues in “Groups of Instructions” on page 67.

Formatting Numeric Output

Reading 2

Columns of figures are easier to read if the numbers are all lined up with the units in the same column. The FORMAT( ) function will help you to do this. The first three arguments are:

1. The number to be formatted
2. The number of character positions before the decimal point
3. The number of character positions after the decimal point.

Here is an example:

```
/* INVOICE EXEC                                         */
/* Example showing how columns of figures are formatted */
qty.1 = 101;     unitprice.1 = 0.73;     remark.1 = OK
qty.2 = 500;     unitprice.2 = 1995;     remark.2 = OK
qty.3 = 60000;   unitprice.3 = 70000;    remark.3 = OK
qty.4 = 500;     unitprice.4 = 400/12;   remark.4 = OK
say "Quantity    Unit Price    Total Price   Observations"
do item = 1 to 4
   say format(qty.item,5,0),
       format(unitprice.item,11,2),
       format(qty.item * unitprice.item,12,2),
       "  " remark.item
end
```

Figure 34. INVOICE EXEC

It displays the data formatted like this:

```
invoice
Quantity    Unit Price    Total Price   Observations
  101           0.73           73.73    OK
  500        1995.00       997500.00    OK
60000       70000.00            4.20E+9    OK
  500          33.33        16666.67    OK
Ready;
```

The numbers to be formatted should always be small enough to fit into the space you have reserved for them with FORMAT( ).

- A simple rule is: always specify at least 9 for the “before the decimal point” argument. If you do, numbers with more than nine digits will be displayed in Exponential Notation, and the extra characters required will cause fields to the right of the number to be shifted right, thus drawing attention to the exception.
- If you do not, the person using your program may be faced with a syntax error that is difficult to understand.

Look at item 3 in the preceding example. The quantity times the unit price (60,000 times 70,000) gives a total price of 4,200,000,000, which is too big for the nine-digit field that was specified. The result has therefore been displayed in exponential notation. This in turn has caused `OK` to be shifted right.

On the other hand, suppose we add the following:

```
qty.5 = 880000;  unitprice.5 = 1; remark.5 = "Big deal"
```

and change the 4 to a 5 in the DO instruction.

Then the display reads:

```
invoice
Quantity    Unit Price    Total Price   Observations
    101           0.73           73.73    OK
    500        1995.00       997500.00    OK
  60000       70000.00            4.20E+9    OK
    500          33.33        16666.67    OK
     12 +++  say format(qty.item,5,0),       format(unitprice.item, 11,2),
 format(qty.item * unitprice.item,12,2),       "  " remark.item
DMSREX475E Error 40 running INVOICE EXEC, line 12: Incorrect call to routine
Ready(20040);
```

This error could have been avoided:

1. In a real program, by testing the input values for a maximum number of 99999, or

2. By allowing space enough for at least nine digits for the integer part.

   ```
   say format(qty.item,9,0),
       format(unitprice.item,9,2),
       format(qty.item * unitprice.item,11,2),
       "  " remark.item
   ```

   Where the formatted data is:

   ```
   invoice
   Quantity    Unit Price    Total Price   Observations
         101         0.73          73.73    OK
         500      1995.00      997500.00    OK
       60000     70000.00           4.20E+9    OK
         500        33.33       16666.67    OK
      880000         1.00      880000.00    Big deal
   Ready;
   ```

Specifying Conventional (Fixed Point) Notation

To stop FORMAT( ) from returning floating point numbers (when results would usually be expressed in floating point numbers) use the fourth argument of FORMAT( ). This argument specifies the number of character positions reserved for the exponent. Exponential notation will not be used if you write:

```
FORMAT(number,before,after,0)
```

Be quite sure that the space you have allowed for `before` and `after` is sufficient.

Specifying Exponential (Floating Point) Notation

To make FORMAT( ) return floating point numbers (when results would usually be expressed in fixed point numbers) use the fifth argument of FORMAT( ). This argument specifies the threshold for expressing the result in exponential notation. Exponential notation will be used if you write:

```
FORMAT(number,before,after,,0)
```

For other uses of the FORMAT( ) function, see the *z/VM: REXX/VM Reference*.

A Special Case

When a floating point number has an absolute value between 1 and 9.99999999 (that is, when the **exponent** is zero) the characters `E+0` are always omitted even when floating point has been specified.

Test Yourself...

1. Write an exec called REFORMAT that expresses numbers entered by the user in both fixed point and exponential notation.
2. Test your program with the numbers:
   123456789
   0.0000000000012345
   999999999999e-6
   1.2e10
   1.2
   1.2e+0

   Or, use any other numbers you can think of.

Answers:

1. A possible answer would be:

```
/* REFORMAT EXEC */

/* Example: to change the format of a number */
do forever
   say "Enter a number"
   pull answer
   if ¬ datatype(answer,number) then exit
   say "Fixed point equivalent:" format(answer,,,0)
   say "Exponential equivalent:" format(answer,,,,0)
end
```

2. The following table lists the results you should get when using the test numbers with the REFORMAT EXEC.

Table 1. Results from the REFORMAT EXEC

Number entered:	Fixed point equivalent:	Exponential equivalent:
123456789	123456789	1.23456789E+8
0.0000000000012345	0.0000000000012345	1.2345E-12
999999999999e-6	1000000.00	1.00000000E+6
1.2e10	12000000000	1.2E+10
1.2	1.2	1.2
1.2e+0	1.2	1.2

Reading 2 continues in “Text” on page 67.

Exponentiation

Reading 3

The operator ** means “raised to the whole-number power of”. So:
2**1 = 2 = 2 (2 to the power of 1)
2**2 = 2*2 = 4 (2 to the power of 2, or 2 squared)
2**3 = 2*2*2 = 8 (2 to the power of 3, or 2 cubed)
2**4 = 2*2*2*2 = 16 (2 to the power of 4).

And, as in ordinary algebra:
2**0 = 1
2**–1 = 1/(2**1) = 0.5 (2 to the power of minus 1)
2**–2 = 1/(2**2) = 0.25 (2 to the power of minus 2).

The number on the right of the ** *must* be a whole number.

In the order of precedence, the exponentiation (**) operator comes below the *prefix operators* and above the multiply and divide operators.

For example:

```
say -5**2       /* Says "25".  Same as (-5)**2        */
say 10**3/2**2  /* Says "250".  Same as (10**3)/(2**2) */
```

The NUMERIC DIGITS Instruction

If you want to avoid using exponential notation, or simply want to increase the accuracy of your calculations, you can use the NUMERIC DIGITS instruction to change the number of significant digits. (The default setting for NUMERIC DIGITS is 9.)

For example:

```
/* ACCURATE EXEC                                     */
/* examples of numbers with unusually high precision */
numeric digits 10
say "The largest signed number that can be held"
say "in a general register is" 2**31 - 1 "exactly."
say
numeric digits 48
say "1/7 =" 1/7
```

Figure 35. ACCURATE EXEC

The sample program results in the display of:

```
accurate
The largest signed number that can be held
in a general register is 2147483647 exactly.
1/7 = 0.142857142857142857142857142857142857142857142857
Ready;
```

To check the current setting of the NUMERIC DIGITS instruction use the DIGITS( ) function. For example, if no setting was specified for NUMERIC DIGITS:

```
DIGITS()
```

would return `9` because the default setting for NUMERIC DIGITS is nine significant digits.

The SIGN( ) Function

You can determine whether a number is positive, negative, or zero by using the SIGN( ) function.

First the number inside the parentheses is rounded according to the current NUMERIC DIGITS setting. If this number is <0, =0, or >0, the value returned by the SIGN( ) function is -1, 0, 1, respectively.

For example:

```
say sign(1/7)                   /*  says "1"        */
```

Rounding and Truncation

Each arithmetical operation is carried out in such a way that no errors are introduced, except during final rounding.

For example:

```
numeric digits 3
say 100.3 + 100.3       /* gives 200.6, which is rounded */
                        /* to "201"                      */
```

For a complete description of rounding, see the *z/VM: REXX/VM Reference*.

When your program performs a series of arithmetical operations, you may inadvertently introduce additional errors. Look at the fourth item in INVOICE EXEC in Figure 34 on page 62. The customer appears to have been overcharged by $1.67! The price was $400 a dozen. FORMAT( ) has rounded this to 33.33 each. But Total Price was not rounded until after it had been multiplied by 500.

For rounding numbers, use FORMAT( ) at the point in your calculations where you want rounding to occur. For rounding down, use TRUNC( ).

```
/* TTRUNC EXEC              */
/* An example of rounding. */
qty.1 = 500;     unitprice.1 = 400/12
qty.2 = 500;     unitprice.2 = 200/12
say
say "Quantity  Unit price  Total price  Remarks"
say copies("-",58)
do item = 1 to 2
   unitprice = FORMAT(unitprice.item,9,2)
   say format(qty.item,6,0),
       format(unitprice,7,2),
       format(qty.item * unitprice,10,2),
       "   Rounding conventionally"
   unitprice = TRUNC(unitprice.item,2)
   say format(qty.item,6,0),
       format(unitprice,7,2),
       format(qty.item * unitprice,10,2),
       "   Rounding down"
end
```

Figure 36. TTRUNC EXEC

When run, the following is displayed:

```
ttrunc
Quantity  Unit price  Total price  Remarks
----------------------------------------------------------
   500      33.33      16665.00    Rounding conventionally
   500      33.33      16665.00    Rounding down
   500      16.67       8335.00    Rounding conventionally
   500      16.66       8330.00    Rounding down
Ready;
```

Test Yourself...

1. In this program:

```
/* EXPONENT EXEC */

/* Example of a negative exponent */
if 2 ** -3 = 1/(2**3) then say "True"
else say "False"
```

   a. What is displayed on the screen?
   b. Are the parentheses in this expression really necessary?

2. What value will be computed for the expression:

```
say      9 ** (1/2)
```

Answers:

1. The answers are:
   a. True

   b. No. The ** operator has a higher priority than the / operator, so the language processor would evaluate the expression in the same way if the parentheses were removed.

2. Syntax error! The ** operator must be followed by a whole number (or an expression which, when evaluated, gives a whole number).

   In mathematics, `x ** (1/2)` means “the square root of x”. There is an example of a SQRT( ) function in “A Square Root Function” on page 51.

Reading 3 continues in “Text.”

Groups of Instructions

Reading 1

This whole section, “Groups of Instructions”, is covered in Reading 1.

We are interrupting our discussion of expressions to explain how instructions can be grouped together.

Instructions can be grouped together using:

```
DO
   instruction1
   instruction2
   instruction3
   ...
END
```

If the keyword `DO` is in a clause by itself, the list of instructions is executed one time (no loop is implied).

The DO instruction and the END keyword make the whole group into a single instruction, which can be used after a THEN or ELSE keyword.

```
IF sun = shining
THEN
   DO
     say "Get up!"
     say "Get out!"
     say "Meet the sun half way!"
   END
```

In this example, if `sun = shining`, all three SAY instructions will be executed. But if `sun ¬= shining`, none of them will.

We shall be using DO in this way in the sections that follow.

Reading 1 continues in “Text.”

Text

In this section:

Reading 1 immediately following, describes:

- How to concatenate
- How to use the SUBSTR( ), LENGTH( ), COPIES( ), LEFT( ), and RIGHT( ) built-in functions for string manipulation.

Reading 2 on page 71, describes:

- How to use a subroutine to simplify tabulation
- How to search for a string of characters using the POS( ) and WORDPOS( ) functions.
- How to display lines from your own program using SOURCELINE( ).

Reading 3 on page 76, describes:

- How to use the OVERLAY( ), WORD( ), and WORDS( ) functions.

Concatenation

Reading 1

To *concatenate* two terms means to join them together to make a string. The concatenate operators are:

|| (two vertical bars)
concatenate with no blanks in between

(blank)
concatenate with one blank in between

abuttal
concatenate with no blank in between (as long as the two terms can be recognized separately).

Here are some examples:

```
say "slow"||"coach"       /* says "slowcoach"           */
say "slow"    "coach"     /* says "slow coach"          */
/* And */
adjective = "slow"
say adjective"coach"      /* says "slowcoach", This is  */
                          /* an example of an abuttal.  */
say adjective   "coach"   /* says "slow coach"          */
say "("adjective")"       /* says "(slow)"              */
```

The SUBSTR( ) Function

The value of any REXX variable is a string of characters. To select a part of a string, use the SUBSTR( ) function. SUBSTR is an abbreviation for substring. The first three arguments are:

1. The string from which a part will be taken
2. The position of the first character that is to appear in the result (Characters in a string are numbered 1,2,3, ...)
3. The length of the result.

(For a complete definition, see the *z/VM: REXX/VM Reference*.)

Here is a simple example:

```
S = "reveal"
say substr(S,2,3)         /* says "eve"                 */
say substr(S,3,4)         /* says "veal"                */
```

The LENGTH( ) Function

To find out the length of a REXX variable, use the LENGTH( ) function.

```
S = "reveal"
say length(S)             /* says "6"                  */
```

Here is an example that uses these two functions:

```
say "Enter a file name"
pull fn .                 /* The period ensures that   */
                          /* FN is assigned only one   */
                          /* word.                     */
if length(fn) > 8
then
   do                     /* A group.  See page 67.    */
      fn = substr(fn,1,8)
      say "The file name you entered was too long. ",
          fn "will be used."
   end
```

The COPIES( ) Function

To produce a number of copies of a string, use the COPIES( ) function. The arguments are:

1. The string to be copied
2. The number of copies required.

For example:

```
say COPIES("Ha ",3)!        /* says "Ha Ha Ha !"     */
```

The LEFT( ) Function

To obtain a string that is always *length* characters long, with *string* at the left hand end of it, use the LEFT( ) function.

```
LEFT(string,length)
```

If *string* is too short, the result will be padded with blanks; if *string* is too long, the extra characters will be truncated.

For example:

```
say "|"left("Long",6)"|"          /* says   "|Long  |"    */

say "|"left("Longer",6)"|"      /* says   "|Longer|"    */

say "|"left("Longest",6)"|"     /* says   "|Longes|"    */
```

The RIGHT( ) Function

The RIGHT( ) function works the same as the LEFT( ) function, except the returned string is padded or truncated on the *left*.

Arranging Your Output in Columns

You can use the LEFT( ) function to arrange your output in columns:

```
/* TABLE1 EXEC                                          */
/* Example: tabulated output                            */
c1 = 14                         /* Width of column 1     */
c2 = 20                         /* Width of column 2     */
ruler =  c1 + c2 + 16           /* Width of ruled line   */
say left("First Name",c1)Left("Last Name",c2)"Occupation"
say copies("-",ruler)
say left("Bill",c1)Left("Brewer",c2)"Innkeeper"
say left("Jan",c1)Left("Stewer",c2)"Cook"
say left("Peter",c1)Left("Gurney",c2)"Farmer"
say left("Peter",c1)Left("Davey",c2)"Laborer"
say left("Daniel",c1)Left("Whiddon",c2)"Gamekeeper"
say left("Harry",c1)Left("Hawke",c2)"Exciseman"
say left("Tom",c1)Left("Cobley",c2)"Sailor (retired)"
```

Figure 37. TABLE1 EXEC

And you can vary the tab settings by changing the values of `C1` and `C2`. The output looks like this:

```
table1
First Name    Last Name           Occupation
--------------------------------------------------
Bill          Brewer              Innkeeper
Jan           Stewer              Cook
Peter         Gurney              Farmer
Peter         Davey               Laborer
Daniel        Whiddon             Gamekeeper
Harry         Hawke               Exciseman
Tom           Cobley              Sailor (retired)
Ready;
```

Test Yourself...

Given that C = “Continent”, what is the value of:

1. C “of America”
2. C || “al”
3. C“al”
4. LENGTH(“Continent”)
5. LENGTH(C)
6. LENGTH(“C”)
7. Substr(c,1,4)substr(c,7,3)
8. Substr(c,1,2)substr(c,5,2)
9. LEFT(“Q”,8)“QUERY”
10. LEFT(“COPY”,8)“COPYFILE”

Answers:

1. Continent of America
2. Continental
3. Continental
4. 9
5. 9
6. 1
7. Content
8. Coin

```
|---+----+----+----|
```

(This scale can help you check the number of blanks in the following answers.)

9. `Q       QUERY`
10. `COPY    COPYFILE`

Reading 1 continues in “Comparisons” on page 79.

Using a Subroutine to Simplify Tabulation

Reading 2

To make your main program easier to read, leave formatting of output to a subroutine. For example, the exec in Figure 38 shows how a subroutine can be used several times in order to create a table.

For example:

```
/* TABLE2 EXEC                                          */
/* Example: a simpler way to obtain tabulated output    */
call tabout "First Name", "Last Name", "Occupation"
say copies("-",50)
call tabout "Bill", "Brewer", "Innkeeper"
call tabout "Jan", "Stewer", "Cook"
call tabout "Peter", "Gurney", "Farmer"
call tabout "Peter", "Davey", "Laborer"
call tabout "Daniel", "Whiddon", "Gamekeeper"
call tabout "Harry", "Hawke", "Exciseman"
call tabout "Tom", "Cobley", "Sailor (retired)"
exit
/*------------------------------------------------------*/
/* Subroutine to tabulate the output                    */
/* =================================                    */
/* Input format:  CALL TABOUT arg1,arg2,arg3            */
/*    (number of arguments is not checked)              */
/*                                                      */
/* Output to screen:  arg1 in Column 1                  */
/*                    arg2 in Column 15                 */
/*                    arg3 in Column 35                 */
/*------------------------------------------------------*/
TABOUT:
say left(arg(1),14),
||  left(arg(2),20),
||       arg(3)
return
```

Figure 38. TABLE2 EXEC

The output will be the same as TABLE1 on page 69.

For the CALL instructions in Figure 38, the arguments are separated by commas. In general, each argument could be an expression.

The expression, arg(1), refers to the first argument passed to the called subroutine. arg(2) refers to the second argument passed to the called subroutine, and arg(3) refers to the third argument passed to the called subroutine. For example, in the TABLE2 EXEC, the first time TABOUT is called, arg(1) is `First Name`, arg(2) is `Last Name`, and arg(3) is `Occupation`.

For example:

```
/* TABLE3 EXEC                                         */
/* Example: arguments can be expressions               */
call tabout "First Name", "Last Name", "Occupation"
say copies("-",50)
r = "(retired)"
firstname = "Tom"
nickname = "Uncle"
lastname = "Cobley"
call tabout firstname "("nickname")", lastname, "Sailor" r
exit
/*-----------------------------------------------------*/
/* Subroutine to tabulate the output                   */
 ... (See Note 1)
```

Notes:

1. Same as TABLE2 EXEC in Figure 38.

Figure 39. TABLE3 EXEC

When run, the following is displayed:

```
table3
First Name    Last Name           Occupation
--------------------------------------------------
Tom (Uncle)   Cobley              Sailor (retired)
Ready;
```

The POS( ) Function

To find the position of a string in another string, use the POS( ) function. The first two arguments are:

1. The `needle` to be found
2. The `haystack` to be searched.

For a complete definition, see the *z/VM: REXX/VM Reference*.

Here is a simple example:

```
S = "reveal"
say pos("eve",S)          /* says "2"                 */
say pos("revel",S)        /* says "0" /* not found */ */
```

Other useful functions of this type are LASTPOS( ) and COMPARE( ).

Example

The next example uses some of the functions that you have just been reading about.

```
/* VALIDFN EXEC                                         */
/* VALIDATE FILE NAME                                   */
/* This program checks that names conform to a set of   */
/* defined standards.  The names must have the form:    */
/*                                                      */
/*     namddiii                                         */
/*                                                      */
/* where "nam" stands for one of the components (INP,   */
/* PRO, or OUT); "dd" are two decimal digits; and       */
/* "iii" are the author's initials (from one to three   */
/* letters).  For example, the fifth module that        */
/* Joe Bloggs writes for the INPut component would be   */
/*                                                      */
/*     INP05JB                                          */
/*                                                      */
do until good              /* assume the name is good   */
   good = 1
   Say "Enter file name"
   pull fn .
   if length(fn) > 8 then do            /* length      */
      say "File name must not be more",
          "than 8 characters long"
      good = 0                      /* bad file name  */
   end
   componentname = left(fn,3)           /* component   */
   select
   when componentname = "INP" then nop   /* valid names */
   when componentname = "PRO" then nop
   when componentname = "OUT" then nop
   otherwise
      say "First three characters must be",
          "a valid component name"
      good = 0                      /* bad file name  */
   end
                                        /*continued ...*/
   serial = substr(fn,4,2)
   if datatype(serial,whole) & pos(".",serial) = 0
   then nop
   else do
      say "Fourth and fifth characters must be numeric"
      good = 0                       /* bad file name  */
   end
   author = substr(fn,6)                 /* author      */
   if ¬ datatype(author,upper)
   then do
      say "Sixth and remaining characters",
          "must be alphabetic"
      good = 0                       /* bad file name */
   end
   if good = 0 then say "Try again"
 end
```

Figure 40. VALIDFN EXEC

Words

In REXX, a *word* is defined as a string of characters delimited by blanks. To process words, rather than characters, use any of the following REXX functions:

DELWORD
FIND
SUBWORD
WORD

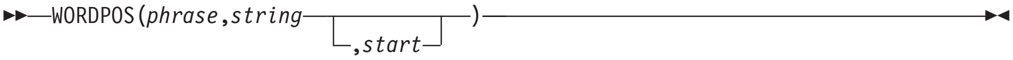
WORDINDEX
WORDLENGTH
WORDPOS
WORDS.

The following description highlights the WORDPOS function; all functions are described fully in the *z/VM: REXX/VM Reference*.

(Also see the PULL, ARG and PARSE instructions, on page The PULL Instruction on page 90 through 96).

The WORDPOS( ) Function

To find a phrase (of one or more words) in a string, use the WORDPOS( ) function.



The arguments are:

1. The *phrase* to be found.
2. The *string* be searched.
3. The *start* point of the search (must be a positive number). The default is the first word in the string.

The language processor searches *string* for the sequence of word(s), *phrase*. The result is the word-number of the first word in *string* that matches the first word in *phrase*. But, if *phrase* is not found, zero is returned.

By default the search starts at the first word in *string*. By specifying *start* you can begin the search for *phrase* on any word in *string*.

For example:

```
/* REVERE EXEC               */
/* "The British are coming!" */
text = "Listen, my children, and you shall hear",
       "Of the midnight ride of Paul      Revere"
name = "Paul Revere"
say WORDPOS(name,text)           /* says  "13"             */
say WORDPOS("my children",text)  /* says "0", because the  */
                                 /* Word in TEXT is        */
                                 /*    "children,"         */
                                 /* (Notice the comma)     */
```

Figure 41. REVERE EXEC

Providing Help

You may have noticed that CMS commands and REXX instructions are provided with a HELP command, so that if you forget how to use them you can always get a definition displayed on the screen.

If you are writing programs that other people will use, it will help *your* users if you do the same. You can either write a separate HELP file for your program or, more informally, you can provide information from within your program file.

Here is a program that provides its own HELP, using the SOURCELINE( ) function to simplify the job of displaying whole lines. SOURCELINE(n) returns the *n*th line of the source file. If n is omitted, SOURCELINE( ) returns the *line number* of the final line in the source file.

```
/* MYPROG EXEC      */
/*
This program processes the input file to give ...
 ... ...
Correct format is:
       MYPROG
Function performed is:
Rhubarb, rhubarb, rhubarb.
*/
say "Enter file ID of file to be processed"
pull fn ft fm
if fn = ? | fn = "" | ft = ""
then do
   /* Display lines until comment-end delimiter alone   */
   line = 2
   do while sourceline(line) ¬= "*/"
      say sourceline(line)
      line = line + 1
   end
   exit
end
/*-----------------------------------------------------*/
/* Main program starts here.                           */
/*-----------------------------------------------------*/
say "This is the program"
```

Figure 42. MYPROG EXEC

Note: Notice that the comment delimiters must be on a separate line in order for the exec to work properly.

Test Yourself...

Write a subroutine to display data on the screen in the following format:

- The first argument occupies columns 1 to 20. The text is left justified.
- The second argument is an amount of dollars and cents (or pounds and pence, or francs and centimes, or marks and pfennigs) with the units position of the cents in column 34.
- The third argument occupies columns 37 to 80.
- As a further refinement, extend your program so that, when the third argument is too long to fit onto one line, it can be extended into columns 37 to 80 of as many lines as necessary.

Answers:

Here is the answer to the fourth item, with some test cases.

```
/* 4MAT EXEC */

/* Example: a subroutine for formatting text, and a     */
/* main routine for testing it.                         */
call formatter "whole number", 12, "An easy case"
call formatter "expression",2000/6, "Rounded up"
call formatter "abcdefghijklmnopqrstuvwxyz",,
               12345678888,,
               "Precision of this number is that",
               "specified by NUMERIC DIGITS"
call formatter "Small number", 1/201,,
               "After rounding, this number is",
               "                 less than .005"
exit
/*------------------------------------------------------*/
/* Subroutine to format data and display it.            */
/* (For specification, see page 75.)                  */
/*------------------------------------------------------*/
FORMATTER:
len = 80 - 37 + 1                    /* length of       */
                                     /*  remark field   */
parse arg name, value, remark
do j = 1 while length(remark) > len  /* slice REMARK    */
   remark.j = substr(remark,1,len)
   remark = substr(remark,len+1)
end
remark.j = remark                    /* last slice      */
say left(name,20),                   /* say first line  */
 || format(value,11,2,0),
 || "  "remark.1
                                     /* say others      */
do line = 2 to j
   say copies(" ",36)||remark.line
end
return
```

Note: Notice the double commas in two of the CALL statements in the 4MAT EXEC. The first comma indicates that the clause is extended to the next line. The second comma indicates the end of the argument.

When this program is run, this is what is displayed:

```
4mat
whole number                 12.00  An easy case
expression                  333.33  Rounded up
abcdefghijklmnopqrst12345678900.00  Precision of this number is that specified b
                                    y NUMERIC DIGITS
Small number                  0.00  After rounding, this number is
                                        less than .005
Ready;
```

Reading 2 continues in “Comparisons” on page 79.

The OVERLAY( ) Function

Reading 3

To overlay one string onto another string, use:

```
►►──OVERLAY(new,target,position,length──)──────────────────────────────►◄
```

The arguments are:
- The string to be overlaid

- The *target* onto which it is to be overlaid
- The position in the *target* where overlaying is to start
- The number of characters to be overlaid.

For example:

```
say overlay("abc","123456",3,2)   /* says  "12ab56"     */
```

(For a complete definition, see the *z/VM: REXX/VM Reference*.)

Here is a useful example.

```
/* ORDCHARS EXEC                                         */
/* This program will help you understand how             */
/* comparisons are made.  The characters typed in by     */
/* the user will be sorted into ascending order.         */
say "Please type in all the characters you would",
    "like to have sorted."
parse pull S                       /* Do not translate  */
                                   /* to uppercase.     */
do until swap = 0
swap = 0
   do p = 1 to (length(S) - 1)
      c1 = substr(S,p,1)
      c2 = substr(S,p+1,1)
      if c1 > c2 then do           /* If out of order,  */
         S = overlay(c2||c1,S,p,2) /* swap them.        */
         swap = 1                  /* Remember the swap */
      end
   end
end
say
say "Here are the same characters",
    "arranged in ascending order:"
say
say S
```

Figure 43. ORDCHARS EXEC

This is not the fastest way of sorting things, but it is one of the simplest.

The WORDS( ) and WORD( ) Functions

A *word* is a string of characters, delimited by blanks. To obtain the number of words in a string, use the WORDS( ) function.

For example:

```
necessity = "the mother of invention."
say words(necessity)                 /* says "4"       */
```

To obtain a particular word from a string, use the WORD( ) function. The arguments are:

- The string
- The number of the word to be extracted from it.

For example:

```
necessity = "the mother of invention."
say word(necessity,2)                /* says "mother"  */
```

This next example demonstrates how the WORD and WORDS functions can be used to search for a word (in this case, a file type) that matches one of a given list of words.

```
/* XE EXEC                                              */
/* This exec helps you select files to be edited by     */
/* the XEDIT editor.  Use the command                   */
/*                                                      */
/*       XE filename [filetype [filemode]] [(options]   */
/*                                                      */
/* You need not specify a file type.  If you do not,    */
/* XE will search for a file in the following order:    */
/*                                                      */
/*       filename SCRIPT        on any filemode         */
/*       filename EXEC          on any filemode         */
/*       filename PLIOPT        on any filemode         */
/*       filename DOC           on any filemode         */
/*       filename LISTING       on any filemode         */
/*                                                      */
/* If none of these can be found, it will select        */
/*                                                      */
/*       filename SCRIPT        A                       */
/*                                                      */
/* However, if you do specify a file type, XEDIT will   */
/* use the file type that you have specified on the     */
/* command line.                                        */
/*                                                      */
/* When the file has been chosen, XEDIT will be called  */
/* and any options that you have specified on the       */
/* XE command line will be passed to XEDIT              */
                                    /* continued ...    */
```

Figure 44. XE EXEC (Part 1 of 2)

```
types = "SCRIPT EXEC PLIOPT DOC LISTING"
/*------------------------------------------------------*/
/* check arguments                                      */
/*------------------------------------------------------*/
arg filename filetype filemode "(" options
      /* Coding note:     */
      /* See 56.       */
if filename = "" | filename = "?"   /* Help needed      */
then do
   do line = 1 while substr(sourceline(line),1,2) = "/*"
      say sourceline(line)
   end
   exit
end
/*------------------------------------------------------*/
/* compute file type                                    */
/*------------------------------------------------------*/
if filetype = "" then do
   do p = 1 to words(types)
      filetype = word(types,p)
      "SET CMSTYPE HT"
      "STATE" filename filetype    /* does file exist? */
      rcs = rc
      "SET CMSTYPE RT"
      select
         when rcs = 28 then nop     /* no               */
         when rcs = 0 then leave p  /* yes              */
                                    /* Coding note:     */
                                    /* See page 147.       */
         otherwise
         say "Unexpected return code" rcs,
             "from STATE command in XE EXEC"
         exit rcs
      end /* select */
   end p
   if rcs = 28                      /* not found yet    */
   then filetype = SCRIPT
end
/*------------------------------------------------------*/
/* call xedit                                           */
/*------------------------------------------------------*/
"XEDIT" filename filetype filemode "("options
exit rc
```

Figure 44. XE EXEC (Part 2 of 2)

Reading 3 continues in "Comparisons."

Comparisons

In this section:

Reading 1 immediately following, describes:
- Comparing numbers
- Comparing character strings.

Reading 2 on page 81, describes:
- Finding the first character that does not match
- Comparing data without regard to case
- Recognizing abbreviations.

Reading 3 on page 82, describes:
- Exact comparisons
- Fuzzy arithmetical comparisons.

General

Reading 1

Comparisons are performed using the operators:

>	Greater than
=	Equal to
<	Less than.

These characters can also be combined with each other and with the **not** character (¬). (For full details, see the *z/VM: REXX/VM Reference*.)

Numbers

If both the terms being compared are numbers, comparison is numeric, rather than character by character.

The value of 5 > 3 is 1 /* true */
The value of 2.0 = 002 is 1 /* true */
The value of 3E2 < 299 is 0 /* false */

Characters

If either of the terms is not a number, leading and trailing blanks are ignored; the shorter string is padded on the right with blanks; and then the strings are compared from left to right, character by character. If the strings are not equal, the first pair of characters that do not match determine the result.

For example, if "   Chalk" is compared with "Cheese "
A character is *less than* another character if it comes earlier in the sequence:

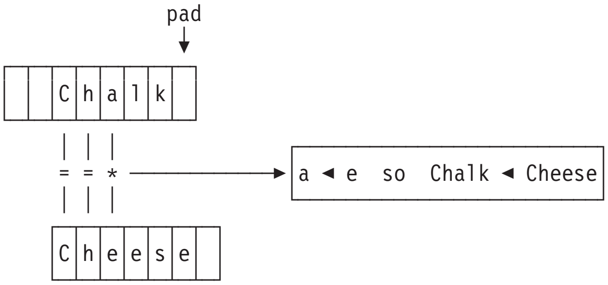


Figure 45. Comparing Character by Character

(lowest)
blank
special characters
a ... z
A ... Z
0 ... 9
(highest).

There may be exceptions to this for some of the special characters, depending on the features of the keyboard you are using. You can use the program ORDCHARS EXEC in Figure 43 on page 77 to discover the sequence of characters for your keyboard.

Test Yourself...

What is the value of each of the following expressions?

1. "3" > "five"

2. “Kilogram” > “kilogram”
3. “a” > “#”
4. “q” > “?”
5. “9a” > “9”
6. “?” > “ ”

Answers:

All are “1” (true).

Reading 1 continues in “Translation” on page 83.

The COMPARE( ) Function

Reading 2

To compare two strings and find the position of the first character in the first string that does not match the second string, use the COMPARE( ) function.

```
►►──COMPARE(string1,string2──)──────────────────────────────────────────►◄
```

For example:

```
/* Given that */
a = "Berry";  b = "Beryl";  c = " Bert"; d = "BEST"
```

The value of compare(a,b) is 4.
The value of compare(a,c) is 1.
The value of compare(a,d) is 2.

In that last example, notice that *e* is not the same as *E*. When you would like your comparisons to be independent of case, translate everything to uppercase first. Of course, if you obtained your data using ARG or PULL, this will have been done for you. If not, you can use the UPPER instruction to change one or more variables to uppercase.

```
/* Given that */
a = "Berry";  b = "Beryl";  c = " Bert"; d = "BEST"
UPPER a b c d
```

The value of compare(a,d) is 3.

The ABBREV( ) Function

In a friendly environment, the user might expect to be allowed to use abbreviations, just as you can with CMS commands. To specify what abbreviations you will accept, use the ABBREV( ) function.

```
►►──ABBREV(information,info──┬─────────┬──)─────────────────────────────►◄
                             └─,length─┘
```

The arguments are:

1. The keyword in full.
2. The user’s answer.
3. The minimum number of characters in the user’s answer. If you leave this argument out, the minimum number is assumed to be the same as the actual length of the user’s answer. A null answer is also accepted.

The result is `1` (true) if `info` (the user’s answer) is at least *length* characters long and all the characters of `info` match the corresponding characters of `information` (the keyword in full).

For example,

```
/* YEP EXEC                         */
/* Example: accepting abbreviations */
do until yes ¬= "YES"                /* until YES is set   */
   say "  ... answer Yes or No"
   pull answer
   select
      when abbrev("YES",answer,1)    /* accepts "YES",     */
                                     /* "YE" or "Y"        */
      then yes = 1
      when abbrev("NO",answer)       /* accepts "NO", "N"  */
                                     /* or ''              */
      then yes = 0
      otherwise say "Try again!"
   end                               /* select */
end
if yes then say "I take that to mean YES"
else say "I take that to mean NO"
```

Figure 46. YEP EXEC

Test Yourself...

Given that:
  Q2 = “COPY”
  Q3 = “PRT”

What is the value of:
1. COMPARE(SUBSTR(Q2,3),Q3)
2. ABBREV(“COPYFILE”,Q2,4)
3. ABBREV(“PRINT”,Q3,2).

Answers:
1. 2
2. 1
3. 0 (“PRT” is not equal to the first 3 letters of “PRINT”.)

Reading 2 continues in “Translation” on page 83.

Exact Comparisons

Reading 3

Strict comparison operators carry out simple character-by-character comparisons, with no padding of either of the strings. They do not try to perform numeric comparisons because they test for an exact match between the two strings.

To find out whether two strings are exactly equal (that is, identical) use the == operator.

Given that:

```
x = "2";  y = "+2"
The value of   x = y                           is   1 /* true  */
The value of   x \= y, x ¬= y  or  x /= y      is   0 /* false */
The value of   x == y                          is   0 /* false */
The value of   x \== y, x ¬== y  or  x /== y   is   1 /* true  */
```

You can also find out whether two strings are exactly greater than or exactly less than using the >> and << operators. (Remember, a character is *less than* another character if it comes earlier in the sequence. Refer to page 80.)

For example:

```
The value of   "cookies" >> "carrots"          is   1 /* true  */
The value of   "$10" >> "nine"                 is   0 /* false */
The value of   "steak" << "fish"               is   0 /* false */
The value of   " steak" << "steak"             is   1 /* true  */
```

In the last example, "` steak`" is strictly less than "`steak`" since the blank is lower in the sequence of characters.

The strict comparison operators would be especially useful if you were interested in leading and trailing blanks, nonsignificant zeros and so on.

For more information on exact comparison operators, see the *z/VM: REXX/VM Reference*.

Fuzzy Arithmetical Comparisons

There are times when an accurate comparison is inconvenient, for instance:

```
/* NOFUZZ EXEC                    */
/* Example: no approximation here */
say 1 + 1/3                        /* says "1.33333333"   */
say 1 + 1/3 + 1/3 + 1/3            /* says "1.99999999"   */
say 1 + 1/3 + 1/3 + 1/3 = 2        /* says "0"  (false)   */
```

Figure 47. NOFUZZ EXEC

To make comparisons less accurate than ordinary REXX arithmetic, use the NUMERIC FUZZ instruction. (For full details, see the *z/VM: REXX/VM Reference*.)

For example:

```
/* FUZZ EXEC                      */
/* Example: allowing approximation */
say 1 + 1/3 + 1/3 + 1/3 = 2        /* says "0"  (false)  */
numeric fuzz 1
say 1 + 1/3 + 1/3 + 1/3 = 2        /* says "1"  (true)   */
```

Figure 48. FUZZ EXEC

To check the current setting of the NUMERIC FUZZ instruction use the FUZZ ( ) function. For example:

```
FUZZ()
```

will return `0` by default. This means that 0 digits will be ignored during a comparison operation.

Reading 3 continues in "Translation."

Translation

In z/VM, each character or *byte* contains 8 *bits*. There are two possible values for each bit, and so there are 2**8 or 256 possible characters in the *character set*.

If you need to translate from one character set to another, or if you are dealing with output from programs that work in binary or hexadecimal, you should study this section.

In this section:

Reading 1 **skips this section.**

- Continue **Reading 1** in Chapter 5, "Conversations," on page 89.

Reading 2 on page 84, describes:

- Conversion between Character, Hexadecimal and Decimal.

Reading 3 on page 87, describes:

- Translation from one character set to another
- The VERIFY( ) function.

Hexadecimal

Reading 2

In z/VM, each character occupies 8 bits. Each bit can have one of two values, 0 or 1. For example, the character + has the value:

0100 1110 (binary)

But, because binary is difficult for humans to read, we might write it as a pair of hexadecimal digits. There are 16 possible hex digits. They are:

0 1 2 3 4 5 6 7 8 9 A B C D E F

So the hexadecimal equivalent of + is 4E.

Finally, we could also write the value of the character + as its decimal equivalent, which is 78.

The language processor will accept strings expressed in either character or hexadecimal form. Hexadecimal numbers are usually expressed with the X in front of the number like X'18'. But REXX only accepts hexadecimal numbers with the X after the number. So, to indicate that a string is expressed in hex, write the letter X after the closing quotation mark like '18'X.

The value of + is the same as the value of '4E'X.

Conversion

To convert from one form to another, you can use various built-in functions.

2 means *translate to*

C means *characters*

X means *hexadecimal*

D means *decimal*

The value of C2X(+) is 4E
The value of X2C(4E) is +
The value of C2D(+) is 78
The value of D2C(78) is +
The value of D2X(78) is 4E
The value of X2D(4E) is 78

All these functions will accept strings more than 1-byte long.

To understand the conversion functions, let's look at the inputs to and the outputs from the functions in hexadecimal. The following chart shows example hexadecimal input, the conversion function performed, and the resultant hexadecimal output. Also shown is another way to remember what the function does.

Table 2. Inputs and Outputs of Hexadecimal Functions

Input	Function	Result	What the function does
0F	C2D	F1F5	binary in, EBCDIC out (represents a decimal value)
0F	C2X	F0C6	binary in, EBCDIC out (represents a hexadecimal value)
F1F5	D2C	0F	EBCDIC representing decimal in, binary out
F1F5	D2X	C6	EBCDIC representing decimal in, EBCDIC representing hexadecimal out
F1C6	X2C	1F	EBCDIC representing hexadecimal in, binary out
F1C6	X2D	F3F1	EBCDIC representing hexadecimal in, EBCDIC representing decimal out

The inputs to C2D and C2X can be any hexadecimal value. Hexadecimal input is typically referred to as *binary* or *character* input. The hexadecimal value does not represent an EBCDIC string. Usually the input to C2D or C2X is generated by another program or a function, such as the REXX DIAG function, that returns a *binary* value.

You would use C2X or C2D to convert this *binary* value into a form that could be displayed on an EBCDIC terminal, or that could be used in other REXX instructions.

In the first function, C2D, the input is hexadecimal '0F'. C2D tells REXX to convert the input into a decimal value and then to convert that decimal value into its EBCDIC representation. Hexadecimal '0F' has a decimal value of 15. The EBCDIC representation of 15 is 'F1F5'. If you were to display hexadecimal 'F1F5' on an EBCDIC terminal, what you would see is the character string `15`.

Try executing:

```
say c2d('0F'x)
```

You should see a `15` displayed on your terminal. Notice that we use the notation '0F'x for input. This is because there is not a key on most EBCDIC terminals that causes a hexadecimal '0F' to be generated.

For the C2X function, the input is, again, hexadecimal '0F'. C2X tells REXX to convert the hexadecimal value into an EBCDIC form. The hexadecimal value is '0F'. The EBCDIC representation of that value is 'F0C6'. If you were to display hexadecimal 'F0C6' on an EBCDIC terminal, you would see the character string `0F`. Try executing:

```
say c2x('0F'x)
```

You should see `0F` on your terminal.

The input to the next two functions, D2C and D2X must be the EBCDIC representation of a decimal value. The output of D2C is *binary*, and hence may be nondisplayable, while the output of D2X is an EBCDIC representation of a hexadecimal value.

In the preceding chart, the input to D2C is hexadecimal 'F1F5'. By definition, the input to the D2C function is an EBCDIC string that represents some decimal value. D2C tells REXX to take the decimal value represented by the input and convert it to a hexadecimal value. The EBCDIC string 'F1F5' represents a decimal value of 15. Hexadecimal notation for decimal 15 is '0F'. Try executing both of these instructions:

```
say d2c('f1f5'x)
say d2c(15)
```

They both mean the same thing. In the first instruction, we supply the hexadecimal string as input. In the second, we type the characters, which are internally represented as hexadecimal 'F1F5'.

Both instructions attempt to display hexadecimal '0F' on your terminal. On most EBCDIC terminals, '0F' does not mean anything. You will either see a blank or, on some models, you might see an unusual character.

In the chart on page 85, hexadecimal 'F1F5' is also the input to D2X. Again, by definition, the input to D2X must be an EBCDIC string that represents some decimal value. D2X tells REXX to convert the EBCDIC representation of the decimal value into the EBCDIC representation of its equivalent hexadecimal value. EBCDIC 'F1F5' represents a decimal value of 15, which is the hexadecimal value F. The EBCDIC representation of the character F is 'C6'. Try:

```
say d2x('f1f5'x)
say d2x(15)
```

Again, the instructions mean the same thing. Both attempt to display hexadecimal 'C6' on your terminal. In EBCDIC, 'C6' represents the character F, which is what you will see on your terminal.

The last two functions, X2C and X2D, accept as input EBCDIC strings that represent hexadecimal values. The output of X2C is binary, while the output of X2D is an EBCDIC string that represents a decimal value.

The input to both functions is hexadecimal 'F1C6'. X2C tells REXX to convert the EBCDIC string into its *binary* hexadecimal form. The EBCDIC string 'F1C6' represents the hexadecimal value '1F'. The output, then, is '1F'. Try executing:

```
say x2c('f1c6'x)
say x2c(1F)
```

Both instructions mean the same thing. By now you can probably predict what will happen: because the output is *binary*, either a blank or an odd character will be displayed.

X2D tells REXX to convert the EBCDIC input of a hexadecimal value into the EBCDIC representation of its decimal equivalent. The EBCDIC string 'F1C6' represents a hexadecimal value of 1F. Decimal notation for hexadecimal '1F' is 31. The EBCDIC representation of '31' is 'F3F1'. Try:

```
say x2d('f1c6'x)
say x2d(1F)
```

Both instructions mean the same thing. The output is EBCDIC, so you will see the characters `31` displayed on your terminal.

Reading 2 continues in Chapter 5, “Conversations,” on page 89.

Character Sets

Reading 3

To translate from one character set to another (for example, to translate data before sending it from an EBCDIC computer to an ASCII printer) use the TRANSLATE( ) function.

Another use would be for changing punctuation, as in this example.

```
/* NOPUNCT EXEC                                            */
/* Example: using the TRANSLATE( ) function to change      */
/* unwanted characters to BLANK                            */
text = "Listen, my children, and you shall hear",
       "Of the midnight ride of Paul      Revere"
say wordpos("my children",text)  /* says "0", because the  */
                                 /* word in TEXT is        */
                                 /* "children,"            */
/*---------------------------------------------------------*/
/* Say whether "my children" can be found in TEXT          */
/*---------------------------------------------------------*/
                              /* remove punctuation        */
nopunct = translate(text,"      ",".;:!,?")
say sign( wordpos("my children",nopunct) )
                              /* says "1"                  */
say sign( wordpos("kids",nopunct) )
                              /* says "0"                  */
```

Figure 49. NOPUNCT EXEC

To help make up strings to put in translation tables use the XRANGE( ) function. For more information on this function see to the *z/VM: REXX/VM Reference*.

The VERIFY( ) Function

To find out whether a string contains only characters of a given character set, use the VERIFY( ) function.

```
►►──VERIFY(string,reference──)─────────────────────────────────►◄
```

returns the position of the first character in *string* that is not also in *reference*. If all the characters in *string* are also in *reference*, zero is returned. For example:

```
/* DIGITS EXEC                                          */
/* Example: testing that all input characters are valid */
say "Please enter the serial number"
say "(eight digits, no imbedded blanks or periods)"
pull serial rest
if verify(serial,"0123456789") = 0,
 & length(serial) = 8,
 & rest = ""
then say "Accepted"
else say "Incorrect serial number.  Please start again"
```

Figure 50. DIGITS EXEC

Reading 3 continues in Chapter 5, “Conversations,” on page 89.

Chapter 5. Conversations

In this chapter:

Reading 1 immediately following, describes:

- How to write lines to the user's screen using the SAY instruction
- How to obtain data from the user's keyboard using the PULL instruction
- How to translate values to uppercase using the UPPER instruction
- How to *parse* this data; that is, to separate it into words and to assign each word or group of words to a different REXX variable.

Reading 2 on page 94, describes:

- How to obtain data from the command line using the PARSE instruction
- How to parse *options* using the ARG instruction
- How to parse variables and expressions.

Reading 3 on page 97, describes:

- How to parse using patterns.

The SAY Instruction

Reading 1

To display data on your screen use:

```
►►──SAY──┬────────────┬──;──────────────────────────────────────────►◄
         └─expression─┘
```

The expression is computed and the result is displayed as a new line on the screen. For example, the instruction:

```
say 3 * 4 "= twelve"
```

causes this to be displayed:

```
12 = twelve
```

If you want to display a clause that occupies more than one line in your program, use a comma at the end of a line to indicate that the expression continues on the next line. For example, the instruction:

```
say "What can't be done today, will have to be put off",
    "until tomorrow."
```

causes this to be displayed:

```
What can't be done today, will have to be put off until tomorrow.
```

Notice that the continuation comma is replaced by a blank when the expression is displayed. (Remember that the continuation comma cannot be enclosed in quotation marks or the language processor will consider it part of the string.)

The PULL Instruction

Having asked the user a question using SAY, you can collect the answer using PULL. When the instruction

```
►►──PULL──┬────────┬──;──────────────────────────────────────────────►◄
          └─symbol─┘
```

is executed the program pauses; `VM READ` appears on the bottom right of the user's screen; the user should enter some data on the command line and press Enter. Whatever the user enters is translated to uppercase and then assigned to the variable `SYMBOL`.

To get the data just as it is, without having the lowercase letters translated to uppercase, use:

```
►►──PARSE PULL──┬────────┬──;────────────────────────────────────────►◄
                └─symbol─┘
```

This example uses both PULL and PARSE PULL.

```
/* CHITCHAT EXEC        */
/* Another conversation */
say "Hello!  What's your name?"
parse pull name
say "Say," name", are you going to the party?"
pull answer
if answer = "YES"
then say "Good.  See you there!"
```

Figure 51. CHITCHAT EXEC

The user's name will be repeated exactly as it was entered. But `ANSWER` will be translated to uppercase. This ensures that whether the user replies *yes*, or *Yes*, or *YES*, the same action is taken.

The UPPER Instruction

To translate the values of one or more variables to uppercase, use the UPPER instruction.

```
            ┌──────────┐
            ▼          │
►►──UPPER────variable──┴──;──────────────────────────────────────────►◄
```

For example, this might have been used in WHATDAY EXEC, Figure 31 on page 56, to let the user reply in mixed case.

```
/* WHATDAY2 EXEC                                    */
/* Example: to make the user say what day of the    */
/* week it is today.  The user's reply may be in    */
/* mixed case.                                      */
today = date(weekday)
upper today                          /* uppercase   */
do until reply = today
   say "What day of the week is it?"
   pull reply                        /* uppercase   */
   if reply ¬= today
   then say "No, it is" today
end
say "Correct!"
```

Figure 52. WHATDAY2 EXEC

Test Yourself...

1. The following program asks a question:

```
/* RIDDLE EXEC */

/* Simple question (?) */
say "Mary, Mary, quite contrary"
say "How many letters in that?"
pull ans
if ans = length(that)
then say "Quite right!"
else say "Oh!"
```

   What happens if the user replies:
   a. 21
   b. 4
   c. Four

2. What would be displayed by:

```
/* NOAH EXEC */

/* Example: expressions that continue for more      */
/* than one line.                                   */
x = 3
say "x =" x
say
say "Ham,",
    "Shem",
    "and Japheth"
say "Silly"
    "Billy"
```

3. Use XEDIT to create a file called PULLIN EXEC containing the following program, then try to run the program!

```
/* PULLIN EXEC */

/* Example: appending input, using PULL,            */
/* to a REXX variable                               */
text = ""
do until input = "QUIT"
   say "Text so far is:"
   say text
   say "Would you like to add to that?",
       " If so, type your message.",
       " If not, type QUIT."
   pull input
   text = text||input
end
```

Answers:

1. What appears on the screen is:
   a. `Oh!`
   b. `Quite right!`
   c. `Oh!`

   Each of these are, of course, followed by `Ready;`.
2. What appears on the screen is:

   ```
   noah
   x = 3
   Ham, Shem and Japheth
   Silly
       10 *-* "Billy"
          +++ RC(-3) +++
   Ready;
   ```

   As there is no comma after `Silly`, `Billy` is treated as a command. If no such command exists CMS sets the return code to minus three. So the language processor displays the line that caused the error and the return code.
3. Did it work? If not, study the error messages and make sure you copied everything correctly.
   a. Notice that:
      - When you run the exec, everything you type in gets changed to uppercase (capital) letters.
      - You are not given any blanks between the old TEXT and the new INPUT.

   b. Now alter `pull input` to `parse pull input`. Alter the concatenate operator "||" to a single blank and try again. Notice that:
      - Your input does not get changed to uppercase.
      - You are always given one blank between the old TEXT and the new INPUT.
      - You cannot get out of the program by entering `quit`. But you *can* get out by entering `QUIT`.

Parsing Words

PULL can also fetch each word into a different variable. In the following example FIRST, SECOND, THIRD, and REST have been chosen as the names of variables:

```
/* PARSWORD EXEC              */
/* An exec that parses words. */
say "Please enter three or more words:"
pull first second third rest
say first second third rest
```

Figure 53. PARSWORD EXEC

If you type “three wise men   on camels” after the prompt (with five spaces between “men” and “on”), you will see this:

```
parsword
Please enter three or more words:
three wise men     on camels
THREE WISE MEN     ON CAMELS
Ready;
```

As usual, the program pauses and the user can type something on the command line. When the user presses Enter, the program continues. The variables are given the values as follows:

Variable	Value
FIRST	“THREE”
SECOND	“WISE”
THIRD	“MEN”
REST	“ ON CAMELS”

In general, each variable gets a word (without blanks) and the last variable gets the rest of the input, if any (with blanks). If there are more variables than words, the extra variables are assigned the null value.

To make sure that the user types in the right number of words, provide one extra variable and test that it is empty. Also, test the variable that holds the last word the user is expected to enter. By testing both variables for a null value, you can be sure that each of your variables contains exactly one word.

```
/* FUSSY EXEC                                           */
/* Example: getting the number of words that you want   */
good = 0
do until good
   say "Please enter exactly three words"
   pull first second third rest
   select
      when third = "" then say "Not enough words"
      when rest ¬= "" then say "Too many words"
      otherwise good = 1
   end
end
```

Figure 54. FUSSY EXEC

The Period as a Placeholder

The symbol “.” (a period by itself) may not be used as a name but it may be used as a place-holder with the PULL instruction. For example,

```
pull . . lastname .
```

would discard the first two words, assign the third word into LASTNAME, and discard the remainder of the input.

Test Yourself...

1. What will be displayed on the screen when this program is run?

```
/* PULLING EXEC */

/* Example: the PULL instruction */
Say "Where did Jack and Jill go?"
parse pull one two three four five six .
        /* User replies "To fetch a pail of water" */
say one two six
say
Say "Will you buy me a diamond ring?"
pull reply .
        /* User replies "Yes, if I can afford it"  */
say reply
```

2. Write a program that asks the user for his name and greets him by his first name. Your program should ignore any other names.

Answers:

1. What appears on the screen is:

```
pulling
Where did Jack and Jill go?
To fetch a pail of water
To fetch water
Will you buy me a diamond ring?
Yes, if I can afford it
YES,
Ready;
```

2. A possible answer would be:

```
/* HOWDY EXEC */

/* Example: selecting a single word */
say "Howdy!  Say, what's your name?"
pull reply .             /* The period causes second */
                         /* and subsequent words to  */
                         /* be ignored               */
say "Pleased to meet you," reply
```

Reading 1 continues in Chapter 6, “Commands,” on page 99.

Getting Data from the Command Line

Reading 2

When you want to run your exec, type its file name on the command line. This can be followed by more data, called arguments.

To obtain the data that the user entered on the command line when starting your program, use the ARG instruction. ARG will parse the arguments in the same way that PULL parses data from the keyboard, except that the first word entered on the command line (the name of the exec) is not parsed. (The ARG instruction gives the same results as the PARSE UPPER ARG instruction.)

In the following program FIRST, SECOND, THIRD, and REST are the variable names:

```
/* MIX EXEC                                          */
/* Example: this program starts by assigning the words  */
/* from the command line to REXX variables              */
arg first second third rest
say first second third rest
```

Figure 55. MIX EXEC

If you type “fresh green salad   and olives” (with three spaces between “salad” and “and”), after the exec name, you will see this:

```
mix fresh green salad   and olives
FRESH GREEN SALAD   AND OLIVES
Ready;
```

When the ARG instruction is executed, the variables are given the values as follows:

Variable	Value
FIRST	“FRESH”
SECOND	“GREEN”
THIRD	“SALAD”
REST	“ AND OLIVES”

Mixed Case

To obtain the data that the user entered on the command line when starting your program, without translating alphabetic characters in the data to uppercase, use the PARSE ARG instruction.

Recognizing Options

In CMS, the ordinary arguments of a command are separated from the options by a left parenthesis. Optionally you can mark the end of the options with a right parenthesis if you wish.

For example,

```
SCRIPT myfile (TWOPASS CONTINUE)
```

tells SCRIPT to process MYFILE SCRIPT with the options TWOPASS and CONTINUE.

Your REXX program can handle data from the command line in a similar way, by using *string patterns*.

String Patterns

To split up the data being parsed, use string patterns. If your PARSE instruction specifies a string (that is, one or more characters enclosed in quotation marks) the data being parsed will be split at the point where the string is found. In this next example, the first pattern is “(” and the second pattern is “)”. The ARG instruction parses the data from the command line.

```
/* TAKE EXEC                    */
/* Example: recognizing options */
arg drink type shelf  "("  opt1 opt2 opt3 ")" rest
say drink type shelf opt1 opt2 opt3 rest
```

Figure 56. TAKE EXEC

If you type “coffee beans (fresh roasted” after the exec name, you will see this:

```
take coffee beans (fresh roasted
COFFEE BEANS  FRESH ROASTED
Ready;
```

When the ARG instruction is executed:

- The words in front of the first pattern will be parsed in the usual way, into DRINK, TYPE, and SHELF. For this example, SHELF will be set to null.
- The words between the first pattern and the second pattern (if there is one) will be parsed in the usual way, into OPT1, OPT2, and OPT3. For this example, OPT3 will be set to null.
- If there is a second pattern, the words that followed it will be parsed into REST. For this example, REST will be set to null.

This technique of parsing using string patterns can be used with any of the parsing instructions.

Parsing Variables and Expressions

As well as parsing replies from the user and the data from the command line, you can parse variables and expressions.

```
                                       ┌──────────────┐
                                       ▼              │
►►──PARSE──┬─VARname───────────────────┬─┬──────────┬─┴──;──────────────────►◄
           └─VALUE──┬────────────┬─WITH┘ └─argument─┘
                    └─expression─┘
```

For example:

```
/* PARSING EXEC                               */
/* Examples: parsing variables and expressions */
phrase = "Three blind mice "
PARSE VAR phrase number adjective noun
say number                      /* says "Three"           */
say adjective                   /* says "blind"           */
say noun                        /* says "mice"            */
PARSE VALUE copies(phrase,2) WITH . a . b . c
say b a c                       /* says "Three blind mice" */
/* and, finally, a very useful trick for taking the        */
/* first word away from a sentence                         */
PARSE VAR phrase first phrase
say first                       /* says "Three"           */
say phrase                      /* says "blind mice"      */
```

Figure 57. PARSING EXEC

Test Yourself...

Modify MYPROG EXEC in Figure 42 on page 75 to use the ARG instruction. Make a further modification to test for a CONTINUE option. Allow any abbreviation of COntinue that is two or more letters long. Test for incorrect options.

Answer:

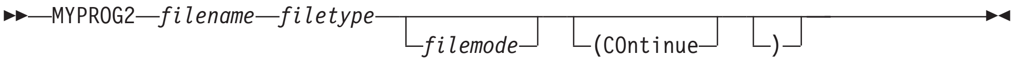
A possible solution is:

```
/* MYPROG2 EXEC */

/*
This program processes the input file to give ...
 ... ...
Correct format is:
    MYPROG2 filename filetype [filemode] [(COntinue [)]]
Function performed is:
Rhubarb, rhubarb, rhubarb.
*/
arg fn ft fm "("option")" rest
if fn = ? | fn = "" | ft = "",
  | option ¬= "" & ¬ abbrev(CONTINUE,option,2),
  | rest ¬= ""
then do
   do line = 2 by 1 while sourceline(line) ¬= "*/"
      say sourceline(line)
   end
   exit
end
/*------------------------------------------------------*/
/* Main program starts here.                            */
/*------------------------------------------------------*/
say "This is the program"
if abbrev(CONTINUE,option,2)
then say "If an error is detected, processing",
         "will continue"
```

Figure 58. MYPROG2 EXEC

When run, the following is displayed:

```
myprog2
This program processes the input file to give ...
 ... ...
Correct format is:
```



```
Function performed is:
Rhubarb, rhubarb, rhubarb.
Ready;
```

Reading 2 continues in Chapter 6, “Commands,” on page 99.

Parsing Using Patterns

Reading 3

The idea of parsing using patterns is fully explained in your *z/VM: REXX/VM Reference*; however, we will briefly describe parsing here.

Data can be parsed using patterns. A pattern is part of the template of a PULL, ARG or PARSE instruction and is recognized if it is:

- In quotation marks, like '(' and ')' in the MYPROG2 EXEC on page 97.
- In parentheses (meaning that it is the name of a variable)
- An unsigned number (meaning that parsing is to continue at the specified character position)
- A signed number (meaning that parsing is to continue at the specified character position, relative to the first character of the last match).

Here is a useful function, in which the second PARSE instruction uses a variable as a pattern.

```
/* CHANGE EXEC                                          */
/* Function:  CHANGE(string,old,new)                    */
/*                                                      */
/* Like XEDIT's  "C/old/new/1 *"                        */
/*                                                      */
/* Changes all occurrences of "old" in "string"         */
/* to "new".  If "old" == "", then "new" is attached    */
/* to the beginning of "string".                        */
parse arg string, old, new
if old=="" then return new||string
out=""
do while pos(old,string)¬=0
   parse var string prepart (old) string
   out=out||prepart||new
end
return out||string
```

Figure 59. CHANGE EXEC

Reading 3 continues in Chapter 6, “Commands,” on page 99.

Chapter 6. Commands

In this chapter:

Reading 1 immediately following, describes:

- How to issue commands to CMS and CP from within your exec
- What are return codes from commands
- The REXX special variable, RC.

Reading 2 on page 105, describes:

- How to debug commands
- How to write a common routine to handle nonzero return codes
- How to access messages from a repository file
- How to suppress messages issued by CMS commands.

Reading 3 on page 115, describes:

- How to suppress messages issued by CP commands
- How to obtain a reply from a CP command
- Using the COMMAND environment as an alternative environment for issuing CMS and CP commands.

Issuing Commands to CMS and CP

Reading 1

The language processor can operate in a number of *environments* (for example, CMS or XEDIT). The way the language processor handles commands depends on the environment it is operating in. For the moment, to keep things simple, let us assume that your program was started by typing its name on the CMS command line. In this case, your program is in the CMS environment.

Clauses That Become Commands

Any clause in your program that the language processor does *not* recognize as an instruction, an assignment, a label, or a null clause will be evaluated and passed to the appropriate environment for execution. For example, if the environment is CMS, CMS and CP commands will be handled in the same way as if they had been entered on the CMS command line.

```
/* Example:  a CMS command in a REXX program          */
"ERASE OLDSTUFF SCRIPT A"
```

The clause that has been recognized as a command is treated as an expression. The language processor will compute the value of the expression in the usual way, and will pass the result to the environment. *The expression is always evaluated first*.

This rule is extremely useful, but you must be careful how you use REXX operators and special characters. Also, look out for use of duplicate names.

- In this example, the value of a variable is substituted in an expression, before the expression is passed to CMS.

```
/* ERASER EXEC                                      */
/* Example:  to erase a number of SCRIPT files.     */
do until fn = "
   say "Enter file name of file to be erased"
   say "  (To return to CMS, enter a null line)"
   pull fn
                        /* The user replies "myfile", */
                        /*    FN =  MYFILE            */
   if fn ¬= " then
   "ERASE" FN "SCRIPT"  /* This clause is treated as  */
                        /* an expression.  The result,*/
                        /* which (in this example) is */
                        /*    ERASE MYFILE SCRIPT     */
                        /* is passed to CMS           */
end
```

Figure 60. ERASER EXEC

- If you want to use a REXX operator or special character as an ordinary character, then you must put it in quotation marks. This is because expressions are evaluated before they are passed to an environment. Therefore, any part of the expression that is not to be evaluated should be written in quotation marks.

  For example:

```
/* ELIST EXEC                                      */
/* Example:  to erase all the files on file mode A   */
/* that have a file type of LIST                   */
"ERASE * LIST"        /* This clause is treated as   */
                      /* an expression.  The result  */
                      /*    ERASE * LIST             */
                      /* is passed to CMS            */
```

Figure 61. ELIST EXEC

In Figure 61, if the asterisk was not in quotation marks, the language processor would attempt to multiply `ERASE` by `LIST`!

Note: Remember to put quotation marks around all operators and parentheses unless already enclosed in quotation marks. Either of the following examples is correct. The last example is better, since nothing has to be evaluated by REXX.

```
"COPYFILE" MYFILE SCRIPT A "=" BACKUP A "(REPLACE"
COPYFILE MYFILE SCRIPT A "=" BACKUP A "("REPLACE
"COPYFILE MYFILE SCRIPT A = BACKUP A (REPLACE"
```

Refer to “When to Use Quotation Marks” on page 101 for more information.

- Another difficulty is the use of duplicate names. In Figure 62 on page 101, the programmer has chosen `A` as the name of a variable. In the COPYFILE instruction, `A` is used as the file mode and must be enclosed in quotation marks; otherwise, the current value of `A` would be substituted.

```
/* BACKUP EXEC                                    */
/* Example:  to save copies of a number of SCRIPT     */
/* files.  Each copy is given the same file name      */
/* as the original, and a file type of BACKUP.        */
do until a = "
   say "Enter file name of file to be backed up"
   say "  (To return to CMS, enter a null line)"
   pull a
                   /* The user replies "myfile",   */
                   /*    A = MYFILE                */
   if a ¬= " then
   "COPYFILE" a "SCRIPT A = BACKUP A (REP"
            /* This clause is treated as an           */
            /* expression.  The result, which in      */
            /* this example is                        */
            /*  COPYFILE MYFILE SCRIPT A = BACKUP A (REP  */
            /* is passed to CMS                       */
end
```

Figure 62. BACKUP EXEC

This example leads on to a more general question.

When to Use Quotation Marks

The syntax for REXX expressions is very flexible. If a symbol, that is not the name of a variable, is written without quotation marks, no error is signaled. The value used in the result is the symbol itself, translated to uppercase. This makes it easier to write simple programs in REXX than in some other languages. However, you must be careful never to use a symbol to stand for itself, when a variable of the same name exists. (In Figure 62, A is the name of a variable, so it must not be used as the literal name of a file mode without putting quotation marks around it.)

In large programs, or programs that are intended to be very reliable, you can voluntarily adopt the rule that every symbol that is not the name of a variable should be in quotation marks. In the example BACKUP EXEC in Figure 62, the COPYFILE command would be written:

```
"COPYFILE" a "SCRIPT A = BACKUP A (REP"
Here, everything is in quotation marks except the symbol "a",
which is the name of a variable.
```

CP Commands

You can write CP commands in a REXX program. Our example is a program that lets you use files that are on another user’s disk. The CP command LINK makes another user’s disk available to you.

```
►►──LINK──┬────┬──userid──hisdisk──mydisk──┬──────┬──┬──────────┬──────────────►◄
          └─TO─┘                           └─mode─┘  └─password─┘
```

where:

userid is the user ID of the person the disk belongs to.

hisdisk
is the virtual address of his disk.

mydisk
is the virtual address that the disk will have on your system. Choose any number that you do not already use.

mode, password
may be required in some installations but are not used in the example found in Figure 63.

(For an introduction to this subject, see “LINK” in the *z/VM: CMS User’s Guide*. For full details, see the *z/VM: CP Commands and Utilities Reference*.)

After LINKing to the other user’s disk, you can use the CMS command ACCESS to make the files on his disk accessible to you.

```
►►──ACCESS──mydisk──filemode──────────────────────────────────────────────►◄
```

For *mydisk*, use the same 3-digit number as you used in the link command. For *filemode*, choose any letter that you do not already use.

Now for the example, suppose someone in your support organization has a number of useful programs that you would like to use. You know that:

- His user ID is HELPDESK.
- The programs are on his disk 196.
- You will not need to use a disk password.

Here is a REXX program that you can use to make everything on his disk available to you.

```
/* LINKHELP EXEC                                         */
/* For linking to Disk 196 belonging to HELPDESK         */
"LINK HELPDESK 196 200"       /* a CP command            */
"ACCESS 200 B"                /* a CMS command           */
```

Figure 63. LINKHELP EXEC

To run the program, type in the command LINKHELP.

Summary

A clause that is an expression by itself will be evaluated, and the result will be passed to the specified environment. By default the result will be passed to CMS; if the result is not known to CMS, it will be passed to CP.

Return Codes

When you write a CMS or CP command in your exec, you should consider what would happen if the command failed to process correctly. For example, a COPYFILE command might result with an error because the user’s disk was full. After such an error, you should at least EXIT from your program. You may also want to issue a warning message to the user.

Here is how you discover such an error. When commands have finished executing, they always provide a return code. A return code of zero nearly always means “all’s well”. Any other number usually means that something is wrong. You can see these codes on your screen when you enter CMS commands from the command line, as in these examples:

```
copyfile profile exec a profile backup a

   Ready;

link fred 591 591
```

```
   FRED not in CP directory
   Ready(00053);

access 591 b

   DMSACC113S B(591) not attached or invalid device address
   Ready(00100);

copyfile profile exec a = = b (for luck

   Invalid parameter LUCK in the option FOR field.
   Ready(00024);

erase junk exec

   File JUNK EXEC A not found
   Ready(00028);
```

The first COPYFILE command worked correctly so the return code was zero and CMS displayed the `Ready;` message on the screen. (When the return code is zero, CMS does not display the return code.) All the other commands failed so CMS displayed their return codes as part of the `Ready;` message. For instance, the return code from the LINK command was 53.

Now that you understand how CMS handles commands and return codes, let us see how the language processor handles them.

Any command that would be valid on the CMS command line is valid as a clause in a REXX program. The language processor treats the clause like any other expression, substituting the values of variables, and so on. The language processor takes the result and passes it to CMS or CP. (The rules are the same as for commands on the CMS command line; for details, see “The CMS Environment” in the *z/VM: REXX/VM Reference*.)

When the language processor has issued a command and CMS or CP has finished executing it, the language processor gets the return code and stores it in the REXX special variable RC. In your program, you should test this variable to see what happened when the command was executed.

For example:

```
"COPYFILE PROFILE EXEC A PROFILE BACKUP A"
if rc ¬= 0
then do
   say "Unexpected return code" rc "from COPYFILE command"
   exit
end
```

The EXIT instruction causes your exec to finish. The language processor gives control back to CMS. This will be explained later in “The EXIT Instruction” on page 149.

To find out what return codes can be expected from a CMS command, look up the command in the *z/VM: CMS Commands and Utilities Reference*. Return codes are listed in the last paragraph of the description of each command.

The return codes associated with CP commands directly correspond to the message numbers. For example, if you received a return code of 22 when executing the LINK command, you could look at the description for message number 022:

```
HCPLNM022E    Virtual device number was not supplied or it was invalid
```

The CP commands are described in the *z/VM: CP Commands and Utilities Reference*.

Special Variables

RC is one of the REXX *special variables*. The other special variables are RESULT and SIGL. You may use RC, RESULT, and SIGL as the names of your own variables, but you should always remember that any of them may be assigned new values by the language processor. For example, the special variable RC is assigned a new value when a command has been executed. (For full details, see the *z/VM: REXX/VM Reference*.)

Test Yourself...

A program is required that will create a file called PR ALL. In this file there is to be a list of all the files on file mode A (a directory in your file space or a R/W minidisk) whose names begin with “PR”.

- Study the CMS command LISTFILE. You will find it in the *z/VM: CMS Commands and Utilities Reference*, or you can get a short description displayed on your screen by entering `HELP LISTFILE`. Use the LISTFILE command to display the required list of files on your screen.
- Study the EXEC option of the LISTFILE command. Write a REXX program that issues a command to generate the required file.
- At the end of the description of LISTFILE in the *z/VM: CMS Commands and Utilities Reference*, you will find a list of possible return codes. Modify your program to handle all possible errors.
- Add to your program a command that RENAMEs the file that has been created as PR ALL A.
- Test your program by running it twice.

Answer:

```
/* LISTPR EXEC */
/* Lists all the files on file mode A whose file names  */
/* begin with "PR".  The result is written into the    */
/* file PR ALL A.  Any previous version of that file    */
/* is overwritten.                                      */
/*                                                      */
/* CMS EXEC A is used as a work file, then destroyed.   */
"LISTFILE PR* * A (EXEC"
if rc ¬= 0
  then do
    say "Unexpected return code" rc "from LISTFILE command"
  exit
end
"ERASE PR ALL A"
"RENAME CMS EXEC A PR ALL A"
if rc ¬= 0
then
  say "Unexpected return code" rc "from RENAME command"
```

Reading 1 continues in Chapter 7, “XEDIT,” on page 121.

Debugging Individual Commands

Reading 2

If you cannot understand what is happening when you enter a command, it is possible that your program did not issue the command correctly. To be sure about this, trace the command that is behaving mysteriously.

```
mad = "Delirious"
...
trace r
"SCRIPT MAD"
trace n
```

Debugging Execs That Contain Commands

As you know, a program that issues a command should always test the return code immediately afterward to see if all is well. One way of doing this is to write:

```
if rc ¬= 0 then ....
```

Also, for programs that are still being tested (or redesigned, or debugged), use the TRACE Errors instruction

```
TRACE E
```

at the beginning of your exec. A nonzero return code will cause the language processor to display the line number of the command in your program, the command, and the return code.

Making a Common Routine for Handling Return Codes

The third way, suitable for programs that can be used by other people, is to use the SIGNAL ON ERROR instruction. This instruction switches on a detector in the language processor that tests the return code from every command. If a nonzero return code is detected, the usual sequence of clauses is abandoned. Instead, the language processor searches through your program for the label

```
ERROR:
```

Processing continues from there. (This label **must** be the symbol ERROR followed by a colon.) The line number of the command is stored in the REXX special variable SIGL. For more information, see “The CALL ON Condition” on page 160 and “The SIGNAL ON Condition” on page 160

You can use SIGL to tell the user which command caused typical processing to be interrupted:

```
...
signal on error
COPYFILE ... ...
"RENAME" ... ...
exit                        /* End of main program      */
/*-----------------------------------------------------*/
/* Error handler:  common exit for nonzero return codes */
/*-----------------------------------------------------*/
ERROR:
say "Unexpected Return Code" rc "from command:"
say "     " sourceline(sigl)
say "at line" sigl"."
```

The EXIT instruction is put there to stop the main program from running on into the error handling routine.

To switch off the detector, use the instruction:

```
SIGNAL OFF ERROR
```

If you know that one of your commands *can* give a nonzero return code, you must switch off for that one command. For example, if you do not know whether OLD LISTING exists, but need to erase it if it does, this series of instructions will do.

```
signal off error
"ERASE" old listing a
signal on error
```

Getting Messages from a Repository File

You can store message texts in a single file that is separate from your program. The CMS XMITMSG command lets you then access and display these messages from a REXX EXEC. See the *z/VM: CMS Commands and Utilities Reference* for a complete description of XMITMSG.

When using XMITMSG in a REXX EXEC, variables are enclosed in quotation marks. For example:

```
/* In these examples we use message number 3,                  */
/* which has one substitution.                                 */

buffer = 'bufferit'       /* Variable with the name of buffer. */

XMITMSG 003 BUFFER        /* This will not work because the    */
                          /* variable buffer resolves to       */
                          /* bufferit, which is itself not a   */
                          /* variable, so no substitution      */
                          /* takes place.                      */

'XMITMSG 003 BUFFER'      /* This example will work because    */
                          /* the variable buffer is in         */
                          /* quotation marks and gets passed   */
                          /* to XMITMSG.                       */
                          /* bufferit is substituted.          */
                          /*     continued ...                 */
'XMITMSG 003 "BUFFER"'    /* Here we substitute the literal    */
                          /* string BUFFER, which will be      */
                          /* taken as the substitution.        */

'XMITMSG 003 8002'        /* This example shows the use of a   */
                          /* dictionary item, (8002).          */
                          /* The value of 8002 as a dictionary */
                          /* item is the literal string BUFFER.*/

'XMITMSG 003 "8002"'      /* This example is another example   */
                          /* of passing literal strings.       */
                          /* In this case, the number 8002     */
                          /* gets passed as a substitution     */
                          /* instead of resolving to BUFFER    */
                          /* because 8002 is in quotation      */
                          /* marks.
```

Note: This is not a complete program and cannot be executed by itself.

How to Suppress Messages Issued by CMS Commands

To suppress all output (except Severe and Terminating messages from CMS commands), use the Halt Typing command.

```
SET CMSTYPE HT
```

To resume typical output, use the Resume Typing command.

```
SET CMSTYPE RT
```

Be sure that your program processes SET CMSTYPE RT before you need to process a SAY instruction. Also, remember that SET CMSTYPE RT will change the special variable RC. If the old value will be needed, it must be saved. In this example, the return code we are interested in is saved in RCSAVE (RC is overlayed by the second SET command).

```
oldtype = CMSFLAG("CMSTYPE")

if oldtype=1
then oldtype=RT
else oldtype=HT

"SET CMSTYPE HT"
"STATE" fn ft fm         /* Does the file exist?          */
rcsave = rc
"SET CMSTYPE" oldtype    /* Assigns another value to RC   */
if rcsave = 28           /* Is the return code from the   */
then ...                 /* STATE command 28 (not found)? */
```

A Useful Subroutine

All of the preceding code makes your program rather difficult to read. So it would be better to use a subroutine, like this:

```
...
signal on error
...
call quiet "STATE" fn ft fm     /* Does the file exist? */
if RESULT = 28 then ...         /* Set by subroutine's  */
                                /*  RETURN instruction  */
...
exit                            /* End of main program  */
/*------------------------------------------------------*/
/* QUIET                                                */
/* =====                                                */
/* Subroutine to issue a CMS command without displaying */
/* a message on the screen and without jumping to ERROR */
/* if the return code is nonzero.                       */
/*                                                      */
/* The first argument is the command to be executed.    */
/* On returning to the caller, the REXX special         */
/* variable RESULT contains the return code from        */
/* this command.                                        */
/*------------------------------------------------------*/
QUIET:
signal off error       /* Coding note:  the null string */
"SET CMSTYPE HT"       /* prevents ARG from being       */
""arg(1)               /* treated as an instruction.    */
rcsave = rc
"SET CMSTYPE RT"
return rcsave
/*------------------------------------------------------*/
/* Error handler:  common exit for nonzero return codes */
/*------------------------------------------------------*/
ERROR:
say "Unexpected Return Code" rc "from command:"
say "      " sourceline(sigl)
say át line" sigl..."
```

Figure 64. Example Subroutine

Note: This is not a complete program and cannot be executed by itself.

Test Yourself...

Review the following program. Make sure that you understand what it is supposed to do. Will it always work correctly?

```
/* PAIRS EXEC                                             */
/* This program requests the user to supply a list of     */
/* files (file name file type only) and replies, for      */
/* each file:                                             */
/*                                                        */
/*  * whether it is on the user's directory or minidisk   */
/*    accessed as file mode A.                            */
/*                                                        */
/*  * whether it is on the directory or minidisk          */
/*    accessed as file mode L.                            */
/*                                                        */
/*  * if there is a copy on each file mode, whether       */
/*    these copies are the same.                          */
/*                                                        */
/* To end the list, the user returns a null line.         */
/*                                                        */
/* Command format: PAIRS                                  */
if arg() ¬= 0             /* help needed                  */
then do n = 1 until LEFT(line,2) ¬= "/*"
   line = sourceline(n)
   say line
   end
   else do forever
            .
            .
            .
   do until ft ¬= " & rest = ""             /* Get fn ft  */
      say "Enter file name and file type",
          "(or null line to exit)"
      pull fn ft rest
      if fn = "" then exit
   end
   home = ""
   call quiet "STATE" fn ft "A"     /* Compute Home, a    */

   if result = 0 then home = "A"    /* list of file modes */
   call quiet "STATE" fn ft "L"     /* where the file     */
   if result=0 then home = home "L" /* can be found       */
   select
      when words(home) = 0
      then say "No files found"
      when words(home) = 1
      then say "Only one file found (on file mode "home")"
      otherwise
      call quiet "COMPARE" fn ft "A" fn ft "L"
                                    /* continued ...      */
```

Figure 65. PAIRS EXEC (Part 1 of 2)

```
      select
         when result = 0
         then say "Same file found on both file modes",
                  "(A and L)"
         when result = 4,  /* files do not match          */
          | result = 32,   /* files have different        */
                       ,   /* formats or LRECLs           */
          | result = 40    /* files not the same length   */
         then say "Files on file modes A and L",
                  "are not the same"
         otherwise say "Unexpected return code" result,
                       "from COMPARE command"
      end                  /* select result   */
   end                     /* select words()  */
end                        /* end do foreever */
exit                       /* end of main program         */
/*--------------------------------------------------------*/
/* Subroutine to issue a CMS command WITHOUT displaying   */
/* a message on the screen and WITHOUT jumping to ERROR   */
/* if the return code is nonzero.                         */
/*                                                        */
/* The first argument is the command to be executed.      */
/* On returning to the caller, RESULT contains the        */
/* return code from this command.                         */
/*--------------------------------------------------------*/
QUIET:
signal off error
"SET CMSTYPE HT"
""arg(1)
rcsave = rc
"SET CMSTYPE RT"
return rcsave
```

Figure 65. PAIRS EXEC (Part 2 of 2)

Answer:

The program will run correctly.

Using the Program Stack

The *program stack* passes data to certain CMS commands, or to obtain data from them.

- We begin with a careful description of the program stack; this will make it easier for you to use later.
- This is followed by a *cookbook* list of things to do when using the program stack in a REXX program.
- Next comes an example of a command putting data into the program stack. Some commands that can do this are:

LINEIN/CHARIN	to read lines or characters from a directory or minidisk
IDENTIFY	to obtain the node ID, rscs ID, and so on
LISTDIR	to find out about directories
LISTFILE	to find out about files
NAMEFIND	to obtain information from a NAMES file
QUERY	to find out about your CMS virtual machine
RECEIVE	to read in files and notes
RDR	to find out what files are in your reader.

- And finally, an example of a command that takes data from the program stack. Some commands that can do this are:

LINEOUT/CHAROUT	to write lines or characters to a directory or minidisk
COPYFILE	to copy files (using the SPECS option)
FORMAT	to format a minidisk
SORT	to sort a file.

Definitions

In computer science, a *stack* is a list of items that you can work with from only one end, the top. You can PUSH an item onto the stack or PULL an item off from it. The item you PULL off will always be the last item you (or somebody else) PUSHed on. This method is called LIFO—last in, first out.

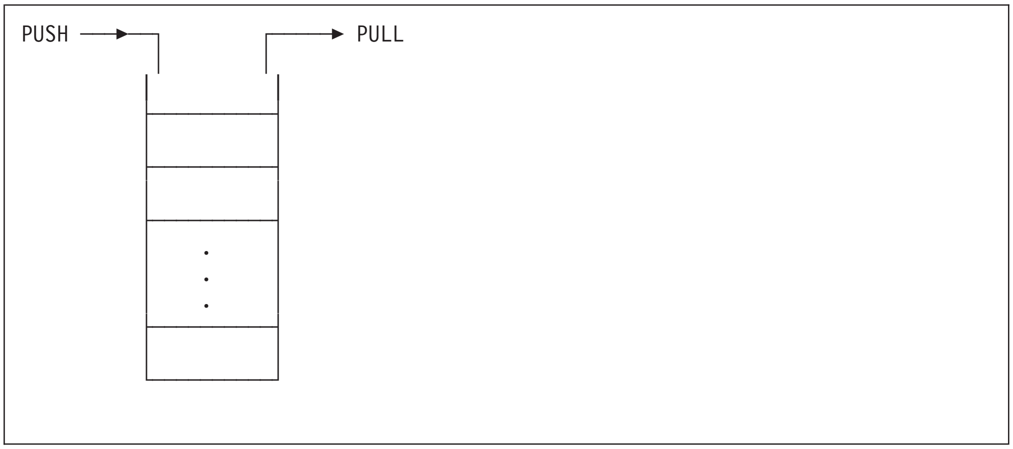


Figure 66. A Stack Using Push and Pull

A *queue*, on the other hand, is a list of items which you can work with from both ends. You can QUEUE (or add) items only at the back and you can PULL items only off at the front. This method is called FIFO—first in, first out.

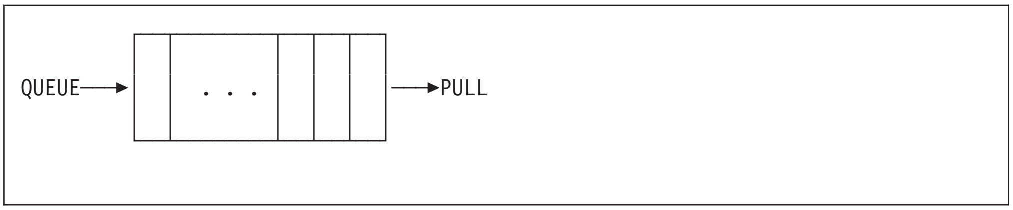


Figure 67. A Stack Using Queue and Pull

The CMS program stack can be used both as a stack and as a queue.

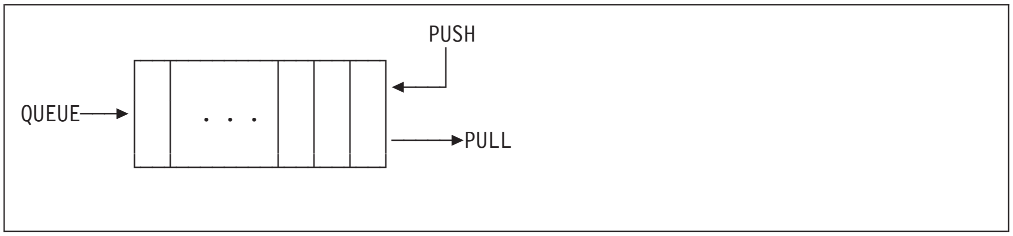


Figure 68. A Stack Using Queue, Push, and Pull

You can use the program stack as a kind of mailbox. CMS commands, for example, can put data in and a REXX instruction can retrieve it for you. Or, a REXX instruction can put data in and a CMS command can retrieve it.

In fact, the program stack can be accessed using REXX instructions, CMS commands, CMS EXEC control words, Callable Services Library routines, and Assembler language macros. But we shall only discuss the first two of these. The table gives you the keywords used in the different languages.

Table 3. Keywords Used in Programming Languages

REXX instruction	QUEUE	PUSH	PULL
CMS command option	(STACK FIFO (FIFO	(STACK LIFO (LIFO	Depends on command
CMS EXEC or EXEC 2 control word	&STACK FIFO	&STACK LIFO	&READ
Assembler macro	CMSSTACK FIFO	CMSSTACK LIFO	LINERD
Callable Services Library routines	StackWrite	StackWrite	StackRead

where:

FIFO means First In, First Out (as in a queue).

LIFO means Last In, First Out (as in a stack).

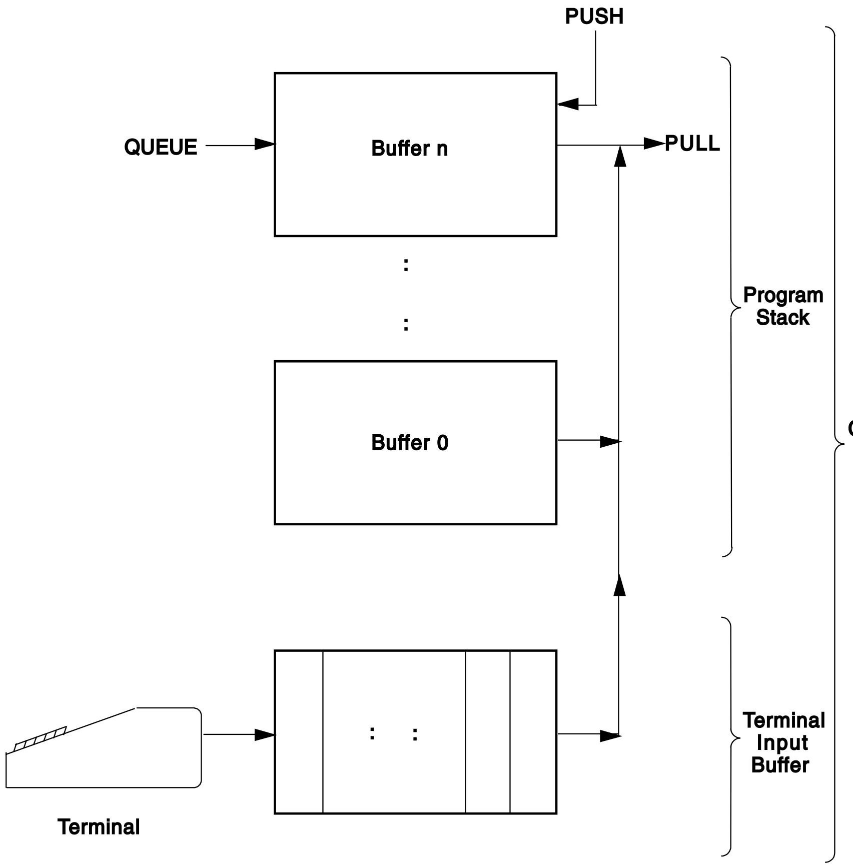
Buffers

A *buffer* is a general term for a part of the computer's storage that is used for input or output.

You can build extensions to the program stack, which are called buffers. Usually there is only one buffer in the program stack.

- You can create new buffers using the MAKEBUF command.
- QUEUE, PUSH and their equivalents put data into the last buffer created.
- PULL and its equivalents remove data from the last buffer created until it is empty, then from the previous buffer until it is empty, and so on.
- When the program stack is completely empty, data is taken from the *terminal input buffer*.

This is what you might call *a stack of buffers*. The entire stack is called the *console stack*.



You may have noticed the terminal input buffer already. The buffer stores data from the CMS command line when you type ahead and press enter while a previous command is still executing.

- If there is nothing in the program stack or the terminal input buffer when a PULL or its equivalent is executed, the program stops, the words `VM READ` appear in the bottom right-hand corner of your screen, and nothing happens until you press Enter, a Program Function key, or certain other keys, depending on the type of terminal you are using.

How to Use the Program Stack

Using the program stack is not quite as complicated as it looks, (as you will see when you read the examples which follow.) The safest way to use the program stack is this:

1. Begin the stack-processing portion of your program with the CMS command MAKEBUF. This will set up your own buffer in the program stack.
2. Find out how many entries are already on the stack using the QUEUED( ) function. For example:

   ```
   theirs = queued()
   ```
3. Use the QUEUE instruction or an equivalent CMS command to put data onto the program stack.
4. Use the PULL instruction or an equivalent CMS command to take data off the stack. If you issue too many PULL instructions the user might see, on the bottom right of the screen:

```
                                                VM READ
```

To continue, you must press Enter.

5. It is important to avoid removing items that your program did not place on the program stack. Remove the items one at a time, first checking that what you are about to remove is yours. For example:

   ```
   do while queued() > theirs   /* THEIRS are not ours */
      pull ...                  /* (see the preceding
                                   information)       */
      ...
   end
   ```

6. Be sure that you have removed all your data from the program stack before you return to CMS. You can use the CMS command DROPBUF to do this.

   Each line left in the program stack, when your REXX program has finished and CMS gets control, will be treated by CMS as a command. Perhaps the user will see the message:

   ```
   Unknown CP/CMS command
   ```

   Or, perhaps something quite unexpected will happen!

This can be simplified slightly. If you are sure that your program will never try to remove items belonging to other programs from the program stack, you can omit Steps 2 and 5.

You might also leave out the commands MAKEBUF and DROPBUF, and nothing would appear to go wrong. But you could have trouble one day, if your exec is called by a program that also uses the program stack. So it is best to use MAKEBUF and DROPBUF in all programs that use the program stack.

Example: A CMS Command That Puts Data onto the Program Stack

This simple program issues a warning message when your primary minidisk, file mode A, is more than 80 percent full. This means that it is time to get a bigger minidisk, or else erase some files you will never need again! You could call this program from your PROFILE EXEC.

Note: This program does not work for a directory. Although the QUERY DISK command provides information about accessed minidisks and accessed directories, the line that describes a directory is different from the line that describes a minidisk.

Before reading this example, try out the CMS command

```
QUERY DISK A
```

Notice that two lines appear on the screen. In a REXX program, to make QUERY put these two lines into the program stack, use the STACK option of the QUERY command.

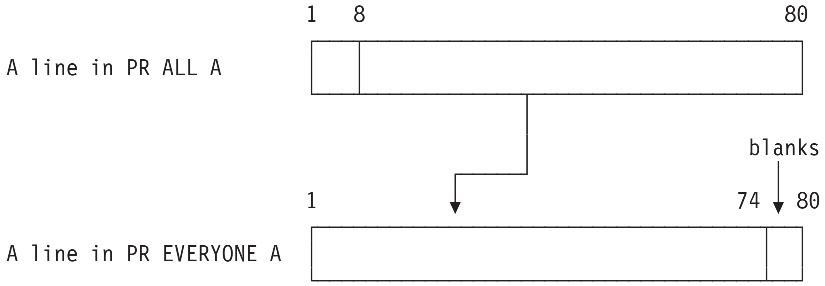
```
/* NEARFULL EXEC                                        */
/* Gives a warning when the user's primary minidisk     */
/* (file mode A) is more than eighty percent full       */
"MAKEBUF"
"QUERY DISK A ( STACK"
if rc = 0 then do
   pull                    /* Discard header            */
   parse pull "-" percentage .
   if percentage > 80
   then say "Warning:  Your disk",
   "is" percentage"% full"
end
else say "NEARFULL EXEC: unexpected return code" rc
"DROPBUF"
```

Figure 69. NEARFULL EXEC

Example: A CMS Command That Requires Data from the Program Stack

There are several CMS commands that ask questions and require answers from the user. To provide these answers from your program, use the program stack.

Here is an example. The file PR ALL A is to be copied into a new file, PR EVERYONE A, moving all the data seven positions to the left.
The COPYFILE command with the SPECS option asks the user to specify the fields in each line of the input file that are to appear in each line of the output file, and where in that line they are to appear. For details, see “COPYFILE” in the *z/VM: CMS Commands and Utilities Reference*.



In this program, the answer is provided by the language processor; it is QUEUEd onto the program stack before the COPYFILE command is issued.

```
/* LEFT7 EXEC                                          */
/* This program will copy the file PR ALL A into a     */
/* new file PR EVERYONE A, shifting the data in        */
/* columns 8 through 80 into column 1, discarding      */
/* columns 1 through 7 and making columns 74 through   */
/* 80 blank.  If the file PR EVERYONE A already        */
/* exists it will be overwritten.                      */
"MAKEBUF"
queue "8-80 1"
"COPYFILE PR ALL A PR EVERYONE A ( SPECS NOPROMPT REPLACE"
if rc ¬= 0 then say "Unexpected return code",
                    rc "from COPYFILE command."
"DROPBUF"
```

Figure 70. LEFT7 EXEC

Reading 2 continues in Chapter 7, “XEDIT,” on page 121.

CP Commands

Reading 3

You will sometimes need to use CP commands in your programs. The following explains how to suppress messages and obtain replies from CP commands.

How to Suppress Messages Issued by CP Commands

To issue a command to CP, suppressing messages and obtaining only the return code, use either the CMS command EXECIO (see the *z/VM: CMS Commands and Utilities Reference*) or the CMS command PIPE:

```
"PIPE CP" cp_command
```

where:

PIPE is a CMS command.

CP is a CMS Pipelines stage command specifying that the remainder of the command text is the CP command to be issued. When used in a REXX program, this can be followed by an expression.

Our example is about a temporary minidisk. If you need to compile something and there is not enough room for the output files in your file space or on your primary minidisk (file mode A), you can obtain a temporary minidisk from CP and put the output files on that minidisk. (Do not put files containing original information on temporary minidisks; if VM has an error, your files could be lost forever.) To obtain a temporary minidisk, with the physical characteristics of an IBM* 3380, a virtual address (vdev) of 192 and an extent of five cylinders, you could type on the CMS command line:

```
define t3380 as 192 cyl 5
```

CP would reply:

```
DASD 192 DEFINED 0005 CYL
```

DASD means Direct Access Storage Device; in this case, the reply refers to a virtual DASD (a minidisk).

To issue the same command from a REXX program suppressing the reply, use:

```
"PIPE CP DEFINE t3380 as 192 cyl 5"
if rc ¬= 0 then ...
```

How to Obtain the Reply from a CP Command

To obtain the reply from a CP command in a REXX program, use:

```
"PIPE CP" cp_command "| STEM RESPONSE."
```

where:

| the CMS Pipelines stage separator that indicates that the output from the first stage (the reply from the CP command) will be the input to the next stage (the stem).

STEM a CMS Pipelines stage command that places the input to the stage into a named stem variable (in this case, RESPONSE.). STEM.0 will contain the number of lines put in the stem, starting with STEM.1.

In our example, RESPONSE.0 will be the number of lines that would usually be displayed on your terminal. RESPONSE.1 will be the first line, RESPONSE.2 will be the second line, and so on. You can also use more advanced features of CMS Pipelines to process the CP output data before building the stem variable.

Another method of obtaining a reply from a CP command is to use the EXECIO CMS command (see the *z/VM: CMS Commands and Utilities Reference* for more information).

Before reading this next example, try out the command:

```
Q DASD
```

CP replies with a list of the minidisks defined for your virtual machine. The TDISK program in the next example reads this list. It then looks through the list for a *vaddr* (virtual address) and a *file mode* that are **not** on the list, and which can, therefore, be used as the vaddr and file mode of a temporary minidisk.

```
/* TDISK EXEC                                            */
/* This program obtains a temporary minidisk, using     */
/* a virtual address (vaddr) and a file mode that are   */
/* not already in use.  The number of cylinders may     */
/* be specified as the first and only argument.  The    */
/* default is 5.                                         */
/*                                                       */
/* If the program was called from the command line and  */
/* is successful, the virtual address and file mode are */
/* displayed.  Otherwise an error message is displayed. */
/*                                                       */
/* If the program was called as a SUBROUTINE (that is,  */
/* by a CALL instruction in a REXX program) or as a     */
/* REXX function, no messages are displayed.            */
/*                                                       */
/* If the program is successful, the return code is     */
/* zero.  If the argument is present and not numeric,   */
/* the return code is 16.  If all 26 file modes are in  */
/* use, the return code is 27.  Otherwise, the return   */
/* code is that of the CMS or CP command that prevented */
/* success.                                              */
```

Figure 71. TDISK EXEC (Part 1 of 3)

```
/*------------------------------------------------------*/
/* Check argument                                       */
/*------------------------------------------------------*/
if arg() = 0              /* argument supplied?         */
then cylinders = 5
else do
   arg cylinders .
   if ¬ datatype(cylinders,whole)
   then do                      /* help needed          */
      do n = 1 while LEFT(line,2) = "/*"
         line = sourceline(n)
         say line
      end
      return 16
   end
end
/*------------------------------------------------------*/
/* How was this program called                          */
/*------------------------------------------------------*/
parse source . howcalled .        /* See the REXX/VM    */
                                  /* Reference          */
/*------------------------------------------------------*/
/* Find unused virtual address                          */
/*------------------------------------------------------*/

                                  /* continued ...      */
"MAKEBUF"
signal on error
"PIPE CP QUERY VIRTUAL DASD",  /* Query attached dasd.            */
  "| SPECS WORD 2 1",          /* Keep only the virtual addresses.*/
  "| JOIN * / /",              /* Join them all into one line.    */
  "| VAR USED"                 /* Load them into variable USED.   */
do newcuu = 200 while pos(newcuu,used) ¬= 0 end
/*------------------------------------------------------*/
/* Find unused file mode                                */
/*------------------------------------------------------*/
alphabet = "ABCDEFGHIJKLMNOPQRTUVWXYZ"
do letter = 1 to 25 until response = "NOT ACCESSED"
   "QUERY DISK" substr(alphabet,letter,1) "(LIFO"   /* PUSH
*/
   pull . . response     /* get last line of last reply */
                         /* pull instruction puts "not  */
                         /* accessed" in uppercase      */
end
signal off error; "DROPBUF"   /* clear our buffer       */
```

Figure 71. TDISK EXEC (Part 2 of 3)

```
"MAKEBUF"; signal on error
if letter = 27 then do
   if howcalled = "COMMAND"
      then say "All file modes in use"
   return 27
end
newfm = substr(alphabet,letter,1)
/*-------------------------------------------------------*/
/* Obtain and format minidisk                            */
/*-------------------------------------------------------*/
"PIPE CP DEFINE T3380 AS" newcuu "CYL" cylinders
push "TEMP"
push "YES"
"SET CMSTYPE HT"
"FORMAT" newcuu newfm
"SET CMSTYPE RT"
signal off error
"DROPBUF"
exit

                                 /* continued ...        */
/*-------------------------------------------------------*/
/* Non-zero return codes                                 */
/*-------------------------------------------------------*/
ERROR:
rcsave = rc
"SET CMSTYPE RT"
"DROPBUF"
if howcalled = "COMMAND"
then do
   say "Unexpected return code" rcsave
   say "from command" sourceline(sigl)
   say "at line" sigl
end
exit rcsave
```

Figure 71. TDISK EXEC (Part 3 of 3)

The COMMAND Environment

So far, we have said that the language processor handles CMS and CP commands in exactly the same way as if they had been entered from the CMS command line. This is called the *CMS environment*; it was chosen as the default because it is the one that most programmers will want to use, most of the time. But there is an alternative environment, the *COMMAND environment*, which has some advantages.

You should use the COMMAND environment:

1. To avoid calling a user's exec, which happens to have the same name as a CMS command. For example, suppose you send a copy of your program to another user, or put your program in a directory or on a minidisk that other users can access. Your program contains the clause "sort ... "; you are telling the language processor to process the CMS command SORT.

   When this command is executed from your program using the usual CMS search order, there might be a file called SORT EXEC in the directory or minidisk that the user has accessed as A. If so, CMS will call the user's exec instead of the command! As far as you are concerned, the result is unpredictable. But to have CMS search for a SORT MODULE—CMS commands are stored in files with a file type of MODULE—write:

```
ADDRESS COMMAND SORT ...
```

And, so long as the SORT MODULE is not on the user's disks, your program will run as you expect.

2. To suppress messages from certain commands. For example, the commands ERASE, LISTFILE, RENAME and STATE issue the message `FILE NOT FOUND` when the specified file is not found and the command was entered from the command line or from a REXX program. If you think a person using your program would find this message confusing, write

   ```
   ADDRESS COMMAND "STATE" fn ft
   ```

   (for example) and the message will be suppressed.

   To suppress nearly all messages, use SET CMSTYPE HT. (See page 106 for details.)
3. To reduce system overhead. This can be important if the user has a large number of directories or minidisks accessed. Each time your program issues a command, CMS searches these directories and minidisks for an exec file of that name before it searches for a MODULE file. (CMS commands are stored in files with a file type of MODULE.)

Instead of writing ADDRESS COMMAND in front of each clause, you can write

```
ADDRESS COMMAND
```

at the beginning of your program. This has the same effect as if all commands have a prefix of ADDRESS COMMAND. If you have done this, and you want to switch back to the CMS environment, use:

```
ADDRESS CMS
```

For more information, see the ADDRESS command in the *z/VM: REXX/VM Reference*.

Reading 3 continues in Chapter 7, "XEDIT," on page 121.

Chapter 7. XEDIT

XEDIT is the editor supplied with z/VM. You can customize XEDIT for your own purposes by writing special REXX programs called macros. This chapter introduces some important ideas about these programs.

In this chapter:

Reading 1 immediately following, describes:

- How your program can be called from the XEDIT command line
- How to enter subcommands to XEDIT from your REXX program
- Names for XEDIT macros
- Return codes from XEDIT subcommands
- How to display messages in the XEDIT message area.

Reading 2 on page 124, describes:

- **Note:** You should not attempt this reading until you have a working knowledge of XEDIT.
- How the private variables of XEDIT can be made available to your REXX program, using the EXTRACT command
- The current line of a file
- An example XEDIT profile.

Reading 3 on page 126, describes:

- How to construct a menu.

XEDIT Subcommands and Macros

Reading 1

Commands to XEDIT are usually called *subcommands* to avoid any possible confusion with commands to CMS.

When you are using XEDIT and you type a word on the XEDIT command line and press Enter, XEDIT will treat this as a:

Subcommand
If the first word on the command line is one of the XEDIT subcommands (defined in the *z/VM: XEDIT Commands and Macros Reference*), XEDIT will obey it.

Macro
If the word is not a subcommand, XEDIT will look for a file of the same name with a file type of XEDIT and execute that. This type of file is called a *macro*.

For example, if the file TEN XEDIT, shown in Figure 72, exists in a directory or on a minidisk that you have accessed, and you type the word

```
ten
```

on the XEDIT command line, XEDIT will try to execute TEN XEDIT.

Note: To find a file in a directory, read authority is required on both the file and the directory. If the file is locked, the execution will result in an error and give you an error message.

CMS or CP command
If a macro does not exist, XEDIT will try to execute what is typed in as a CMS or CP command.

XEDIT Macros

A REXX program that issues subcommands to XEDIT is called a *macro*. It must have a file type of XEDIT. To indicate that your program is written in the REXX language, it must begin with a REXX comment, as usual.

Because the file type of your program is XEDIT, the language processor will assume that the environment is XEDIT. And, therefore, any clause in the program that the language processor does *not* recognize as an instruction, an assignment, a label, or a null clause will be evaluated in the usual way and the result will be passed to XEDIT for execution.

Naming of XEDIT Macros

XEDIT macros, like other CMS files, can have file names from one-to-eight characters long. The file names of XEDIT macros should not contain numeric digits. (This is because XEDIT treats the number as an argument. For example, `MYMAC5` is the same as `MYMAC 5`.)

Example: Changing the Settings of the Scroll Keys

When you are looking through a file, you will usually want to move forward or backward a page at a time. Sometimes you may prefer to move forward or backward half a page at a time. For example, you can use this when checking a program. This forward and backward movement through your file is called scrolling.

Use XEDIT to create the following file called TEN XEDIT.

```
/* TEN XEDIT                                          */
/* This program changes the settings of PF Keys 7 and 8 */
/* so that you scroll backward or forward 10 lines      */
/* at a time.                                           */
"SET PF7 UP 10"
"SET PF8 NEXT 10"
```

Figure 72. TEN XEDIT

Now use XEDIT to display any large file. Type TEN on the XEDIT command line, and press Enter. Press PF8 to scroll down the file. Each time you press PF8 you will advance 10 lines down the file. Similarly, each time you press PF7 you will move 10 lines nearer the top of the file.

To restore the setting that XEDIT usually provides, you could use this program.

```
/* PAGE XEDIT                                        */
/* This program changes the settings of PF Keys 7 and 8 */
/* so that you scroll backward or forward one page      */
/* at a time.                                           */
"SET PF7 BACKWARD"
"SET PF8 FORWARD"
```

Figure 73. PAGE XEDIT

Return Codes

Your REXX program should be able to handle nonzero return codes from XEDIT subcommands.

To find out what return codes can be expected from an XEDIT subcommand, look up the subcommand in the *z/VM: XEDIT Commands and Macros Reference*. Return codes are listed in the last paragraph of the description of each command. For example, the XEDIT subcommand

```
NEXT
```

will give a return code of 1 when end of file is reached.

When you are first learning to write XEDIT macros, you should put the instruction TRACE Errors at the top of your program. This will cause a trace to be displayed if any XEDIT command gives a nonzero return code. For example:

```
/* DENTAL XEDIT                    */
/* Example: tracing a syntax error */
trace errors
"EXTRACT" tooth      /* EXTRACT is a valid command, but */
                     /* "tooth" is not a valid operand  */
```

Figure 74. DENTAL XEDIT

Executing the command DENTAL from the XEDIT command line would cause the following to be displayed:

```
3 *-* "EXTRACT" tooth /* EXTRACT is a valid command, but */
  +++ RC(5) ++++
```

Messages

To display messages in the XEDIT message area, use the XEDIT MSG subcommand:

```
MSG text of message
```

For example:

```
"NEXT"
if rc = 1 then "MSG" "End of file reached"
```

Reading 1 continues in Chapter 8, “Control,” on page 129.

The EXTRACT Subcommand

Reading 2

To obtain almost any variable known to XEDIT, use the EXTRACT subcommand.

For example, the physical size of your screen might be 24 lines or 32 lines; and you could find out the size of your screen by entering QUERY SCREEN on the XEDIT command line.

To obtain the same information for use in your REXX program, enter:

```
EXTRACT /SCREEN/
```

The EXTRACT subcommand requires a delimiter to separate the operands. In this book, we shall use / as the delimiter. Notice how / is used in the preceding EXTRACT command. For this example, it would also be correct to enter:

```
EXTRACT /SCREEN
```

The EXTRACT /SCREEN subcommand assigns values to an array of REXX variables:

SCREEN.0
The number of other variables in the array. (That is, 1 in this case.)

SCREEN.1
Two words, namely the word SIZE followed by the number of lines on the screen.

We could use this subcommand to extend the program TEN XEDIT, described above, to handle any size screen:

```
/* HALF XEDIT                                          */
/* This program changes the settings of PF Keys 7 and 8 */
/* so that you scroll backward or forward half a       */
/* screen at a time.                                   */
"EXTRACT /SCREEN"
amount = (substr(screen.1,6) - 4) % 2
"SET PF7 UP" amount
"SET PF8 NEXT" amount
```

Figure 75. HALF XEDIT

`EXTRACT /SCREEN` assigns SIZE 24 or SIZE 32 to `SCREEN.1`; the SUBSTR( ) function returns the number from this; and the value that `amount` gets will be either 10 (for 24-line screens) or 14 (for 32-line screens).

The Current Line

The *current line* of a file is used as the starting-point for many XEDIT subcommands. You can change its physical position on the screen by using the SET CURLINE subcommand. The default position is the line above the middle of the screen.

To obtain information about the current line, use the XEDIT subcommand:

```
EXTRACT /CURLINE
```

This command assigns values to an array of REXX variables:

CURLINE.0 The number of other variables in the array

CURLINE.1 The operand that positioned the current line on the screen (see the *z/VM: XEDIT Commands and Macros Reference*)

CURLINE.2 The line number of the current line on the screen

CURLINE.3 The contents of the current line

CURLINE.4 ON if the current line has been changed or inserted in this editing session; OFF otherwise.

Here, the most interesting variable is CURLINE.3 (the file data that is displayed on the current line). We shall use it in the next example.

An Example: Moving through a File a Paragraph at a Time

In some files (like the example programs in this book) the writer leaves a blank line between one paragraph and the next. This next program lets you scroll through the file a paragraph at a time.

```
/* PARA XEDIT                                            */
/* This program scrolls forward until the line above     */
/* the current line is blank.  If end of file is         */
/* reached, or if there is an unexpected error, an       */
/* audible warning is given.                             */
do until curline.3 = ""
   "EXTRACT /CURLINE"
   "NEXT"
   if rc ¬= 0 then do
      "SOS ALARM"           /* an XEDIT subcommand: sound */
                            /* audible alarm.  (bleep)    */
      exit
   end
end
```

Figure 76. PARA XEDIT

Your XEDIT Profile

The program PROFILE XEDIT is automatically executed every time you start to edit a new file. Following is an example of a profile that you can use in XEDIT. For more information on creating XEDIT profiles, refer to the *z/VM: XEDIT User's Guide*.

```
/* PROFILE XEDIT                                 */
/* Profile XEDIT to customize XEDIT environment  */
signal on error
/* * * * * * * * * * * * * * * * * * * * * * * * */
/* set desired pf keys not defaulted             */
/* * * * * * * * * * * * * * * * * * * * * * * * */
"SET PF13 FILE"
"SET PF16 LEFT 20"; "SET PF17 RIGHT 20"
/* * * * * * * * * * * * * * * * * * * * * * * * */
/* tailor XEDIT to my specifications             */
/* * * * * * * * * * * * * * * * * * * * * * * * */
"SET VERIFY 1 72"
"SET NULLS ON"
"SET FULLREAD ON"
"SET CASE MIXED IGNORE"
"SET WRAP ON"
"SET HEX ON"
"SET AUTOSAVE 10"
"SET MSGLINE ON 3 OVERLAY"
"SET SCALE ON 2"
"SET CURLINE ON 8"
"SET NUM ON"
"SET PREFIX NULL LEFT"
/* * * * * * * * * * * * * * * * * * * * * * * * */
/* set color for 3279 terminal                   */
/* * * * * * * * * * * * * * * * * * * * * * * * */
SC = "SET COLOR"
SC "ARROW    PINK"        ; SC "CMDLINE  RED"
SC "CURLINE  WHITE  REV"  ; SC "FILEAREA TURQ   REV"
SC "IDLINE   BLUE   REV"  ; SC "MSGLINE  RED    BLINK"
SC "PENDING  WHITE  REV"  ; SC "PREFIX   YELLOW"
SC "SCALE    GREEN  REV"  ; SC "SHADOW   YELLOW BLINK"
SC "STATAREA PINK   REV"  ; SC "TABLINE  RED"
SC "TOFEOF   RED    REV"
/* * * * * * * * * * * * * * * * * * * * * * * * */
/* set TRUNC and SERIAL for special files        */
/* * * * * * * * * * * * * * * * * * * * * * * * */
"EXTRACT /RECFM /FTYPE"
if (ftype.1='DIR-UPDT') | (ftype.1='DIRECT')
   then SET TRUNC 72
   "EXTRACT /TRUNC/"
if (recfm.1='F') & (trunc.1<=72) then SET SERIAL ALL
   return
ERROR:
   "SOS ALARM"
   "MSG" "Unexpected return code" rc "from line" sigl,
       "of XEDIT profile"
   return
```

Figure 77. PROFILE XEDIT

Reading 2 continues in Chapter 8, “Control,” on page 129.

Menus Using XEDIT

Reading 3

XEDIT can be used with REXX to generate full-screen menus. A short example of a full-screen menu is shown in Figure 79 on page 127. It shows the user the name of the last file edited, lets the user select this file or another, and then calls XEDIT on the selected file. This example is presented here for the concept only, and explanations of the technical details are not given. Please refer to the *z/VM: XEDIT Commands and Macros Reference* for information on the XEDIT subcommands and macros.

The TESTMENU program, following, calls XEDIT using SAMPMENU as a profile. To try this, create the TESTMENU program and then enter `testmenu` on the CMS command line.

```
/* TESTMENU EXEC                                        */
/* sample exec to show use of an XEDIT full screen menu */
"XEDIT" lastfile edited "(PROF" sampmenu
```

Figure 78. TESTMENU EXEC

```
/* SAMPMENU XEDIT                                                  */
/* Sample XEDIT full screen menu                                   */
/* First set up control characters needed for the screen           */
"COMMAND SET CTLCHAR % ESCAPE"
"COMMAND SET CTLCHAR @ PROTECT RED HIGH"
"COMMAND SET CTLCHAR ¢ PROTECT YELLOW NOHIGH"
"COMMAND SET CTLCHAR ! PROTECT BLUE NOHIGH"
"COMMAND SET CTLCHAR $ NOPROTECT TURQ HIGH"
"COMMAND SET CTLCHAR & PROTECT PINK REV NOHIGH"
/* Get old file ID and screen length */
':1'
"COMMAND EXTRACT /CURLINE/LSCREEN"
parse var curline.3 oldname oldtype oldmode .
"COMMAND SET MSGLINE ON" LSCREEN.1-2 "2 OVERLAY"
message = ""
/* Loop, reading the user response.  If ENTER, leave the loop      */
do forever
  call display_screen               /* display the current screen  */
  ADDRESS CMS 'MAKEBUF'
  "COMMAND READ NOCHANGE TAG"       /* allow user input, read it   */
  do queued()                       /* process stacked lines       */
    pull key line column string
    select
      when key="RES"                /* reserved line input?        */
        then select                 /* yes, reset file ID items    */
              when line = 8 then oldname = string
              when line = 10 then oldtype = string
              when line = 12 then oldmode = string
            end
      when key="CMD" then line column string /* commands go to host*/
      when key="ETK" then nop
      when key="PFK"                         /* PF key pressed?    */
        then if line=3 | line=15             /* yes, 3 or 15?      */
              then do                        /* yes,               */
                      ADDRESS CMS 'DROPBUF'  /*    clear stack     */
                      ADDRESS CMS 'MAKEBUF'  /*    and quit        */
                      exit
                    end
        else message = "Unsupported PF key"  /* wrong PF key used   */
      otherwise message = "Unsupported function" /* unknown func.  */
    end
  end
  if (message = "") & (words(oldname oldtype)=2) then leave
end
```

Figure 79. SAMPMENU XEDIT (Part 1 of 2)

```
                                             /* continued...       */
/* replace the last file edited with the new file to be edited,     */
/* stack the XEDIT command, and quit                                */
"REPLACE" oldname oldtype oldmode
push "XEDIT" oldname oldtype oldmode
"COMMAND FILE"
exit
/* routine to display the screen                                    */
display_screen:
  "SET RESERVED 1  NOH"
  "SET RESERVED 2  NOH" '%@            ***%&',
                center('Sample XEDIT full screen menu',35),
                '%@***                 '
  "SET RESERVED 3  NOH"
  "SET RESERVED 4  NOH"
  "SET RESERVED 5  NOH" '%¢  The following file was the last one',
                'edited.  Press enter to'
  "SET RESERVED 6  NOH" '%¢  edit the same file, or key in a new file',
                'ID and press enter.'
  "SET RESERVED 7  NOH"
  "SET RESERVED 8  NOH" '%!        File name:  %$'left(oldname,8)'%¢ '
  "SET RESERVED 9  NOH"
  "SET RESERVED 10 NOH" '%!        File type:  %$'left(oldtype,8)'%¢ '
  "SET RESERVED 11 NOH"
  "SET RESERVED 12 NOH" '%!        File mode:  %$'left(oldmode,2)'%¢ '
  do i = 13 to lscreen.1-2
    "SET RESERVED" i "NOH"   /* reserve the rest of the screen */
  end
  if message ¬= ""
  then do
  "EMSG" message
  message=''
  end
  "CURSOR SCREEN 8 23"
  return
```

Figure 79. SAMPMENU XEDIT (Part 2 of 2)

Reading 3 continues in Chapter 8, “Control,” on page 129.

Chapter 8. Control

A program can be:

- A single list of instructions
- A number of short lists connected by instructions indicating which list is to be executed next.

In this chapter we discuss how you can steer a course from one short list of instructions to another.

The chapter is divided into five sections, one for each of the maneuvers that you might want to accomplish. They are:

Selection
To tell the language processor to select for execution one of a number of lists of instructions, use the IF instruction or the SELECT instruction.

Loops
To tell the language processor to repeat a list of instructions, either for a specified number of times or so long as some condition is satisfied, use the DO instruction.

EXIT
To tell the language processor to finish executing your program, use the EXIT instruction.

Calls to subroutines
To tell the language processor to execute a subroutine, then return and execute the next sequential instruction, use the CALL instruction. Subroutines usually perform a separate, well-defined task; and they can be called from more than one place in the main program.

Jumps
To tell the language processor to continue from a different point in the same file, use the SIGNAL instruction.

Note: Some languages allow GOTO to transfer control to any instruction in a program. In practice it was found that this permitted too many programming errors and thus, in modern languages the use of GOTO is restricted. In REXX, the nearest equivalent to GOTO is SIGNAL. Never use SIGNAL for constructing loops; always use DO.

Selection

To tell the language processor how to decide which instructions are to be executed next, you can use the IF instruction or the SELECT instruction.

In this chapter:

Reading 1 immediately following describes:

- The IF instruction and its keywords THEN and ELSE
  - How to specify a group of instructions as the object of a THEN or ELSE keyword
  - How to avoid the *dangling* ELSE
- The SELECT instruction and its keywords WHEN, THEN, OTHERWISE and END
- The NOP instruction.

Reading 2 skips this section.

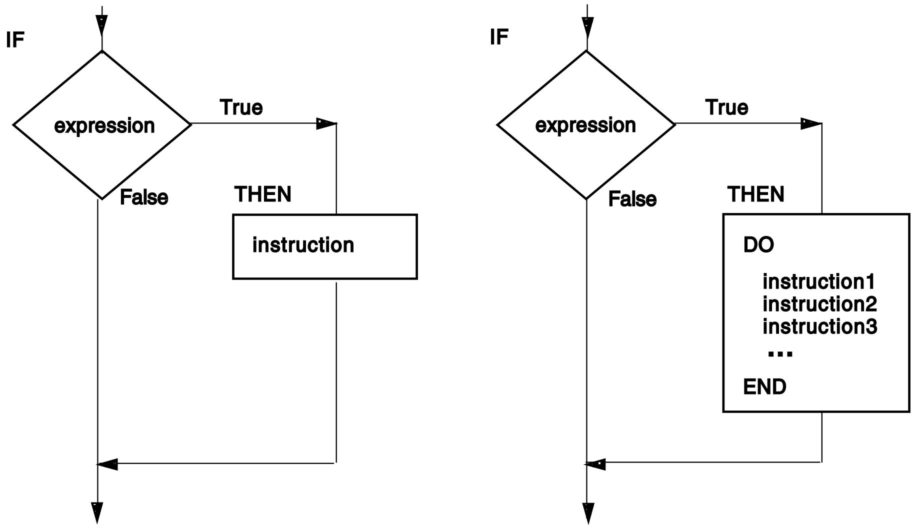
- Continue **Reading 2** in “Loops” on page 138.

Reading 3 skips this section.

- Continue **Reading 3** in “Loops” on page 138.

The IF Instruction

Reading 1



To tell the language processor how to make a decision about a single instruction use:

```
IF expression
THEN instruction
```

The language processor will execute *instruction* only if *expression* is true. For example:

```
if answer = "YES"
then say "OK!"
```

The SAY instruction will be executed, only if ANSWER has the value YES.

To tell the language processor to execute a group of instructions use:

```
DO
   instruction1
   instruction2
   instruction3

⋮
END
```

This form of the DO instruction and the END keyword associated with it tell the language processor to treat the enclosed instructions as a single instruction. You should indent the enclosed instructions three spaces to the right. This will help a person reading the program to see that they belong together.

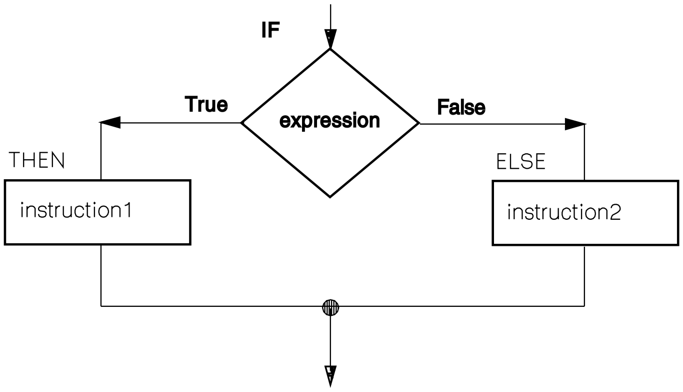
For example:

```
if answer = "YES"
then do
   say "OK.  Please enter the file name and file type",
       "of your input file"
   pull fn ft .
   "STATE" fn ft           /* A CMS command to verify     */
                           /* that file exists.  (It      */
                           /* returns zero if it does.)   */
   if rc = 0 then say "Processing" fn ft
   ...
end
say "What next?"
```

If ANSWER is equal to YES, all the instructions will be executed; if not, only the last instruction will be executed.

The ELSE Keyword

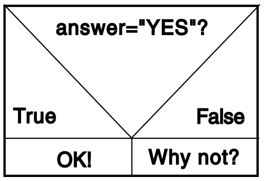
The ELSE keyword looks like this when represented in a flowchart:



When you want the language processor to select from one of two possible instructions use:

```
IF expression
THEN instruction1
ELSE instruction2
```

The language processor will execute *instruction*2 only if *expression* is false. For example, if you wanted:



you could code:

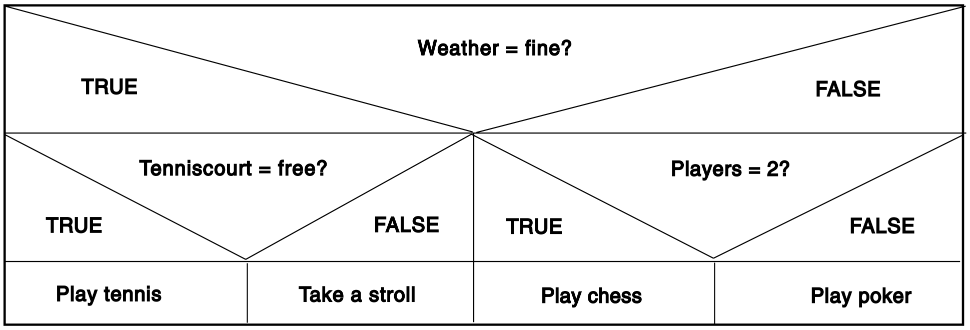
```
if answer = "YES"
then say "OK!"
else say "Why not?"
```

The language processor will display OK! if ANSWER has the value YES; but display Why not? if ANSWER does not have the value YES.

As before, when selecting a list of instructions, you must use DO ... END to mark the beginning and end of the list.

```
if answer = "YES"
then say "OK!"
else do
   say "Why not?"
   pull excuse
   if pos("SORRY",excuse) ¬= 0  /* The REXX function    */
                                /* POS( ) returns '0'   */
                                /* if 'SORRY' does not  */
                                /* appear in EXCUSE     */
                                /* (see page The POS( ) Function on page 72).*/
   then say "I see"
   else say "I just don't understand you"
end
```

More complicated situations can be handled using a series of IFs. The next chart shows two successive decisions leading to one of four possible outcomes.



The best way to code this is:

```
if weather = fine
then do
   if tenniscourt = free
   then say "Shall we play tennis?"
   else say "Shall we take a stroll?"
end
else do
   if players = 2
   then say "Shall we play chess?"
   else say "Shall we play poker?"
end
```

As before, indenting the secondary decisions to the right makes it easier for someone reading the program to see the structure of the program. If you look carefully, you can see that the preceding program has the same structure as the chart above.

The Dangling ELSE

The DO ...; ...; ENDs also help the language processor to keep the ELSEs tied to the right IFs. Look at this fragment:

```
/* The dangling ELSE */
/*     --------      */
if weather = fine
then
    if tenniscourt = free
    then say "Shall we play tennis?"
    ...
  else say "Shall we take our raincoats?"
  /* The language processor will take this ELSE to belong    */
  /* to the nearest preceding IF, but a person        */
  /* reading the program might easily assume that it  */
  /* belonged to the first IF.                        */
```

Avoid writing code like the preceding example. It is too error-prone. Programs that have IFs within IFs should use DO ... END. This example pairs THEN DO with END and THEN with ELSE.

```
if ...
   then do
      if ...
         then do
            ...
            ...
         end
         else do
            ...
            ...
         end
   end
   else ...
```

Test Yourself...

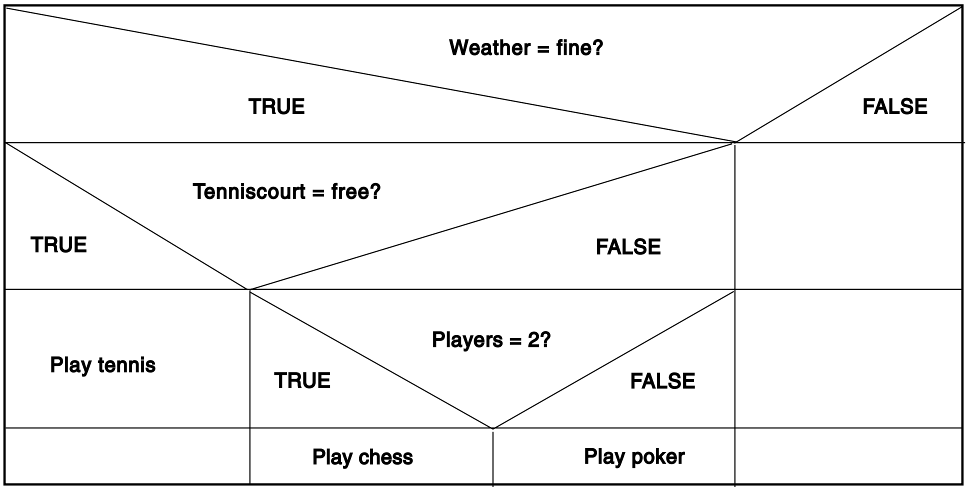
What will the following program do?

```
/* WHATODO EXEC */

/* input data   */
weather = "FINE"
tenniscourt = "FREE"
players = 2
/* example of a program that does not use DO ... END    */
/* as recommended previously                            */
trace results
if weather = fine
then
   if tenniscourt = free
   then say "Shall we play tennis?"
   /* else say "Shall we take a stroll?" DELETED       */
else
   if players = 2
   then say "Shall we play chess?"
   else say "Shall we play poker?"
```

Try it! The REXX instruction TRACE Results will help you to see what is happening.

Answers:

Do not be deceived by the indentation! The ELSE is associated with the nearest preceding IF. The following table can help you determine what happens when certain values are given to `weather`, `tenniscourt`, and `players`.

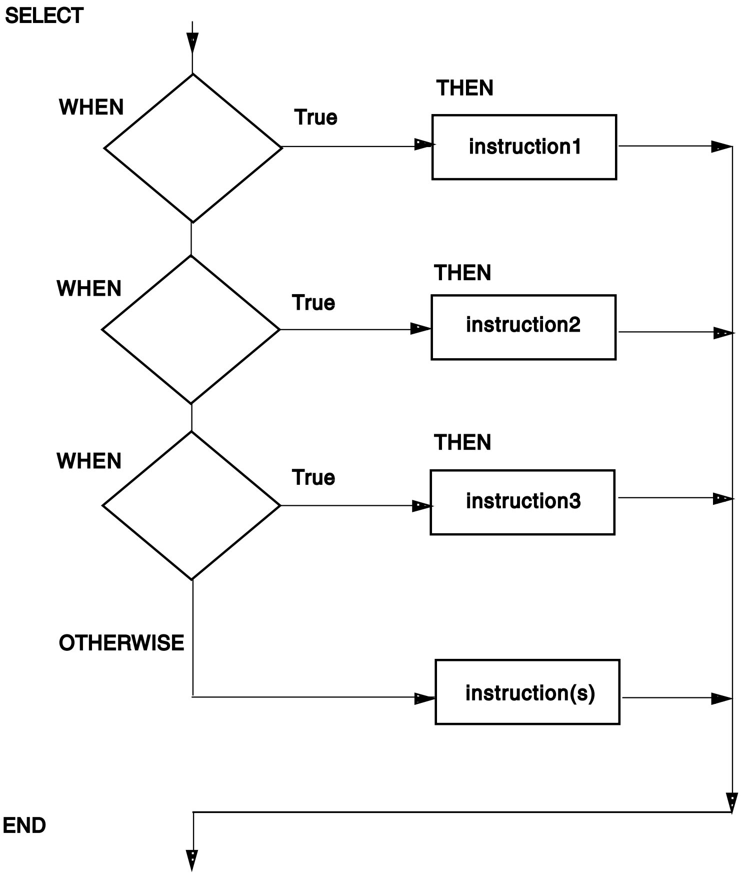


For the values given in the WHATODO EXEC, the following will result:

```
whatodo
    10 *-* if weather = fine
       >>>   "1"
    11 *-*  then
    12 *-*  if tenniscourt = free
       >>>   "1"
    13 *-*   then
       *-*   say "Shall we play tennis?"
       >>>     "Shall we play tennis?"
Shall we play tennis?
    14 *-*   /* else say "Shall we take a stroll?" DELETED       */
Ready;
```

The SELECT Instruction

The SELECT instruction looks like this when represented in a flowchart:



If you want the language processor to select one of any number of instructions, use:

```
SELECT
   WHEN expression1 THEN instruction1
   WHEN expression2 THEN instruction2
   WHEN expression3 THEN instruction3

⋮
    OTHERWISE
       instruction
       instruction
       instruction

⋮
END
```

- If *expression*1 is true, *instruction*1 is executed. After this, processing continues with the instruction following the `END`.
- But if *expression*1 is false, *expression*2 is tested. Then, if *expression*2 is true, *instruction*2 is executed and processing continues with the instruction following the `END`.
- If all of *expression*1, *expression*2, and so forth, are *false*, an OTHERWISE keyword **must** be present. Then,
- Processing continues with the instruction following the `OTHERWISE`.

As before, to tell the language processor to execute a list of instructions following the THEN keyword, use:

```
DO
   instruction1
   instruction2
   instruction3

⋮
END
```

This form of the DO instruction and the END keyword associated with it tell the language processor to treat the enclosed instructions as a single instruction.

A DO; ... ; END; group is not required after the OTHERWISE keyword.

Example

Here is a short program that uses SELECT:

```
/* CENSUS EXEC                                          */
/* This program requests the user to provide a person's */
/* age and sex.  In reply, it displays a person's       */
/* status.  Persons under the age of 5 are BABIES.      */
/* Those aged 5 through 12 are BOYS or GIRLS.           */
/* Those aged 13 through 19 are TEENAGERS.              */
/* The rest are MEN or WOMEN.                           */
/*------------------------------------------------------*/
/* Get input from user                                  */
/*------------------------------------------------------*/
do until datatype(age,NUMBER) & age >= 0
   say "What is the person's age?"
   pull age
end
do until sex = "M" | sex = "F"
   say "What is the person's sex (M or F)?"
   pull sex
end
/*------------------------------------------------------*/
/* COMPUTE STATUS                                       */
/*                                                      */
/* Input:                                               */
/* AGE     Assumed to be 0 or a positive number.        */
/* SEX     "M" is taken to be male;                     */
/*         anything else is taken to be female.         */
/*                                                      */
/* Result:                                              */
/* STATUS  Possible values: BABY, BOY, GIRL, TEENAGER   */
/*         MAN, WOMAN.                                  */
/*------------------------------------------------------*/
Select
   when age < 5 then status = "BABY"
   when age < 13 then do
      if sex = "M"
      then status = "BOY"
      else status = "GIRL"
      end
   when age < 20 then status = "TEENAGER"
   otherwise
      if sex = "M"
      then status = "MAN"
      else status = "WOMAN"
end
say "This person should be counted as a" status
```

Figure 80. CENSUS EXEC

Each SELECT has a corresponding END. To make your program easier for people to read, you should indent everything between the SELECT and the END three spaces to the right.

The NOP Instruction

A THEN or ELSE keyword *must* be followed by an instruction. In cases where you intend that nothing should be done, use a NOP (no operation) instruction.

Here are two examples:

```
/* PILOT EXEC                                           */
/* Example:  steering a course                          */
Say "Where is the harbor?"
pull where
select
   when where = "AHEAD" then nop
   when where = "PORT BOW" then say "Turn left"
   when where = "STARBOARD BOW" then say "Turn right"
   otherwise say "Not understood"
end
```

Figure 81. PILOT EXEC

```
/* TRUCKER EXEC                                         */
/* Example:  using NOP to simplify the presentation of  */
/* a set of conditions.                                 */
If gas = "FULL" & oil = "SAFE" & window = "CLEAN"
then nop
else say "Find a gas station!"
```

Figure 82. TRUCKER EXEC

Test Yourself...

1. Write a program that asks the user to enter two words (on the same line) and computes whether:
   - The words are the same (or numerically equal)
   - The first word is higher
   - The second word is higher.

   The comparison must ignore differences in case. For example, *A* will count as equal to *a*.
2. “Thirty days hath September, April, June, and November; all the rest have thirty-one, excepting February alone ....”

   Write a program that asks the user to specify the month as a number between 1 and 12 and gives the number of days in the month in reply. For month 2, the reply can be 28 or 29.

Answers:

1. A possible answer is:

```
/* COMPARE1 EXEC */
/* This program requests the user to supply two     */
/* words and says which is higher.                  */
say "Enter two words"
pull word1 word2 .
select
   when word1 = word2
   then say "The words are the same",
            "or numerically equal"
   when word1 > word2
   then say "The first word is higher"
   otherwise
   say "The second word is higher"
end
```

An alternative answer is:

```
/* COMPARE2 EXEC */
/* This program requests the user to supply two     */
/* words and says which is higher.                  */
say "Enter two words"
pull word1 word2 .
if word1 = word2
then say "The words are the same",
            "or numerically equal"
else do
   if word1 > word2
   then say "The first word is higher"
   else say "The second word is higher"
end
```

Some people would consider the first solution better, because it is slightly easier to understand.

2. To say how many days in the month:

```
/* CALENDAR EXEC */
/* This program requests the user to enter a whole  */
/* number from 1 through 12 and replies giving the  */
/* number of days in that month.                    */
/*--------------------------------------------------*/
/* Get input from user                              */
/*--------------------------------------------------*/
do until datatype(month,WHOLE),
      & month >= 1 & month <= 12
   say "Enter the month as a number from 1 through 12"
   pull month
end
/*--------------------------------------------------*/
/* Compute days in 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
```

Reading 1 continues in “Loops.”

Loops

A *loop* is a group of instructions that may have to be executed more than one time.

In this section:

Reading 1 immediately following, describes:

- Repetitive DO loops
  - Control variables
  - The BY expression
- Conditional DO loops
  - DO FOREVER and LEAVE instructions
  - DO WHILE instruction
  - DO UNTIL instruction.

Reading 2 on page 147, describes:

- Compound DO instructions
- Leaving a specified loop.

Reading 3 on page 148, describes:

- The ITERATE instruction.

Simple Repetitive Loops

Reading 1

To repeat a loop a number of times, use:

```
DO exprr
   instruction1
   instruction2
   instruction3
⋮
END
```

where:

exprr (the **expr**ession for **r**epetitor) gives a whole number, which is the number of times the loop is to be executed.

To make your program easier for people to read, you should indent the instructions between the DO and the END three spaces to the right.

Here are two examples of repetitive loops, see Figure 83 and Figure 84 on page 140. The first is about preparing for a meeting. Each person attending will require three documents.

The program that prints the documents is:

```
/* HANDOUTS EXEC                                          */
/* To print documents for a meeting:  for each person,    */
/* the agenda, minutes and accounts are printed one       */
/* after the other.  Between sets, the CP output          */
/* header appears.                                        */
"CP SPOOL PRINT CONT"             /* See the following note  */
do 5
   "PRINT AGENDA DOCUMENT"
   "PRINT MINUTES DOCUMENT"
   "PRINT ACCOUNTS DOCUMENT"
   "SPOOL PRINT CLOSE"         /* See the following note  */
end
"CP SPOOL PRINT NOCONT"
```

Figure 83. HANDOUTS EXEC

The program in Figure 83 prints five sets of documents.

Note: The following command, which is used in the HANDOUTS EXEC, tells CP to collect any files that it is asked to PRINT into a batch.

```
SPOOL PRINT CONT
```

The batch accumulates until the command SPOOL PRINT CLOSE is issued. SPOOL PRINT CLOSE causes the batch to be printed, but leaves CONT in effect. See the *z/VM: CP Commands and Utilities Reference* for details.

In this next program, the instruction between the DO and the END will be executed `HEIGHT` times.

```
/* RECTANGL EXEC                                        */
/* The user is asked to specify the height of a         */
/* rectangle (within certain limits).  The rectangle    */
/* is then displayed on the screen.                     */
say "Enter the height of the rectangle",
    " (a whole number between 3 and 15)."
pull height
select
   when  ¬datatype(height,WHOLE) then say "Rubbish!"
   when  height < 3 then say "Too small!"
   when  height > 15 then say "Too big!"
   otherwise
                                /* draw rectangle       */
   do height
      say copies("*",2*height)
   end
   say "What a pretty box!"
end
```

Figure 84. RECTANGL EXEC

Using a Control Variable

To number each pass through the loop, in such a way that you can use that number as a variable in your program, use:

```
DO name = expri [TO exprt]
   instruction1
   instruction2
   instruction3
⋮
END
```

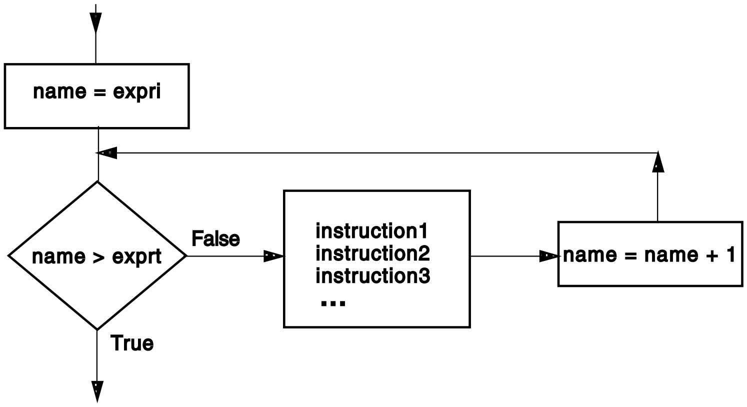
where:

name is the *control variable*. You can use it in the body of the loop. Its value is changed (in this example, increased by 1) each time you pass through the loop.

expri (the **expr**ession for the **i**nitial value) gives the value you want the control variable to have the first time through the loop.

exprt (the **expr**ession for the **T**O value) gives the value you want the control variable to have the last time through the loop.

The next diagram shows exactly how the control variable is changed, and how the decision to leave the loop is made.



You can use the control variable to compute something different each time through the loop. In this example, the control variable is called COUNT, and it computes the width of each row of stars.

```
/* TRIANGLE EXEC                                       */
/* This program displays a triangle on the screen.     */
/* The user is asked to specify the height of the      */
/* triangle.                                           */
say "Enter the height of the triangle",
    " (a whole number between 3 and 15)."
pull height
select
   when  ¬datatype(height,WHOLE) then say "Rubbish!"
   when  height < 3 then say "Too small!"
   when  height > 15 then say "Too big!"
   otherwise
                                /* draw triangle       */
   do count = 1 to height
      say copies("*",2*count - 1)
   end
   say "What an ugly triangle!"
end
```

Figure 85. TRIANGLE EXEC

After you have left the loop, you can still refer to the control variable. It will always exceed the value of the TO expression (*exprt*).

The BY Expression

So far, we have assumed that the control variable will be incremented by 1 each time through the loop. This is the default. To specify some other value, write:

```
►►──DO──name──=──expri──┬──────────┬──┬──────────┬──────────────────────►◄
                        └─BYexprb──┘  └─TOexprt──┘
```

where:

exprb (the **expr**ession for **B**Y) gives the number that is to be added to *name* at the bottom of the loop.

Test Yourself...

1. Using the flowchart on page 140, you should be able to predict what this program will “say”.

```
/* 1MORE EXEC */

/* Example:  use of a control variable */
do digit = 1 to 3
   say digit
end
say "Now we have reached" digit
```

2. What about this program?

```
/* 2LESS EXEC */

/* Example:  use of a control variable */
do count = 10 by -2 to 6
   say count
end
say "Now we have reached" count
```

3. How many lines will this program “say”?

```
/* 3HUP EXEC */

/* Example:  use of a control variable */
do j = 10 to 8
   say "Hup! Hup! Hup!"
end
```

4. How many lines will this program “say”?

```
/* 4NOW EXEC */

/* Example:  use of a control variable */
do NOW = 1
   if NOW = 9 then exit
   say NOW
end
```

Answers:

1. The control variable is changed at the bottom of the loop. The test for leaving is made after this. So the control variable will be beyond the limit value.

   1
   2
   3
   Now we have reached 4

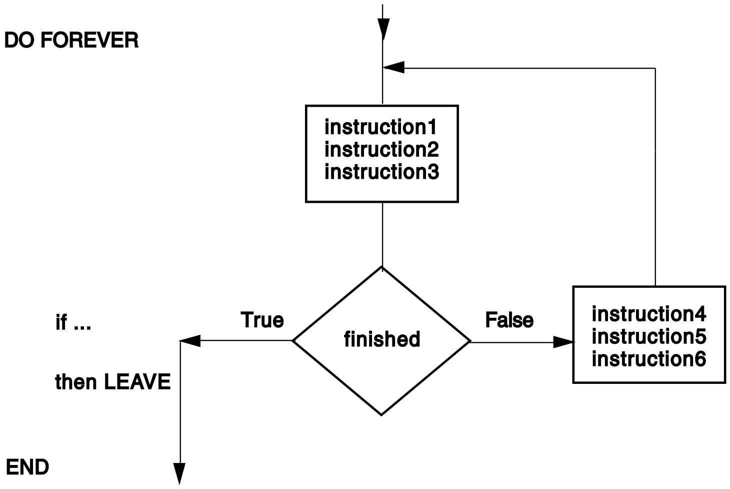
2. If `exprb` is negative, count down:

   10
   8
   6
   Now we have reached 4

3. None (10 already exceeds 8).
4. Eight (on the ninth pass, the EXIT instruction ends the program before the SAY instruction is reached).

Conditional Loops: The LEAVE Instruction

Conditional loops continue to be executed so long as some condition is satisfied. The simplest way to code these loops is to use DO FOREVER and LEAVE.



The instruction

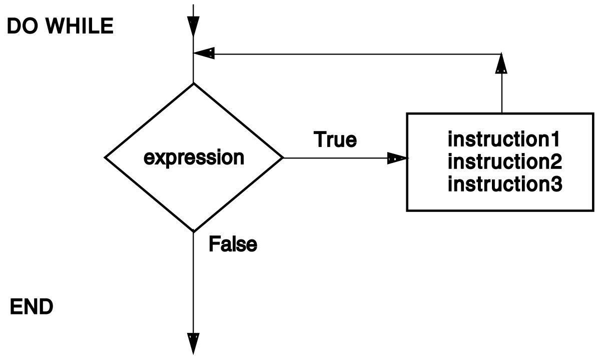
```
LEAVE
```

causes processing to continue with the instruction following the END keyword. For example, the SUM EXEC in Figure 86 will continue executing as long as the user enters a number. If you do not enter a number, the `LEAVE` instruction is executed and processing continues with the `SAY` instruction.

```
/* SUM EXEC                                             */
/* This program adds up the numbers that the user is    */
/* invited to enter.  When the user enters something    */
/* that is not a number, a message is displayed and     */
/* the program ends                                     */
total = 0
do forever
   say "Enter a number"
   pull n
   if ¬datatype(n,NUMBER) then leave
   total = total + n
   say "Total = " total
end
say "'"n"' is not a number.  Returning to CMS."
```

Figure 86. SUM EXEC

Conditional Loops: The DO WHILE Instruction



To build a conditional loop with the test at the top, use:

```
DO WHILE exprw
   instruction1
   instruction2
   instruction3
END
```

where:

exprw (**expr**ession for **w**hile) is an expression that, when evaluated, must give a result of `0` or `1`.

In some cases, it is easiest to design with the test at the top. If so, you should use the DO WHILE instruction.

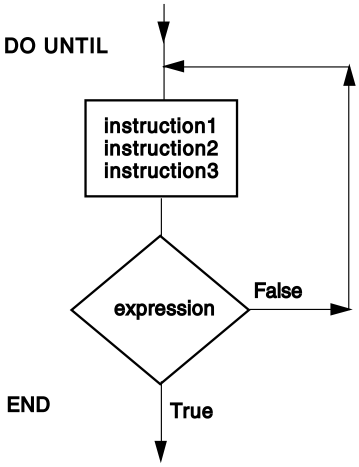
These two fragments will produce the same results.

```
DO WHILE ¬ finished
   instruction1
   instruction2
   instruction3
END
```

or

```
DO FOREVER
   if finished then LEAVE
   instruction1
   instruction2
   instruction3
END
```

Conditional Loops: The DO UNTIL Instruction



To build a conditional loop with the test at the bottom, use:

```
DO UNTIL expru
   instruction1
   instruction2
   instruction3
:
END
```

where:

expru (**expr**ession for **u**ntil) is an expression that, when evaluated, must give a result of `0` or `1`.

Using the DO UNTIL loop allows all instructions to be executed at least once. In some cases, it is easiest to design with the test at the bottom. If so, you should use the DO UNTIL instruction.

These two fragments will produce the same results.

```
DO UNTIL finished
   instruction1
   instruction2
   instruction3
END
```

or

```
DO FOREVER
   instruction1
   instruction2
   instruction3
   if finished then LEAVE
END
```

Conditional Loops: The Choice

There are three kinds of conditional loops:

1. The decision is made *before* processing starts. For example, this program will fill BATH. But if BATH is already full, the body of the loop will not be executed and no water will be added.

   ```
   DO WHILE bath < full
      bath = bath + bucket
   end
   ```

2. The decision is made *after* the first pass through the loop and again after every subsequent pass. For instance, requesting valid data from a user.

   ```
   DO UNTIL datatype(input,NUMBER)
      say "Enter a number"
      pull input
   end
   ```

3. The decision is made *during* each pass. For instance, the decision to leave might depend on information obtained during the loop.

   ```
   DO FOREVER
      say "Enter an item of data.  When there is",
          " no more data, enter QUIT"
      pull answer
      if answer = "QUIT" then leave
         ...   /* process the data */
   end
   ```

   Later, we shall see that a program that reads data from a file should also be programmed using DO FOREVER and LEAVE.

Note: Be careful about the condition for repeating the loop. For WHILE, the condition must be TRUE; for UNTIL, it must be FALSE.

Test Yourself...

1. What kind of DO instruction would you use to code the sequence:

   ```
   Job done?
       instruction1
       instruction2
       instruction3
   Job done?
       instruction1
       instruction2
       instruction3
   Job done?
    ...
   Job done?
   ```

2. What kind of DO instruction would you use to code the sequence:

   ```

       instruction1
       instruction2
       instruction3
   Job done?
       instruction1
       instruction2
       instruction3
   Job done?
    ...
   Job done?
   ```

Answers:

1. DO WHILE job ¬= done (The first operation is to test “Is job done?”)
2. DO UNTIL job = done (The first operation is to execute the list of instructions.)

Reading 1 continues in “The EXIT Instruction” on page 149.

Compound DO Instructions

Reading 2

You can combine one repetitive phrase and one conditional phrase in a single DO instruction. You should know where in the loop the counters are updated and where the tests for leaving the loop will be made. This is explained in a diagram in your *z/VM: REXX/VM Reference*. (You can find it under the description of the DO instruction.)

Compound DO instructions can do a lot of useful work. This next example shows how a simplified version of the POS( ) function might be implemented as a REXX function.

```
/* POSN EXEC                                           */
/* Example:  the POSN( ) function is similar to the     */
/* POS( ), except that the third argument ("start")     */
/* is not allowed                                       */
if arg() ¬= 2
then return               /* wrong number of arguments  */
if arg(1,omitted) | arg(2,omitted)
then return               /* argument was omitted       */
parse arg needle,haystack
last = length(haystack),  /* compute the rightmost      */
       -length(needle)+1  /* position that needle could */
                          /* be found in                */
do result = 1 to last,    /* Search for needle          */
   until substr(haystack,result,length(needle)) = needle
end
if result > last then result = 0
return result
```

Figure 87. POSN EXEC

Leaving a Specified Loop

Sometimes a program is constructed of loops within loops. When you leave a loop, you would like to tell the language processor which loop you want to leave. To do this, give a DO loop a name (that is, specify a control variable in the DO instruction). If the loop does not contain a control variable already, invent one. For example

```
DO outer = 1
   ...
   ...
END
```

is the same, for all practical purposes, as DO FOREVER. In this example, `outer` is the control variable for the loop.

Now, to leave a specific loop, put the name of its control variable after the keyword LEAVE. For example:

```
DO outer = 1
    ...
   do until datatype(answer,WHOLE)
      say "Enter a number. ",
          "When you have no more data, enter a blank line"
      pull answer
      if answer = "" then leave outer
   end
    ...
   /* process answer */
end
/* come here when there is no more data */
```

Reading 2 continues in “Subroutines” on page 150.

The ITERATE Instruction

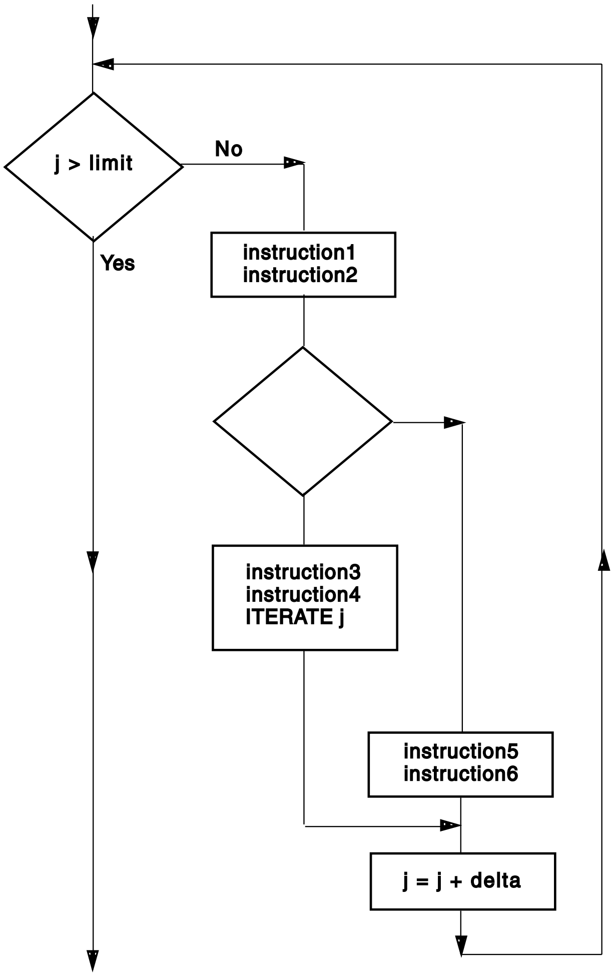
Reading 3

To bypass all remaining instructions in the loop and test the ending conditions, use the ITERATE instruction. Like LEAVE, ITERATE can be introduced by a THEN or ELSE keyword. But, instead of leaving the loop altogether, the language processor proceeds with the operations usually done at the bottom of the loop. If an UNTIL condition has been specified, it is tested; if a control variable has been specified, it is incremented and tested; and if a WHILE condition has been specified, it is tested.

If tests indicate that the loop is still active, typical processing then continues from the top of the loop.

For example:

```
DO j = 1 to limit by delta
   instruction1
   instruction2
   if ...
   then do
      instruction3
      instruction4
      ITERATE j
   end
   instruction5
   instruction6
END;
```



Reading 3 continues in “Jumps” on page 158.

The EXIT Instruction

In this section:

Reading 1 is the entire “EXIT Instruction” section, describing:

- How to leave your program by using the EXIT instruction.

Reading 1

To tell the language processor to leave your exec use:

```
►►──EXIT──┬────────────┬──────────────────────────────────────────►◄
          └─expression─┘
```

If your exec was started by typing its name on the command line:

- `EXIT` will take you back to CMS.
- *expression* must result in a whole number, which CMS will display as a return code in the Ready message.

For example:

```
/* FADE EXEC                               */
/* Example:  using EXIT with a return code */
say "Returning to CMS"
exit 22
```

Figure 88. FADE EXEC

When run, the program in Figure 88 will cause this to be displayed:

```
fade
Returning to CMS
Ready(00022);
```

Reading 1 continues in “Subroutines.”

Subroutines

In this section:

Reading 1 immediately following, describes:

- The idea of a subroutine
- The CALL instruction
- How to obtain the arguments passed to a subroutine:
  - Using the ARG( ) function
  - Using the ARG instruction
  - Using the PARSE ARG instruction
- The RETURN instruction.

Reading 2 on page 156, describes:

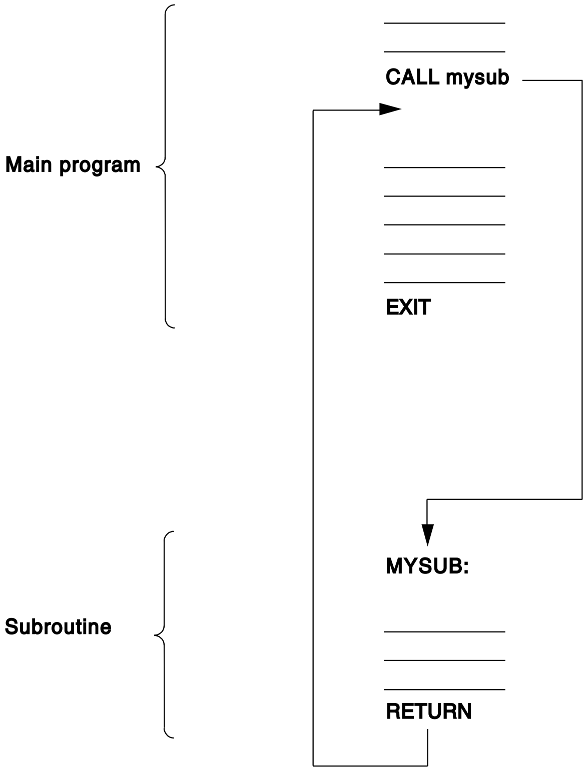
- Subroutines and functions
  - What are the differences
  - What are the similarities
- Parsing the arguments
- External subroutines.

The Idea of a Subroutine

Reading 1

A *subroutine* is a separate piece of code that can be called from more than one place in your main program.

Subroutines can be in the same file as the main program, or they can be in a separate EXEC file. The diagram shows a subroutine that is in the same file as the main program.



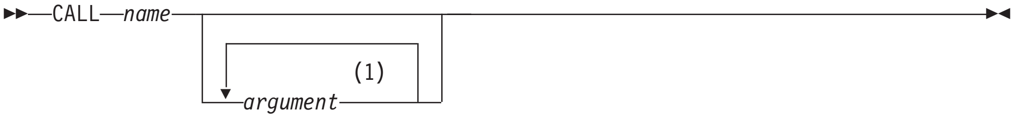
A CALL instruction will cause the language processor to look through your program until it finds the *label* that marks the start of the subroutine. Processing continues from there until the language processor finds a RETURN instruction that causes the language processor to return to the main program.

A subroutine can be called from more than one place in a program. The language processor always returns to the clause following the CALL instruction from which it came.

Each CALL instruction can supply data, called *arguments*, which the subroutine can use when called. In the subroutine, you can find out what data has been supplied by using the ARG( ) function or the ARG instruction.

The CALL Instruction

To direct the language processor to execute a subroutine use:



Notes:

1 A maximum of 20 arguments.

where:

name is the name of the subroutine. The language processor will first search for the corresponding label in your program. A label consists of a symbol followed by a colon (:), for example:

name:

If no such label is found, the language processor looks for a built-in function, exec file, or module file of that name. (To be discussed later, on page 159.)

argument
is an expression. The value of each is computed, and can be obtained in the subroutine by using the ARG( ) function.
    ARG(1) returns the first argument
    ARG(2) returns the second argument
    ...

You can have up to 20 arguments on a CALL instruction.

You can also obtain the arguments by using the ARG or PARSE ARG instructions, discussed later.

For example:

```
/* CHEER EXEC                                           */
/* Example:  calling a subroutine                       */
do 3
  call triple "R"
  call triple "E"
  call triple "X"
  call triple "X"
  say
end
say "R...!"
say "E...!"
say "X...!"
say "X...!"
say
say "REXX!"
exit                             /* end of main program */
/*------------------------------------------------------*/
/* Subroutine to repeat a shout three times             */
/* ========================================             */
/* The first argument is displayed on the screen, three */
/* times on one line, with suitable punctuation.        */
/*------------------------------------------------------*/
TRIPLE:
say arg(1)",  "arg(1)",  "arg(1)"!"
return
```

Figure 89. CHEER EXEC

This is what appears on the screen:

```
cheer
R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!
R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!
R,  R,  R!
E,  E,  E!
X,  X,  X!
X,  X,  X!
R...!
E...!
```

```
X...!
X...!
REXX!
Ready;
```

The EXIT instruction causes a return to CMS. In the program shown in Figure 89, the EXIT instruction stops the main program from running on into the subroutine.

The ARG Instruction

In your subroutine, you may want to refer to an argument many times; if so, it would make your program easier to read if the argument had a memorable name, rather than just ARG(1). To assign the arguments to variables, use the PARSE ARG instruction or the PARSE UPPER ARG instruction.

For example, if you want the results of the four expressions on the call instruction to be assigned FLOUR, BUTTER, SUGAR, and COOKIES, you could write:

```
PARSE ARG flour, butter, sugar, cookies
```

The other form of the instruction, PARSE UPPER ARG, can be shortened to ARG. If you wanted the four arguments to be translated to uppercase you could write:

```
ARG flour, butter, sugar, cookies
```

Notice that, just as there are commas between the expressions in the CALL instruction, so there are commas between the symbols in the PARSE ARG or ARG instruction when it is used in this way.

The RETURN Instruction

The RETURN instruction takes you back to the main routine. Processing continues with the instruction following the CALL. The full form of the instruction is

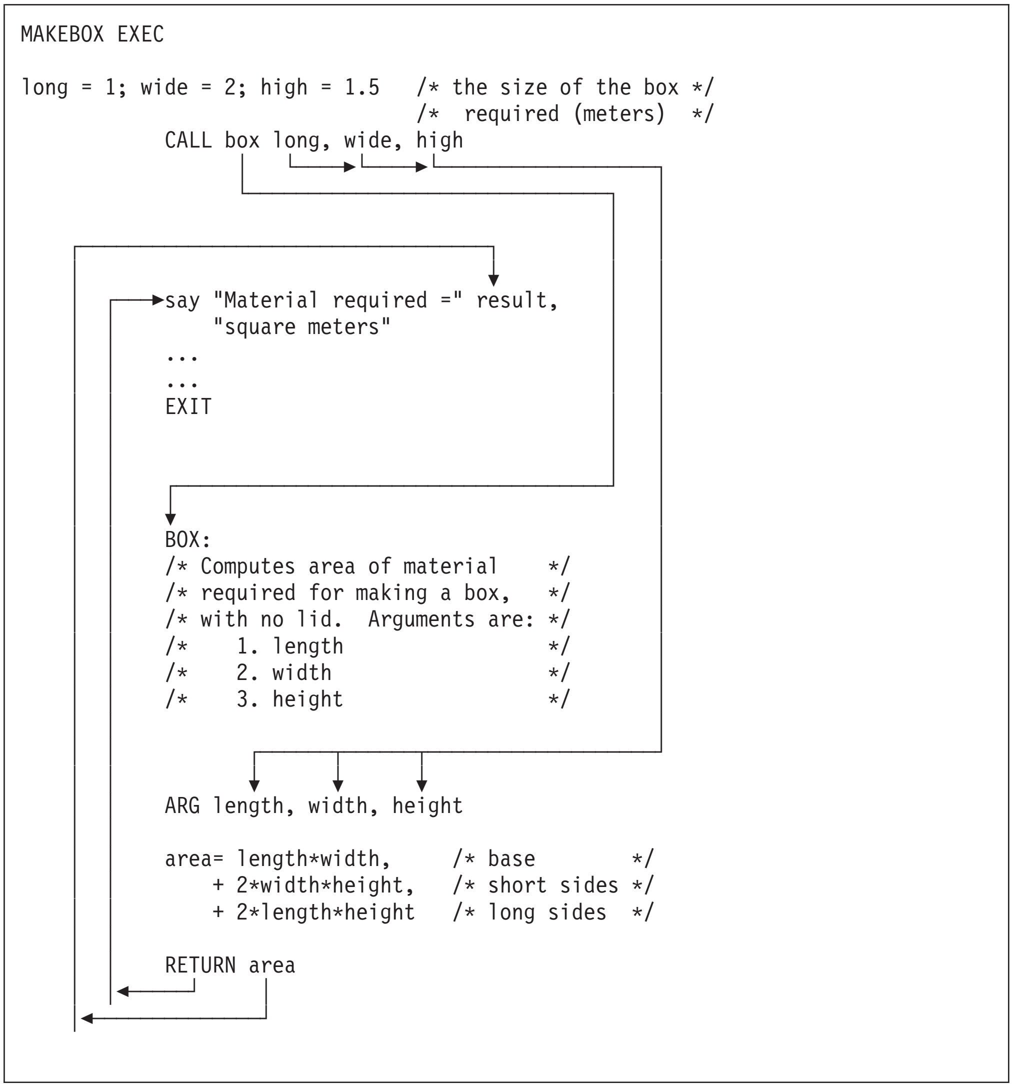
```
►►──RETURN──┬────────────┬──────────────────────────────────────────►◄
            └─expression─┘
```

where, if `expression` is specified, it will be assigned to the REXX special variable, RESULT. (But if `expression` is omitted, RESULT is *dropped*.) That is, RESULT is not assigned a value and thus, when used in an expression, takes on the value of itself, translated to uppercase (RESULT).

The variable RESULT can be used in an expression by the calling program when it resumes.

Example

This example shows how CALL passes arguments to a subroutine; ARG assigns the arguments' values to variables; RETURN assigns a value to RESULT; and the main program uses this data.



```
MAKEBOX EXEC

long = 1; wide = 2; high = 1.5   /* the size of the box */
                                 /*  required (meters)  */
            CALL box long, wide, high

            say "Material required =" result,
                "square meters"
            ...
            ...
            EXIT

            BOX:
            /* Computes area of material    */
            /* required for making a box,   */
            /* with no lid.  Arguments are: */
            /*    1. length                 */
            /*    2. width                  */
            /*    3. height                 */

            ARG length, width, height

            area= length*width,     /* base        */
                + 2*width*height,   /* short sides */
                + 2*length*height   /* long sides  */

            RETURN area
```

When to Leave Out the Arguments

If program variables are referred to by the same names both outside and inside an *internal* routine (a routine that exists in the same file as the CALL instruction), it is not necessary to include them as arguments on the CALL or ARG instructions.

However, not including them could make it more difficult for a person reading your program to understand what your subroutine does. So it will be especially important in this case to give a list of the arguments in the comments that introduce the subroutine.

Test Yourself...

This program simulates a children’s race game, of the kind that used to be played with dice.

Write the subroutine TELL to tell who is winning.

```
/* RACEGAME EXEC */

/* Example of a subroutine: a child's race game         */
a = 0                    /* Arthur starts from zero     */
b = 3                    /* Barry gets a headstart of 3 */
do 15
   a = a + random(1,6)   /* Arthur gets first turn      */
   b = b + random(1,6)   /* Now it is Barry's turn      */
   call tell             /* Who's ahead now             */
end
exit                     /* End of main program         */
```

Copy the main program and your subroutine into an exec file and test your program.

Answer:

A possible solution is:

```
/* RACEGAME EXEC */

/* Example of a subroutine: a child's race game     */
a = 0                    /* Arthur starts from zero */
b = 3                /* Barry gets a headstart of 3 */
do 15
   a = a + random(1,6)   /* Arthur gets first turn  */
   b = b + random(1,6)   /* Now it is Barry's turn  */
   call tell             /* Who's ahead now         */
end
exit                     /* End of main program     */
/*--------------------------------------------------*/
/* Subroutine to display the position               */
/* ==================================               */
/* INPUT: a (Arthur's score)                        */
/*        b (Barry's score)                         */
/* RESULT: displayed on user's screen               */
/*--------------------------------------------------*/
TELL:
values = "Arthur =" a";  Barry =" b"; "
select
   when a > b then say values "Arthur is ahead"
   when b > a then say values "Barry is ahead"
   otherwise say values "Neck and neck!"
end
return
```

In this sample solution, there are no arguments on the CALL instruction. Nevertheless, a person reading the program will still need to know what data the subroutine is using.

A well-designed subroutine will operate on a clearly defined set of data. To make your program more readable, you should define this data in comments at the beginning of the subroutine.

Reading 1 continues in “Jumps” on page 158.

Subroutines and Functions

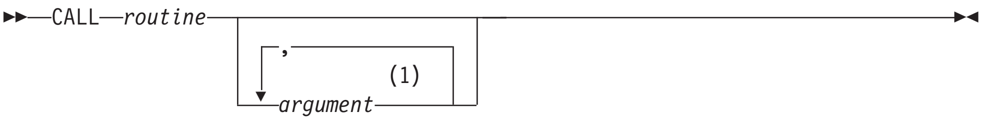
Reading 2

You can write your own subroutines (described earlier) and your own functions. You can also use subroutines and functions written by other people.

What are the differences between subroutines and functions, and what do they have in common?

The **differences** are:

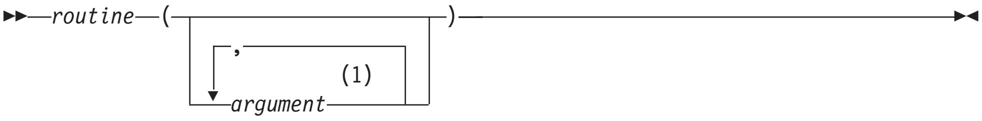
- To call a subroutine, you use a CALL instruction:



Notes:

1 A maximum of 20 arguments.

But to call a function, you use a function call:



Notes:

1 A maximum of 20 arguments.

- A subroutine need not return a result, but a function *must* return a result. In a subroutine, you can write:

```
RETURN
```

But in a function you must at least write:

```
RETURN ""   /* This returns a null string */
```

- A subroutine sets the value of the special variable RESULT. But the result returned by a function is used in the expression where the function call appeared.

The **similarities** are:

- Both use the ARG and PARSE ARG instructions, and the ARG( ) function, for obtaining the values of their arguments.
- Both can be either *internal* (that is, starting with a label in the same file as the CALL instruction or the function call) or *external* (that is, in a different file).
- Both have the same *search order*. When a call to *routine* is recognized, the language processor searches for:
  1. The label *routine*: in the same file
  2. A REXX function called *routine*
  3. An external routine.

  (For full details, see the *z/VM: REXX/VM Reference*.)
- Both, when they are *internal*, can use the PROCEDURE instruction (described in Reading 3, page The PROCEDURE Instruction on page 30).
- Where it is reasonable to do so, functions can be used as subroutines. Subroutines that return a result can be used as functions.

Using a Call of the Other Kind

Where convenient, programs designed as functions can be called as subroutines. And, if they always return a result, programs designed as subroutines can be called as functions.

For example, the subroutine QUIET, which we discussed in Figure 64 on page 107, could be called as a function:

```
if quiet("STATE" fn ft) = 0
then  ...
```

and the POS( ) function could be called as a routine:

```
/* to remove NEEDLEs from haystack */
do forever
   call pos needle,haystack
   if result = 0
   then leave
   else haystack = delstr(haystack,result,length(needle))
end
```

Note: DELSTR( ) is a REXX built-in function. See the *z/VM: REXX/VM Reference* for details.

Parsing the Arguments

Each of the arguments passed by a CALL instruction can be parsed using the PARSE ARG instruction or the ARG instruction. For example, the instruction:

```
CALL words "a string of words",5
```

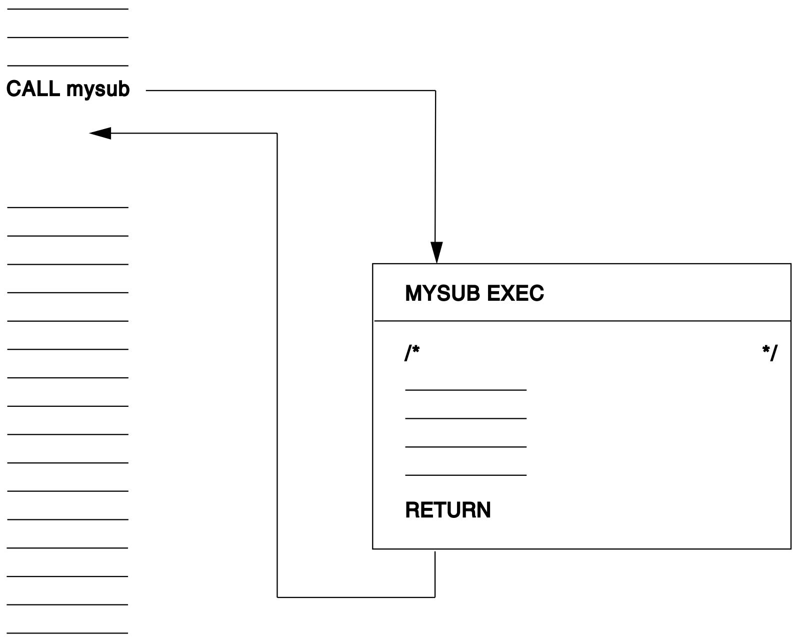
might be parsed using:

```
WORDS:
PARSE ARG first second third fourth rest, number
```

The result would be that:
`FIRST` gets “a”
`SECOND` gets “string”
`THIRD` gets “of”
`FOURTH` gets “words”
`REST` gets “”
`NUMBER` gets “5”

External Subroutines

When we first discussed subroutines, we mentioned only the internal routines. But subroutines can also exist as a separate exec file.



In an *external* routine, the variables belonging to the caller are **not** available to the subroutine. All the data must be formally passed, using arguments on the CALL instruction, and all the data must be returned using the RETURN instruction. (If necessary, the calling routine can PARSE the variable RESULT into a number of variables.)

For more information about sharing variables, see the GLOBALV command in the *z/VM: CMS Commands and Utilities Reference*.

Reading 2 continues in “Jumps.”

Jumps

In this section we discuss instructions that cause the language processor to continue processing at a different point in your program.

In this section:

Reading 1 immediately following, describes:

- Using the SIGNAL instruction for *jumps*.

Reading 2 on page 159, describes:

- How to use the SIGNAL instruction for abnormal changes of control.

Reading 3 on page 159, describes:

- How to set a condition trap.
- How to set a condition trap using the CALL instruction.
- How to use the SIGNAL instruction to set “ON-conditions”.
  – SIGNAL ON FAILURE
  – SIGNAL ON HALT
  – SIGNAL ON NOVALUE
  – SIGNAL ON SYNTAX.
- How to obtain information about a current trapped condition.

The SIGNAL Instruction

Reading 1

The SIGNAL instruction can jump (that is, transfer control) to another part of your program.

If your SIGNAL instruction is in the middle of a program, the language processor forgets all about the SELECT constructs and DO loops you were in; therefore, you cannot jump back into or jump around within a DO loop. This usually means that you can only use SIGNAL for an abnormal end. For other purposes, it is better to construct your jumps using IF, SELECT, or DO, as described earlier.

Reading 1 continues in Chapter 9, “Input and Output,” on page 165.

Abnormal Changes of Control

Reading 2

To tell the language processor to go to another part of the same file, use the SIGNAL instruction:

```
SIGNAL label
```

This causes a jump to the specified label. A *label* consists of a symbol followed by a colon (:). The language processor searches from the top of the file for the clause:

```
LABEL:
```

Processing continues from there.

Here is an example of an abnormal end using SIGNAL. The SIGNAL instruction always stores its own line number in the REXX special variable SIGL.

```
SIGNAL abend
...

EXIT                            /* end of ordinary code */
/*------------------------------------------------------*/
/* This code handles abnormal ends                      */
/*------------------------------------------------------*/
ABEND:
say "Abnormal end signaled at line" sigl,
 ||".  Cannot continue."
exit
```

The first EXIT instruction is put there to stop the normal program from running on into the *abnormal end* routine.

Reading 2 continues in Chapter 9, “Input and Output,” on page 165.

Conditions and Condition Traps

Reading 3

The CALL ON|OFF and SIGNAL ON|OFF instructions modify the flow of execution in a REXX program by using condition traps. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see “The CALL Instruction” on page 151 and “Abnormal Changes of Control”).

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*, which is described later. SIGNAL or CALL is used, depending on whether the most recent trap for the condition was set using SIGNAL ON or CALL ON respectively.

Condition traps can be set using the CALL ON Condition and SIGNAL ON Condition syntax described below.

The CALL ON Condition

For information about the CALL Instruction, see page 151.

The CALL ON instruction will turn on trapping of the condition you specify. The format is:

```
►►──CALL──ON───┬─ERROR────┬──┬────────────────┬──────────────────────────────►◄
               ├─FAILURE──┤  └─NAME  trapname─┘
               ├─HALT─────┤
               └─NOTREADY─┘
```

The CALL OFF instruction will turn off any trapping of the condition you specified. The format is:

```
►►──CALL──OFF───┬─ERROR────┬─────────────────────────────────────────────────►◄
                ├─FAILURE──┤
                ├─HALT─────┤
                └─NOTREADY─┘
```

The SIGNAL ON Condition

For information about the SIGNAL Instruction, see page 159.

The SIGNAL ON instruction will turn on trapping of the condition you specify. The format is:

```
►►──SIGNAL──ON───┬─ERROR────┬──┬────────────────┬────────────────────────────►◄
                 ├─FAILURE──┤  └─NAME  trapname─┘
                 ├─HALT─────┤
                 ├─NOTREADY─┤
                 ├─NOVALUE──┤
                 └─SYNTAX───┘
```

The SIGNAL OFF instruction will turn off any trapping of the condition you specified. The format is:

```
►►──SIGNAL──OFF───┬─ERROR────┬───────────────────────────────────────────────►◄
                  ├─FAILURE──┤
                  ├─HALT─────┤
                  ├─NOTREADY─┤
                  ├─NOVALUE──┤
                  └─SYNTAX───┘
```

Condition Trap Explanations

The conditions and their corresponding events which can be trapped are:

ERROR

raised if any host command indicates an error condition upon return. It is also raised if any host command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE are set. The condition is raised at the end of the clause that called the command, but will be ignored if the ERROR condition trap is already in the delayed state.

CALL ON ERROR and SIGNAL ON ERROR trap all positive return codes; and will trap negative return codes if neither CALL ON FAILURE nor SIGNAL ON FAILURE are set.

FAILURE

raised if any host command indicates a failure condition upon return, but will be ignored if the FAILURE condition trap is already in the delayed state; that is, a failure is currently being handled.

CALL ON FAILURE and SIGNAL ON FAILURE trap all negative return codes from commands.

HALT

raised if an external attempt is made to interrupt execution of the program. The condition is raised at the end of the clause that was being interpreted when the interruption took place.

NOTREADY

raised if an error occurs during an input or output operation. This condition is ignored if the NOTREADY condition trap is already in the delayed state.

NOVALUE

raised if an uninitialized variable is used:
- As a term in an expression
- As the name following the VAR subkeyword of the PARSE instruction
- As an unassigned variable pattern in a parsing template.

This condition may only be specified for SIGNAL ON.

SYNTAX

raised if an interpretation error is detected. This condition may only be specified for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON or OFF, and any trap name) of that condition trap. Thus, a SIGNAL ON HALT replaces any current CALL ON HALT, and so on.

Action Taken When a Condition is Trapped

When a condition trap is currently enabled (ON has been specified), the trap is in effect. So, when the specified condition occurs, control is passed to the label corresponding to the trapped condition.

If no explicit *trapname* was specified, control is passed to the label or routine that matches the name of the *condition* itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX).

If *trapname* was specified following the NAME subkeyword of the CALL ON or SIGNAL ON instruction, control is passed to the label or routine specified, rather than the name of the *condition*.

The sequence of events, once a condition has been trapped, varies depending on whether a SIGNAL or CALL is executed:

- 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 159).

  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 reenable it once 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 normal syntax error termination will occur.
- If the action taken is a CALL, the CALL is made in the usual way (see page 151) except that the special variable RESULT is not affected by the call. If the routine should RETURN any data, then the returned character string is ignored.

  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.

  On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

The CONDITION Function

The CONDITION function returns the condition information associated with the current trapped condition. See the CALL ON Condition and the SIGNAL ON Condition just described for a description of condition traps. Four pieces of information can be requested:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction executed as a result of the condition trap (SIGNAL or CALL)
- The status of the trapped condition.

The following *parameters* can be supplied to select the requested information. Only the first letter is significant.

Condition name
returns the name of the current trapped condition.

Description
returns any descriptive string associated with the current trapped condition. If no description is available, a null string is returned.

The descriptive string varies, depending on the condition trapped. In the case of SIGNAL or CALL, the descriptive string that is passed to the external environment as command results in one of the following:

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.

NOTREADY
The fully-qualified name of the stream being manipulated when the error occurred and the NOTREADY condition was raised.

NOVALUE The derived name of the variable whose attempted reference caused the NOVALUE condition.

SYNTAX Any string associated with the error by the language processor. This can be the null string if no specific string is provided. Note that the special variable RC and SIGL provide information on the nature and position of the processing error.

Instruction returns the keyword for the instruction executed when the current condition was trapped, being either CALL or SIGNAL. This is the default if *option* is not specified.

Status returns the status of the current trapped condition. This can change during execution, and is either:
ON - the condition is enabled
OFF - the condition is disabled
DELAY - any new occurrence of the condition is delayed.

If no condition has been trapped (that is, there is no current trapped condition), 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 information returned by the CONDITION function is saved and restored across subroutine calls (including those caused by a CALL ON condition trap). Therefore, once a subroutine called because of a CALL ON trap has returned, the current trapped condition reverts to the current condition before the CALL took place. CONDITION returns the values it returned before the condition was trapped.

Congratulations! You have successfully completed Reading 3. Now you can put your REXX skills into action.

If you want more practice with writing REXX programs, you can review Chapter 10, “Programming Style and Techniques,” on page 185.

Chapter 9. Input and Output

Reading 1

REXX can do more than manipulate the information that you have typed at the keyboard and then process it for display on the screen. REXX can store, access, print, and organize data outside the program.

In this chapter:

Reading 1 immediately following, describes:
- A stream of information
- File processing
- Writing data to a stream
- Reading data from a stream
- Counting the data remaining
- Handling streams.

Reading 2 on page 174, describes:
- Data streams
- Default streams
- The STREAM function
- Accessing data within a stream
- Techniques for using the I/O functions.

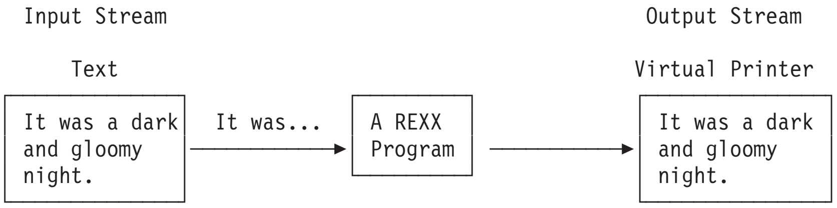
A Stream of Information

In computing, the form of the information is often as important as its content.

A script file, for example contains not only text information, but other information that you never see, such as details about the structure of the file. This additional data describes how the data if formatted and printed. The structure of an exec file is very different from that of a script file—the information takes on other forms. A command in an exec lets you perform a specific function.

The goal of the REXX language is to keep things as simple as possible. Therefore, REXX takes the simplest possible view of the information it receives. The simplest way to look at information is one line at a time.

For example, in a REXX program that reads from a file and then sends the file to a virtual printer, both the file and the virtual printer are character streams:



In the preceding diagram:
- The text being read, `It was a dark and gloomy night.`, is the *input stream*.
- The virtual printer device written to (VIRTUAL PRINTER) is the *output stream*.

In this discussion, a *stream* means any source or destination of external information a REXX program uses. A stream can be a:
- reader file

- punch
- printer
- SFS file
- minidisk file
- program stack (or external data queue)
- default stream (terminal input buffer and the display).

(For the names of these streams that the input and output functions use, see the "Input and Output Streams" chapter of the *z/VM: REXX/VM Reference*.)

The input and output operations of REXX fall into these broad categories:
- Streams of characters
- Lines, or segments of a stream, separated by line end characters.

The way you use data streams in a program depends upon the kind of information you are working with and what you want to do with it.

File Processing

You begin file processing by putting line data into a more or less permanent form, such as in simple files on a disk. You already know some ways to create, read, and write disk files using XEDIT.

In REXX, a set of stream functions that read and write data a single character at a time or line by line perform stream processing. The first function you will try is one that writes lines to a file.

Writing Data to a Stream

You can write line or character data to a stream using the LINEOUT and CHAROUT functions.

LINEOUT (Line Output) Function

To write a line to a file, use the LINEOUT function. The format is:

```
►►──LINEOUT(──┬──────┬──┬──────────────────────────┬──)──────────────────────►◄
              └─name─┘  └─,──┬────────┬──┬───────┬─┘
                             └─string─┘  └─,line─┘
```

where:

name is the name of the file (that REXX regards as a stream) to which the data is written.

string is a line of data to be written.

line a line number to set the write position.

The first time a program uses LINEOUT in this way, the file (*name*) is opened for writing and the line (*string*) is written to the end of the stream.

The stream remains open, and each subsequent LINEOUT call writes a new line to the end of the file (*name*).

When the program ends, REXX automatically closes *name* unless you close it explicitly with the STREAM function or specific forms of CHAROUT and LINEOUT. Or, you can close *name* explicitly at any time by omitting the *string* argument. For example:

```
LINEOUT(name)
```

A LINEOUT Example

Figure 90 shows an example of a simple text editor. It writes new text to a file. Look closely at the use of the LINEOUT function.

```
/* EDDY EXEC */
/* World's smallest editor for CMS minidisk file */
say 'Enter a file name and file type.'
pull fileid
say 'Enter as many lines as you like.',
    'To finish, press Enter.'
do forever
   parse pull line


   if line = '' then          /* empty line?           */
      do
         call LINEOUT          /* if so, close the file */
         exit                  /* and end the program   */
         end


   /* otherwise, write LINE to the end of the file */
   call LINEOUT fileid, line
   if result = 1 then leave
end
say 'Error, return code' result
exit
```

Figure 90. EDDY EXEC

You enter a file name, which the PULL instruction then parses and stores in the variable *fileid*. As each line is typed:

- The PARSE PULL instruction stores it as a string in the variable *line*.
- The LINEOUT function (called as a subroutine) writes the string contained in *line* to the file name stored in the variable *fileid*.
- The DO loop continues until you press Enter twice, thereby entering a null string.
- The program then calls LINEOUT with only the file name (closing the file), and exits.

Calling LINEOUT

LINEOUT is a function call rather than a keyword instruction. That means that it not only performs a given task (writing data to a file, in this example), but that it also *returns a value of*:

- `0` if *string* was successfully written to *name*.
- `1` if for any reason *string* could not be written, for example, if you try to write to a *read-only* file.

Your program can use the return value to detect whether something has gone wrong in the course of writing to a stream.

Note: If you use LINEOUT without the *string* argument, the return value tells you if *name* was successfully closed.

LINEOUT is a function call; therefore, you have a choice about how you can use it. You can call LINEOUT:

- As part of a REXX instruction. For example, with the keyword SAY:

```
say LINEOUT("mybook text","Chapter 1.")      /* displays "0"         */
                                             /* if successful        */
```

or as a variable assignment:

```
ready = LINEOUT("mybook text","Chapter 1.")  /* assigns "0" to READY */
                                             /* if successful        */
```

- As a subroutine with arguments (this is true of all function calls). For example:

```
call LINEOUT "mybook script", "Chapter 1." /* Result=0 if successful */
```

When you use LINEOUT (or any function call) in this way, the return value of the function is automatically assigned to the REXX special variable RESULT.

Resetting the Write Position

You can specify a line number to set the write position to the start of a particular line in a file. This line number must be positive and within the bounds of the file (although you can specify the line number immediately after the end of the file). A value of 1 for *line* refers to the first line in the stream. For example:

```
LINEOUT(myfile,,1)       ->   0              /* now at line 1        */
LINEOUT(myfile,,6)       ->   0              /* now at line 6        */
```

CHAROUT (Character Output) Function

The CHAROUT function writes single-byte characters. The format is:

```
►►──CHAROUT(──┬──────┬──┬────────────────────────┬──)──────────────────────►◄
              └─name─┘  └─,──┬────────┬──┬────────┬┘
                             └─string─┘  └─,start─┘
```

where:

name is the name of the stream to which the character is written

string is a string of characters to write

start is the starting position of a stream of characters.

Like the LINEOUT function, CHAROUT returns 0 if all the characters in the string (*string*) are successfully written to the file (*name*). Unlike LINEOUT, if for any reason CHAROUT cannot write to the named file (*name*), it returns the number of characters that remain unwritten.

CHAROUT is also similar to LINEOUT in that it is more convenient to call it as a subroutine.

The first time a program uses CHAROUT, the named stream is opened for writing and the characters are written to the end of the stream. If the write pointer is moved (using LINEOUT or STREAM), CHAROUT will write over existing data in the line. The start value for CHAROUT must be 1 (default value). CHAROUT will always start with the first character in the line.

A CHAROUT Example

The following exec will write out the variable in a do loop using CHAROUT. This example shows how multiple uses of CHAROUT will continue to write to the same line. The output of this exec will be one line. If LINEOUT was used, the output would be 12 separate lines, each with one character.

```
/* CHAROUT1 EXEC */
file = 'TEST FILE A1'                      /* File name          */
do i=1 to 3                                /* First Do Loop      */
 z = CHAROUT(file,i)                       /* Write value for i  */
  do j=9 to 7 by -1                        /* Second Do Loop     */
   y = CHAROUT(file,j)                     /* Write value for j  */
  end                                      /* End second Do Loop */
end                                        /* End first Do Loop  */
exit
```

Figure 91. CHAROUT1 EXEC

Reading Data from a Stream

If a stream supports reading, you can read characters or lines from a stream using the LINEIN and CHARIN functions.

LINEIN (Line Input) Function

To read a line from a stream into a REXX program, use the LINEIN function. The format is:

```
►►──LINEIN(──┬──────┬──┬───────────────────────┬──)──────────────────────────►◄
             └─name─┘  └─,─┬──────┬─┬────────┬─┘
                           └─line─┘ └─,count─┘
```

where:

name is name of the data stream (such as a file) from which the line is read.

line the read position of the string.

count the number of lines read from the character input stream.

The first time a program calls LINEIN, it opens the named file (*name*) for reading and returns the first line of data. The second call to LINEIN, reads the second line and returns the second line of data, and so on until the program ends (or you close the stream with LINEOUT). In other words, LINEIN keeps track of its place in the stream with a kind of “bookmark”, called the *read pointer*. LINEOUT uses a similar marker, the *write pointer*. For more information, see “Accessing Data within a Stream” on page 181.

Resetting the Read Position

You can select how many lines are displayed. If you type a number larger than the number of lines in the file, the program cycles back to the beginning. To do this, use LINEIN with its second and third arguments. For example:

`LINEIN(`*`stream`*`,`*`line`*`,`*`count`*`)`

The options for *line* are:

- no argument (the default)—to leave the read position where it is.
- `1`—to set the read pointer to the beginning of the stream (line 1).
- *n*—to point to a line within the bounds of the file

The options for *count* are:

- `1` (the default)—to read one line and advance the read position.
- `0`—to read no lines and not advance the read position.

The first time a program calls LINEIN, it opens the named file:

```
LINEIN(filename)
```

So far, LINEIN has included the default values for the second and third arguments to simply read the next line. By setting `1` for the *line* and `0` for the *count*, LINEIN can reset the read position to the beginning of the stream without reading a line (or advancing the read position). For example:

```
LINEIN(filename,1,0)
```

Because you are not interested in the return value (which would be empty anyway), you can call LINEIN as a subroutine:

```
call LINEIN filename,1,0
```

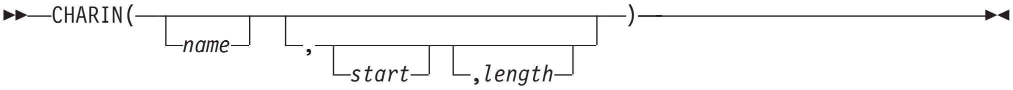
Figure 92 shows an example calling LINEIN as a subroutine to reset the read position.

```
/* SHOLIN1 EXEC                                      */
/* Displays a given number of lines in a text file   */
/* If the given number exceeds the number of lines   */
/* in the file then the read position is reset back  */
/* to the beginning of the file                      */
say "Type a file name and file type"
pull fileid
say "Type number of lines to display"
pull howmany
lineno = 1
do howmany
   say lineno LINEIN(fileid)
   lineno = lineno + 1

   if LINES(fileid) = 0 then       /* if the end of file is reached */
     do
        call LINEIN fileid,1,0     /* reset the read position       */
        say ">>> End of File <<<"  /* display end-of-file marker    */
        lineno = 1                 /* reset line counter            */
        end
   end
exit
```

Figure 92. SHOLIN1 EXEC

CHARIN (Character Input) Function

You can read characters from a stream into a REXX program with CHARIN. The format is:



where:

name is the name of the data stream (such as a file) from which the characters are read.

start the starting position of the character in the stream.

length the number of single-byte characters to read from the character input stream.

CHARIN works very much the same as LINEIN, except that CHARIN reads characters instead of lines.

The start value for CHARIN must always be 1 (the default). Figure 93 shows one way to find a specific character in the line by using CHARIN. The exec will loop until the correct character is located, and the position will be written to the screen.

```
/* SHOCHAR1 EXEC                                */
/*                                              */
/* Shows one way to find a specific character in */
/* a line by using CHARIN.  The exec will loop   */
/* until the correct character is located, and   */
/* the position will be written to the screen.   */
/*                                              */
/* Assume a file, CHARIN1 TESTDATA A, exists and */
/* contains all of the keyboard characters.      */

fileid = 'CHARIN1 TESTDATA A'
char_count = 0
done = 0
say "Type character to locate"
parse pull char
do until done
  if chars(fileid) ¬= 0 then
    do
      in_char = charin(fileid,,1)
      char_count = char_count + 1
      if in_char = char then done = 1
    end
    else done = 1
end
if chars(fileid) = 0 then
  do
    say 'Selected character was not found in the file'
  end
  else say 'Character' char 'was found in character position' char_count
exit
```

Figure 93. SHOCHAR1 EXEC

Counting the Data Remaining

You can count the number of lines or characters remaining in your file using LINES and CHARS.

LINES (Lines Remaining) Function

To find out if any lines remain between the read position and the end of a stream, use LINES. The format is:

```
►►──LINES(──────────)────────────────────────────────────────────►◄
          └─name─┘
```

where:

name is the character input stream

The LINES function returns the following:

0 if no complete lines remain to be read

n if any lines remain, the number remaining.

Figure 94 on page 172 shows an example, where *name* is a text file, so LINES would return 0 when the end of the file has been reached.

LINEIN has no way of knowing when there are no more lines to read in a stream, such as when it gets to the end of a file. To know when the read position has reached the end of the file, use the LINES function before LINEIN.

```
/* SHOLIN2 EXEC                       */
/* Program to display an entire file */
/* exits when end-of-file is reached */

say "Type a file name and file type"
                                /* get the name of the file           */
pull fileid                     /* from the user                     */

lineno = 1                      /* initialize a counter to display   */
                                /* the line number

do until LINES(fileid) = 0      /* repeat this loop until no lines   */
                                /* remain in the selected file...    */

   say lineno LINEIN(fileid)    /* display the line number, and then */
                                /* read and display a line of text,  */
                                /* advancing the read position       */
                                /* with each pass though the loop    */

   lineno = lineno + 1          /* increment the line-number counter */

   end
exit                            /* end the program                   */
```

Figure 94. SHOLIN2 EXEC

CHARS (Characters Remaining) Function

To find out if any characters remain in the input stream, user CHARS. The format is:

```
►►──CHARS(──┬──────┬──)──────────────────────────────────────────────►◄
            └─name─┘
```

where:

name is the character input stream.

The CHARS function returns the following:

0 if no characters remain in the stream

1 if at least one character remains in the stream.

CHARS is similar to LINES with respect to reaching the end of the file. CHARS will return a 0 when the read pointer has reached the end of the file.

Handling Streams

Sometimes you will need to do things with the streams you are using, such as opening and closing them. These streams can be several things: minidisk files, SFS files, spool files (reader, printer, and punch), or the program stack. For this discussion, we will assume that input and output are communicating with a human user. A character stream might, in fact, have a variety of sources or destinations, such as files or displays.

A character stream can be *transient* or *persistent*. Transient (also referred to as dynamic) means that it is considered temporary, for example, the default input stream or the program stack. Persistent means it is a static form or more permanent form, for example, a file or data object.

The STREAM function is provided to help you handle streams.

Opening and Closing Files

The STREAM function (see "STREAM Function" on page 178) lets you explicitly open or close a stream. However, as you have already learned, there are other ways of opening or closing a stream. Any line or character I/O function implicitly opens a stream if it is not already open:

- CHARS and LINES implicitly open the stream for writing if possible, reading otherwise.
- CHARIN and LINEIN implicitly open the stream for reading.
- CHAROUT and LINEOUT implicitly open the stream for writing.
- The STREAM function explicitly opens a stream for reading or writing.

Each time an SFS file is opened, it is associated with a new work unit ID. This also applies to multiple opens of the same file. The stream then remains open for subsequent I/O as long as an explicit close is not issued. You can close a stream with the STREAM function or specific forms of CHAROUT and LINEOUT. If you do not explicitly close the stream, it remains open until the completion of the last active REXX program, at which time it is automatically closed. Before closing the stream, any remaining buffered data from character output operations is written out.

Note: Only SFS and minidisk files can be opened multiple times.

To Summarize

Here is a review of the input and output functions that have been discussed so far.

Use this function	To do this
LINEOUT(*name*,*string*)	To open *name* and append *string* (write it to the end of *name*). Returns 0 if successful; 1 if otherwise.
LINEOUT(*name*,*string*,*line*)	To open file *name*, and write *string* to *line*. The existing line will be overwritten.
LINEOUT(*name*)	To close *name* when writing is completed. Returns 0 if successful; 1 if otherwise.
CHAROUT(*name*,*string*,*start*)	To open file *name*, and write characters in *string* to end of the file starting in position 1.
CHAROUT(*name*)	To close *name*.
LINEIN(*name*) or LINEIN(*name*,,1)	To open *name*, read the first line and advance the read pointer to the second line. If *name* is already open, then LINEIN reads the current line and advances the read pointer one line ahead.

Use this function	To do this
LINEIN(*name*,1,0)	To put the read pointer at the beginning of the *name* **without** reading a line or advancing the read pointer.
LINEIN(*name*,1,1)	To put the read pointer at the beginning of the *name* **and** specifically read the first line (advancing the read pointer to the second line). The action is the same whether or not *name* is already open.
LINEIN(*name*,,0)	To open *name* without reading the first line or advancing the read pointer. No action is taken if *name* is already open.
CHARIN(*name*,*start*,*length*)	To open file *name* and read *length* number of characters beginning with the first character in the line.
CHARIN(*name*,1,0)	To place the read pointer to the beginning of *name* without reading a character.
CHARIN(*name*)	To close *name*.
CHARS(*name*)	To find out if any characters remain in the stream.
LINES(*name*)	To find out how many lines remain in the stream.

Table 4 shows the REXX functions that read from and write to a data stream.

Table 4. Read and Write Functions

	Read Data	Write Data	Check for Lines or Characters Remaining
Characters	CHARIN()	CHAROUT()	CHARS()
Lines	LINEIN()	LINEOUT()	LINES()

Reading 1 continues in Chapter 10, “Programming Style and Techniques,” on page 185.

Additional Stream I/O Information

Reading 2

In this section you will learn more about:
- Data streams
- Default streams
- STREAM function
- Accessing data within a stream
- Techniques for using the I/O functions.

More about Data Streams

REXX regards all information from any file or device as a continuous *stream* of single characters. Data read into a REXX program (whether from a disk file, the keyboard, a device, or another program) is also processed as a character stream. The same is true for output data that a REXX program writes to a file or other device. All of these are streams.

Your program can work with the information in a stream one character at a time; or if the data is in line form, you can manipulate the information from the stream (or put information into it) line by line.

As shown earlier in this chapter, your REXX programs can access and manipulate text files by using:

LINEIN()	to read a line
LINEOUT()	to write a line
LINES()	to count the lines remaining in the stream
CHARIN()	to read one or more characters
CHAROUT()	to write one or more characters
CHARS()	to count the characters remaining in the stream.

Default Streams

Each of the I/O functions listed here has as its first argument the name of a stream that is read or written to. Each of these functions also has a *default stream* that is used if you omit the name of a specific stream. The default input stream (for LINEIN and CHARIN functions) is the terminal input buffer or user input if the terminal buffer is empty. The default output stream is your terminal.

This means that you can use the LINEIN function to pause processing and read a line entered at the keyboard, as you can with the PARSE PULL instruction. But note these differences:

- Unlike PARSE PULL, the LINEIN function reads *only* keyboard entries, regardless of whether there are outstanding items on the default data queue.
- LINEIN would cause a `VM READ` if no string is available in the default input stream.

To understand how this works, use the first file-reading program, SHOLIN2 EXEC (see Figure 94 on page 172) and add an instruction as shown in Figure 95 on page 176.

```
/* SHOLIN3 EXEC                                 */
/* Displays a file one line at a time           */
/* as you press Enter; program                  */
/* ends when the end-of-file is reached         */
/* OR if user types any character.              */
say "Type a file name and file type"
pull fileid
lineno= 1
do until LINES(fileid) = 0
   say lineno LINEIN(fileid)
   if LINEIN() \= "" then leave    /* waits for user to press Enter  */
                                   /* if anything else is typed     */
                                   /* (if LINEIN() does not return  */
                                   /* an empty string), then the    */
                                   /* loop (and the program) ends   */
   lineno = lineno + 1
   end
exit
```

Figure 95. SHOLIN3 EXEC

Or, you could modify the cycling version of this program, SHOLIN1 EXEC (see Figure 92 on page 170) to let you choose the number of lines to display. To do this, put the display routine inside a DO FOREVER loop, as shown in Figure 96 on page 177.

```
/* SHOLIN4 EXEC                         */
/* Displays a file one line at a time   */
/* as you press Enter, or               */
/* displays a given number of lines.    */
/* Cycles back to the beginning of the  */
/* file when the end-of-file is reached */
/* Program ends only when user types    */
/* any non-numeric character.           */
say "Type a file name and file type"
pull fileid
say "Type a number of lines to display"
say "or press Enter to advance one line"
say "Type any other character to end."
lineno = 1
do forever
   howmany = LINEIN()                 /* pause for user entry and store */
                                      /* it in the variable HOWMANY     */

   if howmany = "" then howmany = 1   /* pressing Enter                 */
                                      /* is the same as entering '1'    */

   if \datatype(howmany,n) then leave /* entering any non-numeric       */
                                      /* character ends the program     */

   do howmany
      say lineno LINEIN(fileid)
      lineno = lineno + 1

      if LINES(fileid) = 0 then
        do
           call LINEIN fileid,1,0
           say ">>> End of File <<<"
           lineno = 1
           end
      end

   end
exit
```

Figure 96. SHOLIN4 EXEC

Parsing Default Input

You can parse the input for individual words, either by using the instruction:

```
PARSE VALUE LINEIN() WITH var1 var2 ...
```

For more information about the PARSE instruction and its options, see “Parsing Variables and Expressions” on page 96. Also, see the *z/VM: REXX/VM Reference*.

Performing Stream Tasks

You can use the STREAM function to get the following information about a stream:

- To determine if a stream exists
- To determine if a stream is ready for input or output
- To get characteristics of the stream.

You can also use STREAM for more complex input and output tasks:

- Open a stream for reading and writing.
- Close a stream at the end of the operation.
- Query the stream.

- Move the pointer.

See “STREAM Function” for examples of STREAM.

STREAM Function

For more intricate and specialized input and output tasks, REXX provides another function called STREAM. The format is:

```
►►──STREAM(name──┬───────────────────────────────┬──────────────────────►◄
                 │     ┌─State──────────────────┐│
                 └─,───┼────────────────────────┼┘
                       ├─Command,stream_command─┤
                       └─Description────────────┘
```

where:

name is the stream you want to work on. You must specify the stream name.

S the state of the stream

C a command or action to be taken

D a more detailed description

stream_command
is an action to perform. You must use this argument when and only when you specify C.

The syntax may look a bit complicated at first, because STREAM has a wide variety of applications, such as:

- The **C** (Command) operation lets your program select and gain access to a named stream.
- The **S** and **D** (State and Description) operations report the current status of a stream; that is, whether:
  - the stream is READY or NOTREADY for input or output
  - it is UNKNOWN (not yet identified)
  - an input or output ERROR has occurred.

The following exec will open a file with an LRECL of 80 and fixed format. After the file is open, the stream is queried (with STREAM OPEN) to verify the file has been created with the correct parameters. The LRECL and RECFM parameters on the STREAM OPEN command should only be used with NEW or REPLACE (or non-existent files). If the file already exists, the LRECL and RECFM will not change the characteristics of the file. The values on the STREAM OPEN are ignored.

```
/* STREAM EXEC */
SIGNAL ON NOTREADY                    /* Setup for any problems        */
filename = 'OPENTEST FILE A'          /* filename                      */
parse value stream(filename,'C','OPEN LRECL 80 RECFM F') with ok handle
                                      /* Open file with specific format */

say stream(handle,'C','QUERY FORMAT') /* Query to ensure file created   */
c = stream(handle,'C','close 2')      /* Close the stream              */
exit                                  /* End the program               */

NOTREADY:                             /* If a NOTREADY condition occurs */
say stream('STREAM FILE A','D');      /* return description            */
```

Figure 97. STREAM EXEC

Another use of the STREAM command is to move the read or write pointer to a particular line. In Figure 98 an existing file (with more than two lines) is opened. The write pointer position will be queried. The STREAM command moves the write pointer to the second line in the file. (See the *z/VM: REXX/VM Reference* for different ways to move the pointer.) The second query should return a `2` to show the write pointer is now pointing at the second line.

```
/* STREAMLP EXEC */
CALL ON NOTREADY
file = 'TESTIT FILE A'                       /*File with > 2 lines    */
parse value stream(file,'C','OPEN WRITE')with ok handle
                                             /* Open file for write   */
say stream(handle,'C','query linepos write') /* Query write pointer   */
a = stream(handle,'C','linepos 2 WRITE')     /* Move write pointer to */
                                             /* second line of file   */
say stream(handle,'C','query linepos write') /* Query write pointer   */
                                             /* Should = 2            */
c = stream(handle,'C','close')               /* Close the stream      */
exit                                         /* End of program        */
NOTREADY:                                    /* NOTREADY trap         */
say stream('TESTIT FILE A','D');             /* Stream Description    */
```

Figure 98. STREAMLP EXEC

Note: The STREAM LINEPOS should be used with care, especially if the write pointer is moved. Data may be lost if the wrong line is written over.

For the full syntax of STREAM and the other REXX input and output functions, see the *z/VM: REXX/VM Reference*.

Getting Information about a Stream

To determine if a particular stream exists, use the stream command QUERY EXIST with the STREAM function call. For example:

```
stream(name,C,'query exists')
```

Note that the stream command is enclosed in matching quotation marks.

If the stream *name* exists, then this function call returns the name of the stream. For example:

```
test file a1
```

If the stream *name* does not exist, then the result is a null string.

Figure 99 shows an example of a program that reads a file.

```
/* QRYFILE1 EXEC                         */
/* For a program that reads a file       */
say "Type a file name and file type (or press Enter to exit): "
pull fileid
if fileid = "" then exit
/* Check that the file exists:           */
/* if STREAM() returns a null string,    */
/* then report the stream not found      */
/* and exit....                          */
call stream fileid, C, 'query exists'
if result = "" then
   do
     say "Cannot find" fileid"."
     exit
   end
/* ...else store the fully-qualified name  */
/* (in RESULT) to the variable FILEID.     */
else fileid = result
say "Full name is" fileid
```

Figure 99. QRYFILE1 EXEC

You can also query the size of a stream and the date and time of the last edit; see Figure 100.

```
/* QRYFILE2 EXEC                  */
/* How big and when last changed? */
say "Type a file name and file type (or press Enter to exit): "
pull fileid
⋮
LINECOUNT = stream(fileid,c,'query size')
ledit = stream(fileid,c,'query datetime')
say fileid "is" lines "lines."
say "Last edit of" fileid "was" ledit"."
```

Figure 100. QRYFILE2 EXEC

Opening and Closing Streams

The functions LINEOUT, LINEIN, CHARIN and CHAROUT do much of their own *housekeeping*. They automatically open the streams they work on and leave REXX to close the stream at the end of a program unless you explicitly close the stream with the STREAM function or CHAROUT or LINEOUT.

However, there are cases where it is necessary (or at least more prudent) to explicitly open and close a stream, such as in a program that reads from more than one device or one that writes to the middle of a file.

You can do this with the STREAM function:

```
stream(name,c,"open")
```

This default form opens a stream *name* for both reading and writing text. To open a stream for:

- Reading only, add the word `read`. For example:

  ```
  /* Open the file for read. */
  parse value stream('TEST FILE A','C','OPEN READ') with ok file_handle
  if ok ¬= 'READY:' then signal open_error
  number_of_lines = lines(file_handle)
  ```

- Reading and writing, add the word `write`. For example:

  ```
  /* Open the file for write. */
  parse value stream('TEST FILE A','C','OPEN WRITE') with ok file_handle
  if ok ¬= 'READY:' then signal open_error
  number_of_lines = lines(file_handle)
  ```

When you open a stream in this way, STREAM returns the string `READY:` if the stream has been successfully opened. It returns an error message if for any reason it was unable to open the stream.

The variable returned on the stream opened in `file_handle` is a unique identifier for that particular opening of the stream. This is especially important when a named data stream can be opened more than one time, and a unique identifier is needed to reference the different stream openings. When the stream is opened implicitly (by LINEIN or CHARIN, for example), the user is unable to obtain the unique identifier. In the previous examples, the unique identifier is used for the LINES function. The unique identifier may be used for any I/O functions other than `stream(`*`filename`*`,'c',open)`. By using the unique identifier, performance will be increased.

To explicitly close a stream, use:

```
stream(name,c,"close")
```

In this form, STREAM returns the string `READY:` if the operation is successful, the string `ERROR:` if the operation fails.

Accessing Data within a Stream

REXX regards all external data as streams of information. Nonetheless, these streams can take different forms. A minidisk file, for example, differs from a spool file in that it has a static, physical form. A minidisk file is one example of a *persistent* stream. This means a program can read from or write to anywhere in the file.

As a program reads a file, REXX keeps a place marker, called the *read position*, that points to the next character (or line).

The same is true when writing. REXX maintains a *write position* that marks the next place to write.

You can specify another position for the read or write positions by giving additional arguments on the stream functions LINEIN, LINEOUT, CHARIN, CHAROUT, or by using the `LINEPOS` option of the STREAM function (see "STREAM" in the *z/VM: REXX/VM Reference*.)

For more information about these functions, see the *z/VM: REXX/VM Reference*.

A stream of data may be a string of characters separated by line end (LINEEND) characters. When you open a stream, you can specify the type of file and if LINEEND characters are important. If the opened stream specified BINARY, it means that all character codes may be present in the data stream, and no indication of LINEEND characters will be provided or searched for. If the TEXT option is specified, it means LINEEND characters are appended to the end of each line when passing data to the user on character input operations. These LINEEND characters are never written to the data stream. Line operations are not affected by this parameter, only character operations. Figure 101 on page 182 shows how a

LINEEND character is inserted:

```
/* CHAROUT2 EXEC */
lend= '15'x                          /* line end character = 15'x' */
file = 'TESTIT FILE A'               /* file name                  */
Z = stream(file,C,'OPEN TEXT lineend 15')/* open with TEXT and lineend */
g = charout(file,'abc'lend'def')     /* write abc on one line      */
                                     /* write def on next line     */
Z = stream(file,C,'CLOSE')           /* close the file             */
exit                                 /* end the program             */

/* Output file would look like:                                     */
/* .                                                                */
/* .                                                                */
/* abc                                                              */
/* def                                                              */
```

Figure 101. CHAROUT2 EXEC

For more information on LINEEND and TEXT files, see “STREAM” in the *z/VM: REXX/VM Reference*.

Techniques For Using REXX I/O Functions

This section addresses some techniques to considered when writing a REXX program using the REXX I/O functions. Some questions you may have are:

- When should I explicitly open and close streams?
- How do the REXX functions interact with existing CMS I/O facilities such as EXECIO, XEDIT, and CMS Pipelines?
- How can I code my programs to tolerate errors encountered while performing an I/O operation?

The following sections address these and other questions.

To Open or not To Open

As you have seen, functions such as LINEIN and LINEOUT allow the REXX programmer to implicitly open a stream. That is, the functions open the stream on behalf of the programmer using defaults as appropriate. However, the defaults sometimes do not satisfy the needs of your program. What if you want to create a fixed format file? The default creates a variable format file. In this case, you would need to code the following before writing any lines or characters to the file:

```
Call STREAM name,'C','OPEN WRITE RECFM F'
```

It is recommended that you use explicit calls to the STREAM function to open a stream before doing any I/O.

Note: The exception is the default stream, where an explicit open or close is not recommended. Explicit opens enhance the logical flow of the program. They also let the programmer use the “handle” the OPEN command returns in subsequent I/O operations.

In addition, the program should close any stream it opens before the REXX program ends. The stream command CLOSE is the recommended way to close a stream.

REXX I/O and CMS

Your program should do input and output to a given stream exclusively with REXX or exclusively with the CMS supplied routines. REXX needs to maintain its own control blocks that are different from the ones CMS maintains. Mixing types of I/O can cause unpredictable results. One situation in particular, however, could happen under usual circumstances. CMS should close a stream while REXX is actively doing I/O on the stream. This would occur when the program is using a shared file system file, and a rollback happens. The file will no longer be open to REXX, and any attempted I/O on that file generates an error with a special reason code. In this case, REXX releases the control block for that file and considers the file closed.

Error Handling

Any REXX program that uses the built-in I/O functions should enable a NOTREADY condition trap. This lets the program provide more detailed diagnosis information in the event that one of the I/O functions encounters an error. A sample NOTREADY condition trap follows:

```
/* A sample REXX program
Call on NOTREADY
.

NOTREADY:
Say "I/O Error:" CONDITION('D')
Return
```

Alternate Techniques

For alternate techniques for doing I/O in your programs using CSL, EXECIO, and PIPE commands, see the following books:

- *z/VM: CMS Application Development Guide*
- *z/VM: CMS Commands and Utilities Reference*
- *z/VM: CMS Pipelines User's Guide*.

Reading 2 continues in Chapter 10, “Programming Style and Techniques,” on page 185.

For more complete information about using input and output streams, see the *z/VM: REXX/VM Reference*.

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.

In this chapter:

Reading 1 immediately following, describes:
- Consider the data
- Happy hour with a *real* program.

Reading 2 on page 188, describes:
- Designing a program: stepwise refinement
- Correcting your program
- Coding style.

Consider the Data

Reading 1

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!
- Ready;
- That's not a valid answer. Try again!

And this sequence must be repeated indefinitely, ending with the return to CMS (Ready;).

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.

Test Yourself...

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

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
```

Happy Hour

As this is the end of Reading 1, 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 on, you may want to improve it. (We shall discuss this at the end of the second reading.)

```
/* 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 102. CATMOUSE EXEC (Part 1 of 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's 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.             */
/*------------------------------------------------------*/
display:
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 102. CATMOUSE EXEC (Part 2 of 2)

Congratulations! You have successfully completed Reading 1. Now, maybe you want to take a while to put your new skills into action, or maybe you want to start right in with the second reading.

Reading 2 begins in Chapter 2, “Starting Out with REXX,” on page 5.

Designing a Program

Reading 2

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
```

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.

The Conclusion

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

```
call display
say who won
```

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?

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.

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.

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 “Tracing” on page 37).

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

Modifying Your Program

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

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

Another debug method is:

```
 PIPES REXX VARS / > REXX VARS A
```

This will put into file REXX VARS A all the values of the variables at the point when the PIPE command is inserted into the file. See *z/VM: CMS Pipelines Reference* for more information.

Tracing Your Program

Or you can use the TRACE instruction, described in your *z/VM: REXX/VM Reference*.

- 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 103. 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:
Ready;
```

- 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.

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 coworker to read it. Be sure to choose a coding style that your coworkers find easy to read.

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

```
/********************************************************/
/* SAMPLE #1:  A portion of CATMOUSE EXEC (page 187),   */
/* 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 187),   */
/* 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 187),   */
/* 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! You have successfully completed Reading 2. Now, maybe you want to take a while to put your new skills into action, or maybe you want to start right in with Reading 3.

Reading 3 begins in Chapter 2, “Starting Out with REXX,” on page 5.

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 all countries. Consult your local IBM representative for information on the products and services currently available in your area. Any reference to an IBM product, program, or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any IBM intellectual property right may be used instead. However, it is the user's responsibility to evaluate and verify the operation of any non-IBM product, program, or service.

IBM may have patents or pending patent applications covering subject matter described in this document. The furnishing of this document does not grant you any license to these patents. You can send license inquiries, in writing, to:

IBM Director of Licensing
IBM Corporation
North Castle Drive
Armonk, New York 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 106-0032, Japan

The following paragraph does not apply to the United Kingdom or any other country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF NON-INFRINGEMENT, MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This information could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time without notice.

Any references in this information to non-IBM Web sites are provided for convenience only and do not in any manner serve as an endorsement of those Web sites. The materials at those Web sites are not part of the materials for this IBM product and use of those Web sites is at your own risk.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation to you.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact:

IBM Corporation
Mail Station P300
2455 South Road
Poughkeepsie, New York 12601-5400
U.S.A.
Attention: Information Request

Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

The licensed program described in this document and all licensed material available for it are provided by IBM under terms of the IBM Customer Agreement, IBM International Program License Agreement or any equivalent agreement between us.

Any performance data contained herein was determined in a controlled environment. Therefore, the results obtained in other operating environments may vary significantly. Some measurements may have been made on development-level systems and there is no guarantee that these measurements will be the same on generally available systems. Furthermore, some measurement may have been estimated through extrapolation. Actual results may vary. Users of this document should verify the applicable data for their specific environment.

Information concerning non-IBM products was obtained from the suppliers of those products, their published announcements or other publicly available sources. IBM has not tested those products and cannot confirm the accuracy of performance, compatibility or any other claims related to non-IBM products. Questions on the capabilities of non-IBM products should be addressed to the suppliers of those products.

All statements regarding IBM's future direction or intent are subject to change or withdrawal without notice, and represent goals and objectives only.

This information may contain examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

COPYRIGHT LICENSE:

This information may contain sample application programs in source language, which illustrate programming techniques on various operating platforms. You may copy, modify, and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the application programming interface for the operating platform for which the sample programs are written. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs.

Programming Interface Information

This book documents intended Programming Interfaces that allow the customer to write programs to obtain services of z/VM.

Trademarks

IBM, the IBM logo, and ibm.com are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at “Copyright and trademark information” at www.ibm.com/legal/copytrade.shtml

Adobe is either a registered trademark or trademark of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Glossary

For a list of z/VM terms and their definitions, see *z/VM: Glossary*.

The z/VM glossary is also available through the online z/VM HELP Facility. For example, to display the definition of the term “dedicated device”, issue the following HELP command:

```
help glossary dedicated device
```

While you are in the glossary help file, you can do additional searches:

- To display the definition of a new term, type a new HELP command on the command line:

  ```
  help glossary newterm
  ```

  This command opens a new help file inside the previous help file. You can repeat this process many times. The status area in the lower right corner of the screen shows how many help files you have open. To close the current file, press the Quit key (PF3/F3). To exit from the HELP Facility, press the Return key (PF4/F4).
- To search for a word, phrase, or character string, type it on the command line and press the Clocate key (PF5/F5). To find other occurrences, press the key multiple times.

  The Clocate function searches from the current location to the end of the file. It does not wrap. To search the whole file, press the Top key (PF2/F2) to go to the top of the file before using Clocate.

Bibliography

See the following publications for additional information about z/VM. For abstracts of the z/VM publications, see *z/VM: General Information*.

Where to Get z/VM Information

z/VM product information is available from the following sources:

- z/VM Information Center at publib.boulder.ibm.com/infocenter/zvm/v6r1/index.jsp
- z/VM Internet Library at www.ibm.com/eserver/zseries/zvm/library/
- IBM Publications Center at www.elink.ibmlink.ibm.com/publications/servlet/pbi.wss
- *IBM Online Library: z/VM Collection on DVD*, SK5T-7054

z/VM Base Library

Overview

- *z/VM: General Information*, GC24-6193
- *z/VM: Glossary*, GC24-6195
- *z/VM: License Information*, GC24-6200

Installation, Migration, and Service

- *z/VM: Guide for Automated Installation and Service*, GC24-6197
- *z/VM: Migration Guide*, GC24-6201
- *z/VM: Service Guide*, GC24-6232
- *z/VM: VMSES/E Introduction and Reference*, GC24-6243

Planning and Administration

- *z/VM: CMS File Pool Planning, Administration, and Operation*, SC24-6167
- *z/VM: CMS Planning and Administration*, SC24-6171
- *z/VM: Connectivity*, SC24-6174
- *z/VM: CP Planning and Administration*, SC24-6178
- *z/VM: Getting Started with Linux on System z*, SC24-6194
- *z/VM: Group Control System*, SC24-6196
- *z/VM: I/O Configuration*, SC24-6198
- *z/VM: Running Guest Operating Systems*, SC24-6228
- *z/VM: Saved Segments Planning and Administration*, SC24-6229
- *z/VM: Secure Configuration Guide*, SC24-6230
- *z/VM: TCP/IP LDAP Administration Guide*, SC24-6236
- *z/VM: TCP/IP Planning and Customization*, SC24-6238
- *z/OS and z/VM: Hardware Configuration Manager User's Guide*, SC33-7989

Customization and Tuning

- *z/VM: CP Exit Customization*, SC24-6176
- *z/VM: Performance*, SC24-6208

Operation and Use

- *z/VM: CMS Commands and Utilities Reference*, SC24-6166
- *z/VM: CMS Pipelines Reference*, SC24-6169
- *z/VM: CMS Pipelines User's Guide*, SC24-6170
- *z/VM: CMS Primer*, SC24-6172
- *z/VM: CMS User's Guide*, SC24-6173
- *z/VM: CP Commands and Utilities Reference*, SC24-6175
- *z/VM: System Operation*, SC24-6233
- *z/VM: TCP/IP User's Guide*, SC24-6240
- *z/VM: Virtual Machine Operation*, SC24-6241
- *z/VM: XEDIT Commands and Macros Reference*, SC24-6244
- *z/VM: XEDIT User's Guide*, SC24-6245
- *CMS/TSO Pipelines: Author's Edition*, SL26-0018

Application Programming

- *z/VM: CMS Application Development Guide*, SC24-6162
- *z/VM: CMS Application Development Guide for Assembler*, SC24-6163
- *z/VM: CMS Application Multitasking*, SC24-6164
- *z/VM: CMS Callable Services Reference*, SC24-6165
- *z/VM: CMS Macros and Functions Reference*, SC24-6168

- *z/VM: CP Programming Services*, SC24-6179
- *z/VM: CPI Communications User's Guide*, SC24-6180
- *z/VM: Enterprise Systems Architecture/ Extended Configuration Principles of Operation*, SC24-6192
- *z/VM: Language Environment User's Guide*, SC24-6199
- *z/VM: OpenExtensions Advanced Application Programming Tools*, SC24-6202
- *z/VM: OpenExtensions Callable Services Reference*, SC24-6203
- *z/VM: OpenExtensions Commands Reference*, SC24-6204
- *z/VM: OpenExtensions POSIX Conformance Document*, GC24-6205
- *z/VM: OpenExtensions User's Guide*, SC24-6206
- *z/VM: Program Management Binder for CMS*, SC24-6211
- *z/VM: Reusable Server Kernel Programmer's Guide and Reference*, SC24-6220
- *z/VM: REXX/VM Reference*, SC24-6221
- *z/VM: REXX/VM User's Guide*, SC24-6222
- *z/VM: Systems Management Application Programming*, SC24-6234
- *z/VM: TCP/IP Programmer's Reference*, SC24-6239
- *Common Programming Interface Communications Reference*, SC26-4399
- *Common Programming Interface Resource Recovery Reference*, SC31-6821
- *z/OS: IBM Tivoli Directory Server Plug-in Reference for z/OS*, SA76-0148
- *z/OS: Language Environment Concepts Guide*, SA22-7567
- *z/OS: Language Environment Debugging Guide*, GA22-7560
- *z/OS: Language Environment Programming Guide*, SA22-7561
- *z/OS: Language Environment Programming Reference*, SA22-7562
- *z/OS: Language Environment Run-Time Messages*, SA22-7566
- *z/OS: Language Environment Writing ILC Applications*, SA22-7563
- *z/OS MVS Program Management: Advanced Facilities*, SA22-7644
- *z/OS MVS Program Management: User's Guide and Reference*, SA22-7643

Diagnosis

- *z/VM: CMS and REXX/VM Messages and Codes*, GC24-6161
- *z/VM: CP Messages and Codes*, GC24-6177
- *z/VM: Diagnosis Guide*, GC24-6187
- *z/VM: Dump Viewing Facility*, GC24-6191
- *z/VM: Other Components Messages and Codes*, GC24-6207
- *z/VM: TCP/IP Diagnosis Guide*, GC24-6235
- *z/VM: TCP/IP Messages and Codes*, GC24-6237
- *z/VM: VM Dump Tool*, GC24-6242
- *z/OS and z/VM: Hardware Configuration Definition Messages*, SC33-7986

z/VM Facilities and Features

Data Facility Storage Management Subsystem for VM

- *z/VM: DFSMS/VM Customization*, SC24-6181
- *z/VM: DFSMS/VM Diagnosis Guide*, GC24-6182
- *z/VM: DFSMS/VM Messages and Codes*, GC24-6183
- *z/VM: DFSMS/VM Planning Guide*, SC24-6184
- *z/VM: DFSMS/VM Removable Media Services*, SC24-6185
- *z/VM: DFSMS/VM Storage Administration*, SC24-6186

Directory Maintenance Facility for z/VM

- *z/VM: Directory Maintenance Facility Commands Reference*, SC24-6188
- *z/VM: Directory Maintenance Facility Messages*, GC24-6189
- *z/VM: Directory Maintenance Facility Tailoring and Administration Guide*, SC24-6190

Open Systems Adapter/Support Facility

- *System z10, System z9 and eServer zSeries: Open Systems Adapter-Express Customer's Guide and Reference*, SA22-7935
- *System z9 and eServer zSeries 890 and 990: Open Systems Adapter-Express Integrated Console Controller User's Guide*, SA22-7990

- *System z: Open Systems Adapter-Express Integrated Console Controller 3215 Support*, SA23-2247

Performance Toolkit for VM™

- *z/VM: Performance Toolkit Guide*, SC24-6209
- *z/VM: Performance Toolkit Reference*, SC24-6210

RACF® Security Server for z/VM

- *z/VM: RACF Security Server Auditor's Guide*, SC24-6212
- *z/VM: RACF Security Server Command Language Reference*, SC24-6213
- *z/VM: RACF Security Server Diagnosis Guide*, GC24-6214
- *z/VM: RACF Security Server General User's Guide*, SC24-6215
- *z/VM: RACF Security Server Macros and Interfaces*, SC24-6216
- *z/VM: RACF Security Server Messages and Codes*, GC24-6217
- *z/VM: RACF Security Server Security Administrator's Guide*, SC24-6218
- *z/VM: RACF Security Server System Programmer's Guide*, SC24-6219
- *z/VM: Security Server RACROUTE Macro Reference*, SC24-6231

Remote Spooling Communications Subsystem Networking for z/VM

- *z/VM: RSCS Networking Diagnosis*, GC24-6223
- *z/VM: RSCS Networking Exit Customization*, SC24-6224
- *z/VM: RSCS Networking Messages and Codes*, GC24-6225
- *z/VM: RSCS Networking Operation and Use*, SC24-6226
- *z/VM: RSCS Networking Planning and Configuration*, SC24-6227
- *Network Job Entry: Formats and Protocols*, SA22-7539

Prerequisite Products

Device Support Facilities

- *Device Support Facilities: User's Guide and Reference*, GC35-0033

Environmental Record Editing and Printing Program

- *Environmental Record Editing and Printing Program (EREP): Reference*, GC35-0152
- *Environmental Record Editing and Printing Program (EREP): User's Guide*, GC35-0151

REXX Compiler

- *IBM Compiler and Library for REXX on zSeries: Diagnosis Guide*, SH19-8179
- *IBM Compiler and Library for REXX on zSeries: User's Guide and Reference*, SH19-8160

Index

Program Number: 5741-A07

Printed in USA

SC24-6222-00