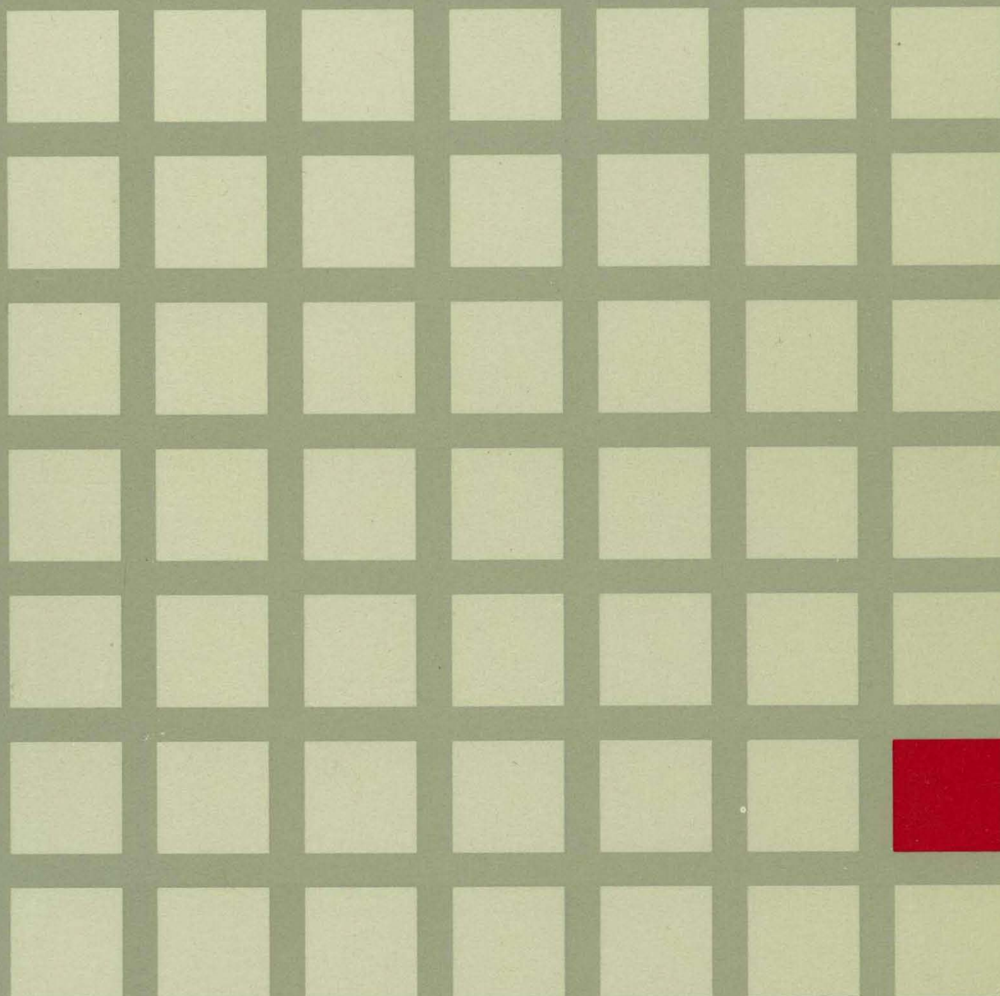
Virtual Machine/System Product

SC24-5238-04

System Product Interpreter User's Guide

Release 6



Virtual Machine/System Product

SC24-5238-04

System Product Interpreter User's Guide

Release 6

| **Fifth Edition (July 1988)**

| This edition, SC24-5238-04, is a major revision of SC24-5238-03, and applies to Release 6 of the IBM Virtual Machine/System Product (5664-167) unless otherwise indicated in new editions or Technical Newsletters. Changes are periodically made to the information contained herein; before using this publication in connection with the operation of IBM systems, consult the latest *IBM System/370, 30xx, 4300, and 9370 Processors Bibliography*, GC20-0001, for the editions that are applicable and current.

Summary of Changes

For a detailed list of changes, see "Summary of Changes" on page 229.

Changes or additions to the text and illustrations are indicated by a vertical line to the left of the change.

In this manual are illustrations in which names are used. These names are fanciful and fictitious; they are used solely for illustrative purposes and not for identification of any person or company.

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM licensed program in this publication is not intended to state or imply that only IBM's licensed program may be used. Any functionally equivalent program may be used instead.

Ordering Publications

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

A form for reader's comments is provided at the back of this publication; if the form has been removed, comments may be addressed to IBM Corporation, Information Development, Department G60, P.O. Box 6, Endicott, NY, U.S.A. 13760. IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you.

© Copyright International Business Machines Corporation 1983, 1984, 1986, 1988. All rights reserved.

Preface

If you would like to be able to write programs, then this book is for you. You will need a terminal with access to Virtual Machine/System Product (VM/SP), and you should be reasonably familiar with VM/SP, but you need not have had any previous experience of programming.

The programming language described by this book is called the Restructured Extended Executor language (sometimes abbreviated REXX). The book also describes how the System Product Interpreter (shortened, hereafter, to the interpreter) processes or *interprets* the Restructured Extended Executor language.

This book is a step-by-step guide that will help you learn REXX in three levels. You are advised to read through this book in three passes, focusing each time on different material.

On your first reading, you will learn fundamental concepts, and be invited to write a dozen or so small programs of your own.

On your second reading, you will get to know the main body of the language, and be shown practical programs that you can copy or modify.

The third reading deals with more difficult tasks; about the more sophisticated features of the language that you may need to use occasionally.

On each reading, you will find that your path through the book is clearly marked with headings and reminders.

Chapter 1: Introduction — discusses how to use this publication and introduces you to the Restructured Extended Executor (REXX) language.

Chapter 2: How Your Program Is Interpreted — describes the rules of syntax and substitution in REXX programs.

Chapter 3: Variables — provides you with information on naming, using, and assigning variables. It also shows you how to build a collection of variables and create variables that are unique to your programs.

Chapter 4: Expressions — shows how to write expressions that the interpreter can compute (operators, true or false, arithmetic, comparisons, etc..).

Chapter 5: Conversations — shows how to write lines to a user's screen, and how to get data that has been entered on the command line and use it in your program. The chapter also describes parsing of options, variables, and other data.

Chapter 6: Commands — describes how to issue CMS and CP commands from your program. You will learn how to pass information back and forth between command environments and your program.

Chapter 7: File Processing — provides you with examples of REXX programs that modify CMS files by reading and writing lines. The chapter also describes how to use REXX to process a file to produce a new file.

Chapter 8: XEDIT — introduces ways in which you can tailor XEDIT though the use of REXX programs.

Chapter 9: Control — describes how to determine the course that your program will take. You can select one of a number of lists, repeat a list of instructions, continue from a different point in the same file, pass control to a subroutine and return, and exit from the program.

Chapter 10: Programming Style and Techniques — introduces the method you should use to construct and design your REXX programs. The section includes various problems and solutions.

For information about writing REXX programs in the GCS environment, see the *VM/SP System Product Interpreter Reference*.

Contents

Chapter 1. Introduction . . . 1
What is REXX? . . . 1
  Features of REXX . . . 1
  REXX and the Systems Application Architecture™ (SAA) . . . 2
How to Use This Book . . . 2
  Before You Start . . . 2
  The Reading Plan . . . 3
Conversations . . . 4
  Translation to Uppercase . . . 5
  The PARSE PULL Instruction . . . 5
  Writing a Program . . . 5
  Running a Program . . . 6
  Did You Understand That? . . . 6

Chapter 2. How Your Program Is Interpreted . . . 9
Rules . . . 9
  Comments in Programs . . . 9
  Comments with Special Meaning to CMS . . . 10
  Strings . . . 10
  Clauses . . . 10
  When Does a Clause End? . . . 11
  Tidying Up . . . 12
  Syntax Errors . . . 12
  Did You Understand That? . . . 13
  Substitution Rules . . . 15
Repeated Substitution . . . 16
  The VALUE( ) Function . . . 16
  Compound Symbols . . . 16
  The INTERPRET Instruction . . . 17

Chapter 3. Variables . . . 19
Assignments . . . 19
  Choosing Names for Variables . . . 20
  Example: Setting Variables . . . 20
  Did You Understand That? . . . 21
Compound Symbols . . . 22
  Using Compound Symbols . . . 22
  Stems . . . 24
  Did You Understand That? . . . 25
Avoiding Duplicate Names . . . 27
How Much Should You Tell Your Subroutine? . . . 29
  The PROCEDURE Instruction . . . 29
  The PROCEDURE EXPOSE Instruction . . . 30
The Existence of Variable Names . . . 31
  The SYMBOL( ) Function . . . 31
  The DROP Instruction . . . 31
Arrays with More Than One Dimension . . . 32

Chapter 4. Expressions . . . 35
Operators . . . 36
  Operators and Terms . . . 36

Order of Evaluation 37
Parentheses 37
Did You Understand That? 37
Tracing 39
Data Types 40
Prefix Operators 41
Priority of Operators 41
Using Parentheses 42
Did You Understand That? 42
True and False 44
Comparisons 44
Using True and False 44
The Equal Sign (=) 45
The AND (&) Operator 45
The OR (|) Operator 45
Did You Understand That? 45
Logical Operators 47
Did You Understand That? 47
Functions 49
The Idea of a Function 49
Built-in Functions 50
User-Written Functions 50
Did You Understand That? 50
User-Written Functions 52
ARG Instruction 52
The ARG( ) Function 52
RETURN Instruction 52
Did You Understand That? 53
A Square Root Function 56
Internal Functions 57
Functions Written in S/370 Assembler Language 58
Loops 59
The DO Instruction 59
A "DO UNTIL" Loop 59
Getting Out of Loops 60
Did You Understand That? 61
Arithmetic 63
Numbers 63
Checking Your Input 64
Addition, Subtraction, Multiplication 65
Division 65
Range of Numbers 67
Exponential Notation 67
Did You Understand That? 68
Formatting Numeric Output 69
Specifying Conventional (Fixed Point) Notation 70
Specifying Exponential (Floating Point) Notation 71
A Special Case 71
Did You Understand That? 71
Exponentiation 73
The NUMERIC DIGITS Instruction 73
The SIGN( ) Function 74
Rounding and Truncation 74
Did You Understand That? 75
Groups of Instructions 77

Text 78
  Concatenation 78
  The SUBSTR( ) Function 78
  The LENGTH( ) Function 79
  The COPIES( ) Function 79
  The LEFT( ) Function 79
  Arranging Your Output in Columns 80
  Did You Understand That? 80
  Using a Subroutine to Simplify Tabulation 82
  The POS( ) Function 83
  Example 84
  Words 85
  The WORDPOS( ) Function 85
  Providing Help 86
  Did You Understand That? 87
  The OVERLAY( ) Function 90
  The WORDS( ) and WORD( ) Functions 91
Comparisons 93
  General 93
  Numbers 93
  Characters 93
  Did You Understand That? 94
  The COMPARE( ) Function 95
  The ABBREV( ) Function 95
  Did You Understand That? 96
  Exact Comparisons 97
  Fuzzy Arithmetical Comparisons 97
Translation 99
  Hexadecimal 100
  Conversion 100
  Character Sets 103
  The VERIFY( ) Function 103

Chapter 5. Conversations 105
The SAY Instruction 105
The PULL Instruction 106
  The UPPER Instruction 106
  Did You Understand That? 107
  Parsing Words 109
  The Period as a Placeholder 110
  Did You Understand That? 110
Getting Data from the Command Line 112
  Mixed Case 113
  Recognizing Options 113
  Literal Patterns 113
  Parsing Variables and Expressions 114
  Did You Understand That? 115
  Parsing Using Patterns 117

Chapter 6. Commands 119
Issuing Commands to CMS and CP 119
  Clauses That Become Commands 119
  When to Use Quotes 121
  CP Commands 122
  Summary 122

Return Codes . . . 123
Special Variables . . . 124
Did You Understand That? . . . 125
Debugging Individual Commands . . . 126
Debugging Execs That Contain Commands . . . 126
Making a Common Routine for Handling Return Codes . . . 126
Getting Messages from a Repository File . . . 127
How to Suppress Messages Issued by CMS Commands . . . 128
A Useful Subroutine . . . 129
Did You Understand That? . . . 130
Using the Program Stack . . . 131
Definitions . . . 132
Buffers . . . 133
How to Use the Program Stack . . . 134
Example: A CMS Command That Puts Data onto the Program Stack . . . 136
Example: A CMS Command That Requires Data from the Program Stack 137
CP Commands . . . 138
How to Suppress Messages Issued by CP Commands . . . 138
How to Obtain the Reply from a CP Command . . . 138
The COMMAND Environment . . . 141

Chapter 7. File Processing . . . 143
Writing Files . . . 143
Example: An Editor . . . 144
Reading Files . . . 145
End of File . . . 145
Example: To Display a File on the Screen . . . 146
The FINIS Command . . . 146
Example: A Time Recording Program . . . 147
Did You Understand That? . . . 149
Using Data From an Existing File to Create a New File . . . 152
Example: Processing a File to Produce a New File . . . 152
Precautions To Be Taken When Modifying an Existing File . . . 154
Other Techniques . . . 156
An Example: Sorting a File . . . 156
Processing Files in the CMS Shared File System . . . 159

Chapter 8. XEDIT . . . 161
XEDIT Subcommands and Macros . . . 161
XEDIT Macros . . . 162
Naming of XEDIT Macros . . . 162
Example: Changing the Settings of the Scroll Keys . . . 162
Return Codes . . . 163
Messages . . . 164
The EXTRACT Subcommand . . . 165
The Current Line . . . 165
An Example: Moving through a File a Paragraph at a Time . . . 166
Your XEDIT Profile . . . 166
Menus Using XEDIT . . . 168

Chapter 9. Control . . . 171
Selection . . . 172
The IF Instruction . . . 173
The ELSE Keyword . . . 174
The "Dangling" ELSE . . . 176

Did You Understand That? 177
The SELECT Instruction 178
Example 180
The NOP Instruction 181
Did You Understand That? 181
Loops 184
Simple Repetitive Loops 184
Using a Control Variable 186
The BY Expression 187
Did You Understand That? 187
Conditional Loops: The LEAVE Instruction 190
Conditional Loops: The DO WHILE Instruction 191
Conditional Loops: The DO UNTIL Instruction 192
Conditional Loops: The Choice 193
Did You Understand That? 194
Compound DO Instructions 196
Leaving a Specified Loop 196
The ITERATE Instruction 198
The EXIT Instruction 200
Subroutines 201
The Idea of a Subroutine 201
The CALL Instruction 203
The ARG Instruction 205
The RETURN Instruction 205
Example 205
When to Leave Out the Arguments 206
Did You Understand That? 207
Subroutines and Functions 209
Using a Call of the Other Kind 210
Parsing the Arguments 210
External Subroutines 211
Jumps 212
The SIGNAL Instruction 212
The SIGNAL Instruction 213
SIGNAL ON Condition 214

Chapter 10. Programming Style and Techniques 215
Consider the Data 215
Did You Understand That? 217
Happy Hour 217
Designing a Program 220
Methods for Designing Loops 220
The Conclusion 220
What Do We Have So Far? 221
Stepwise Refinement: An Example 222
Consider the Data 222
Correcting Your Program 223
Modifying Your Program 223
Tracing Your Program 223
Coding Style 224

Summary of Changes 229

Glossary of Terms and Abbreviations 231

Bibliography 237
Related Publications 237

Index 241

Chapter 1. Introduction

In this chapter we discuss:

- What is REXX?
- Features of REXX.
- REXX and the Systems Application Architecture.
- How to use this book.
- Conversations in REXX.

What is REXX?

The REstructured eXecutor eXtended language, or REXX, is a programming language that is extremely versatile. Features such as ease of use and free format make it a good language for beginners and general users. REXX is also suited for more serious computer professionals because of its ability to issue commands to different environments, its powerful functions, and its extensive mathematical capabilities.

REXX is an adaptation of 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/1. There are a number of differences, but the main difference is that a REXX program is interpreted (the interpreter operates on the program directly as it executes). In PL/1, the program is compiled (translated into machine language) first, then executed.

Features of REXX

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

Free format: There are few rules about REXX format. You need not start an instruction in a particular column, you can also skip spaces in a line or skip entire lines, you can have an instruction span many lines or have multiple instructions on one line, variables do not need to be pre-defined, and you can type instructions in upper, lower, or mixed case.

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

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

Interpreted: The REXX language is an interpreted language. When a REXX exec runs, the language processor directly interprets each language statement. Languages that are not interpreted must be compiled into machine language before they are run.

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

REXX and the Systems Application Architecture™ (SAA)

REXX is one of the programming languages supported by the Systems Application Architecture (SAA) to help provide cross-system consistency. Programs written in REXX according to the SAA specifications can be transported to other SAA environments. For example, a REXX exec in CMS can also run in a TSO/E environment, if the exec does not use system-specific functions or commands.

How to Use This Book

If you are searching for particular topics, look in the Preface, the Table of Contents, or the Index at the back of the book. If you are going to use this book as a self-study text, read on.

Before You Start

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

- We expect you to know something about CMS. For instance, you should know what a CMS file is, how to create a file using an editor, and some of the ways that you can use CMS commands to manipulate files. You should also know about the additional file management and file sharing functions available through the CMS Shared File System (SFS). If you are not familiar with CMS or SFS, read the *VM/SP CMS Primer* first.
- A complete description of REXX can be found in the *VM/SP System Product Interpreter Reference*. You will need to have your own copy of this on hand, so you can look up any instruction or function that is not completely defined in this book.
- If you have little or no experience in computer programming or programming in REXX, it may be worthwhile for you to read *VM/IS Writing Simple Programs with REXX*. This book is an excellent introduction to REXX and can help you get started in programming, regardless of the system you are working with.
- Another related publication that may be useful to more experienced REXX users is the *SAA Common Programming Interface Procedures Language Reference*. This book defines the SAA Procedures Language, which is a subset of the VM/SP System Product Interpreter, and REXX. Descriptions include use and syntax of the language as well as explanations on how the language processor interprets a program as it is executing.

---

* Systems Application Architecture is a trademark of the IBM Corporation.

- You may occasionally need to refer to the other IBM manuals that are mentioned at the end of the book (see "Related Publications" on page 237).
- Quite early on, we shall invite you to try entering small programs from your terminal. If you do not already have a VM user ID and logon password, you should arrange to get them as soon as possible.
- If you are using REXX in the GCS environment, see the *VM/SP System Product Interpreter Reference*. Included is a description of the differences between writing REXX programs for the GCS environment and writing REXX programs for the CMS environment.

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 the System Product Interpreter. 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 heading (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 **bold type** reminders at the beginning and end of each reading. These reminders will tell you where a particular reading begins and ends and where you should go next. Following is an example.

Reading 1

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

Reading Steps: Reading 1 and Reading 2 are divided into steps to help you measure your progress. They also give you places to stop and take breaks. The steps are another way to pace yourself through the self-study. Reading 1 consists of steps 1 through 23 and Reading 2 consists of steps 24 through 37.

Conversations

Reading 1 begins here. It is also the beginning of step 1.

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 shows a sample REXX program. The sample program asks the user to give his name, and then responds to him by name. For instance, if the user types in the name `Joe`, the reply `Hello JOE` is displayed. Or else, if the user does not type anything in, the reply `Hello stranger` is displayed.

First, we shall discuss how it works; then you can try it out for yourself.

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

Figure 1. HELLO EXEC

Briefly, the various pieces of the sample program are:

`/* ... */`	A comment explaining what the program is about. All REXX programs must start with a comment. This distinguishes them from CMS EXEC and EXEC 2 language programs. Apart from this, comments are ignored.
`say`	An instruction to display `Hello! What's your name?` on the screen.
`pull`	An instruction to read the response entered from the keyboard and put it into the computer's memory.
`who`	The name given to the place in memory where the user's response is put.
`if`	An instruction that asks a question.
`who = ""`	A test to determine if `WHO` is empty.
`then`	A direction to execute the instruction that follows, if the tested condition is true.
`say`	An instruction to display `Hello stranger` on the screen.

else	An alternative direction to execute the instruction that follows, if the tested condition is not true.
say	An instruction to display `Hello`, followed by whatever is in `WHO` on the screen.

Translation to Uppercase

You may have noticed in this example that the letters:

a, b, c, ... z

sometimes get changed to:

A, B, C, ... Z

This is called *translating to uppercase*. Anything in your program that is not in quotes gets translated to uppercase (for example, the names of places in the computer's memory).

Anything in quotes is not translated to uppercase; it is left *as is*.

The PARSE PULL Instruction

There are two forms of the PARSE PULL instruction

PARSE PULL Reads everything from the keyboard *as is*, without translating it to uppercase.

PARSE UPPER PULL
Translates everything to uppercase as it is read from the keyboard. This form of the instruction may be abbreviated to **PULL**.

In Figure 1, PULL is used. This is the same as PARSE UPPER PULL. You will see when you run the example that whatever you type in is translated to uppercase.

Writing 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.

To write this program, 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 VM/SP System Product Editor.

The name of the program is HELLO EXEC (for now, assume that the filetype must be exec). Log on to VM/SP and type the command:

```
xedit hello exec
```

Type in the program, exactly as it is shown in Figure 1, beginning with `/* 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.

Reading 1

Running a Program

If you want to run a program that has a filetype of exec, you just type in its filename. In this case, type `hello` on the command line and press ENTER. Try it!

This is what happened when Fred tried it.

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

The PULL instruction paused, waiting for a reply. Fred typed `fred` on the command line and, when he pressed the ENTER key, the PULL instruction put the word FRED into the place in the computer's memory called `WHO`. The IF instruction asked, is `WHO` equal to nothing:

```
WHO = ""
```

meaning, is the value of `WHO` (in this case, FRED) equal to nothing:

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

This was not true; so, the instruction after `then` was not executed; but the instruction after `else`, was.

And this is what happened when Mike tried it:

```
hello
Hello! What's your name?

Hello stranger
Ready;
```

Mike did not understand that he had to type in his name. Perhaps the program should have made it clearer to him. Anyhow, he just pressed ENTER. The PULL instruction put "" (nothing) into the place in the computer's memory called `WHO`. The IF instruction asked, is:

```
WHO = ""
```

meaning, is the value of `WHO` equal to nothing:

```
"" = ""
```

In this case, it was true. So, the instruction after `then` was executed; but the instruction after `else` was not.

Did You Understand That?

1. Did you get your version of HELLO EXEC to run on your VM/SP system? If not, check that you have correctly typed it in. If it still does not work and you can not 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.

You have just completed Step 1.

Reading 1 continues in Chapter 2, "How Your Program Is Interpreted" on page 9.

Reading 1

Chapter 2. How Your Program Is Interpreted

In this chapter:

Reading 1 **immediately following, describes:**

- Comments
- Strings (in quotes)
- Clauses
- Blanks
- Lowercase characters (a..z).

Reading 2 **on page 15, describes:**

- Substitution rules.

Reading 3 **on page 16, describes:**

- Repeated substitution using the

  The VALUE( ) function
  Compound Symbols
  The INTERPRET instruction.

Rules

Reading 1

The System Product Interpreter works on your REXX program, line by line and word by word, doing what you have written. If you know the rules for interpreting a REXX program, it will make it easier for you to write programs that will correctly execute. The rules are quite simple if you learn them in the right order.

The first few rules are explained in this chapter. We start with the rule for comments.

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 interpreter 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 VM/SP. The oldest is called CMS EXEC; the next is EXEC 2; and the latest is REXX. For technical reasons, they all have a filetype 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.

Strings

When the interpreter sees a quote (either " or ') it stops interpreting and just goes along looking for the matching quote. The *string* of characters inside the quotes is used just as it is. Examples of strings are:

'Hello'
"Final result:"

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

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

There is another way. Within a string, a pair of quotes (of the same kind as was used to delimit 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"')

Clauses

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

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

   ```
   say "the word"
   ```

   In this case, the interpreter 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 memory.

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 19).

3. A *label*, which is a name followed by a colon; for example,

   ```
   MYSUB:
   ```

   (Labels are discussed in "The CALL Instruction" on page 203 and "The SIGNAL Instruction" on page 212).

4. A *null clause*, such as a completely blank line, or

   ```
   ;
   ```

   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 119).

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 can not, 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?

```
/* 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 2. RAH EXEC

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

Reading 1

Tidying Up

When a line is being interpreted, everything that is not in quotes is translated to uppercase. In other words the letters

a, b, c, ... z

get changed to

A, B, C, ... Z

The interpreter 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 quotes. Figure 3 shows an example.

```
/* Example: cases and spaces */
say  a        long         story

say "A        long         story"

say about"    "a dog
```

Figure 3. SHAGGY EXEC

Enter shaggy on the command line and press ENTER to call this EXEC. Here is what appears on your screen:

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

Syntax Errors

The rules governing the arrangement of words and punctuation marks in a language are called the *syntax* of the language. The rules we have been discussing are part of the syntax for interpreting REXX.

If the interpreter finds something that does not make sense, it stops running your program and displays the bad line, followed by an error message saying what is wrong. Suppose we alter our first program to read:

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

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

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

- 3 +++ means the interpreter 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 help, look up `Error 6` in the list of error messages at the back of your *VM/SP System Product Interpreter Reference.*
- `Ready(20006);` is the return code that the interpreter returns to CMS.

Leaving out a final quotation mark at the end of a string would cause the interpreter to issue a similar REXX error message.

Did You Understand That?

1. Read the following program carefully. Take a pencil and write down what each word is and what the interpreter will do with it.

```
MADAM EXEC

/* Polite enquiry */
Jane = "Mrs. Doe"
say "How" is jane ?
```

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

2. This next program has an error in it! Type the program in and run it. Notice what the return code is.

```
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 *VM/SP System Product Interpreter Reference.* Correct the error and test the program again.

Answers:

1. The syntax of this program is:

   - `/* Polite enquiry */` is a *comment* describing the program. (The first line of a REXX program must start with a comment.)
   - `Jane = "Mrs. Doe"` is an *assignment*. The variable `JANE` gets the value 'Mrs. Doe'.
   - `say` is a REXX *instruction*. The rest of the line is interpreted and the result is displayed.
     - `"How"` is a *string*. It is displayed just as it is: `How`
     - `is` is changed to uppercase. Because it is not the name of a variable, it appears as: `IS`
     - `JANE` is the name of a variable. The value 'Mrs. Doe' is substituted.
     - `?` is a valid character. Because it is not the name of a variable, it is displayed as: ?

   What actually appears on the screen is:

```
madam
How IS Mrs. Doe ?
```

2. The error number is 37. To get advice on how to correct the error, use your *VM/SP System Product Interpreter Reference*.

   Note: You can also use the HELP command to obtain this advice. Use XEDIT to add to your PROFILE EXEC A the line:

   `CP SET EMSG ON`

   File your PROFILE EXEC A and execute the command PROFILE. Now when you get a syntax error, it will be prefixed by a 10-character error code:

   `DMSREX473E`

   Use the first three and the last four characters or the entire 10-characters of the error code to obtain the HELP panel. In this example you would use the command:

   `HELP DMS473E` or `HELP DMSREX473E`

You have just completed Step 2.

Reading 1 continues in Chapter 3, "Variables" on page 19.

Reading 2 begins here. It is also the beginning of step 24.

If you would like to review Reading 1 of this section, read "Conversations" on page 4 and Chapter 2, "How Your Program Is Interpreted" on page 9.

If you wish to start Reading 2, continue on.

Substitution Rules

When replacing the names of variables with their values, the interpreter 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 next ("Compound Symbols" on page 22), can be used to provide a further level of substitution.

Reading 2 continues in Chapter 3, "Variables" on page 19.

Repeated Substitution

Reading 3 begin 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 15 could be redesigned like this:

```
/* 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 4. 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.

```
/* 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 5. 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 your *VM/SP System Product Interpreter Reference.*)

Here is an example:

```
/*  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 6. 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 19.

Reading 3

Chapter 3. Variables

In this chapter:

Reading 1 immediately following, describes:

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

Reading 2 on page 22, describes:

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

Reading 3 on page 29, 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.

Assignments

Reading 1

A basic requirement of programs is that they handle variety — variations in input data and variations in their own results from processing. They do this by using variables. A *variable* is one or more characters that represent a value. This value can change during the execution of a program, but the variable name must stay the same.

In an *assignment*, you name a variable and give it a value. An assignment can be used to assign an initial value to a variable or to change its value.

- To give the variable called SOMETHING whatever value the user will type on the command line, use the PULL instruction.

  ```
  pull something
  ```

- To give the variable called TOTAL the value '0', use the assignment:

  ```
  total = 0
  ```

- 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
  ```

More generally, a REXX clause of the form

```
symbol = expression
```

is called an assignment. The interpreter evaluates (computes) what the expression is and puts the result into the variable called symbol. We say

"Assign the expression's value to the symbol".

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 interpreter 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 22, 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 this program:

```
/* 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 7. 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 the ENTER key, 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 178.)

`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 interpreter finds a symbol (a word that is not in quotes) 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 interpreter 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 7) 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.

Did You Understand That?

1. Which of the following could be used as the name of a REXX variable?
   a. DOG
   b. K9
   c. 9T
   d. nine_to_five
   e. #7

Answers:

1.
   a. OK
   b. OK
   c. Invalid, because the first character is a numeric digit.
   d. OK, same as NINE_TO_FIVE
   e. OK

You have just completed Step 3.

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

Compound Symbols

Reading 2

Compound symbols can be used for building collections of variables.

For example:

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

/* so that, if we know what day it is, we know what     */
/* gift will be given.  Suppose, for example that       */

day = 3

/* then */

say gift.day                /* says 'Three French hens' */
```

The symbol GIFT.DAY is recognized as compound because it contains a period. The characters following the period may be the name of a variable. If the variable exists, its value is substituted in the name to give a derived name. The derived name is then used as the name of the variable to be processed. In the example, because DAY equals 3 the derived name of GIFT.DAY is GIFT.3.

If DAY had never been given a value, its value would have been 'DAY', and the derived name of GIFT.DAY would have been GIFT.DAY.

Using Compound Symbols

The example in Figure 8 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. A collection of consecutively numbered variables like this is sometimes called an *array*. Thus, using compound symbols, the array of SCOREs is processed to give the result in the required form.

```
/* 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 8 (Part 1 of 2). GAME EXEC

```
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
say
```

Figure 8 (Part 2 of 2). GAME EXEC

Stems

To refer to all the variables in an array, use their stem. A *stem* is a symbol that contains only one period, which is its ending character.

For example:

```
player. = 0
say player.1 player.2 player.golf       /* Says '0 0 0'  */
```

It is often convenient to set all variables in an array to zero in this way.

Did You Understand That?

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"
do 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
```

You have just completed Step 24.

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 help to show to how the values of the variables LINE and DATA change, during execution.

```
/* 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 9. 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:

```
1
ENTER A 1 OF DATA
melvin
MELVIN
DATA MELVIN
11
DONE
```

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 quotes.

  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 quotes, 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 interpreter can not 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

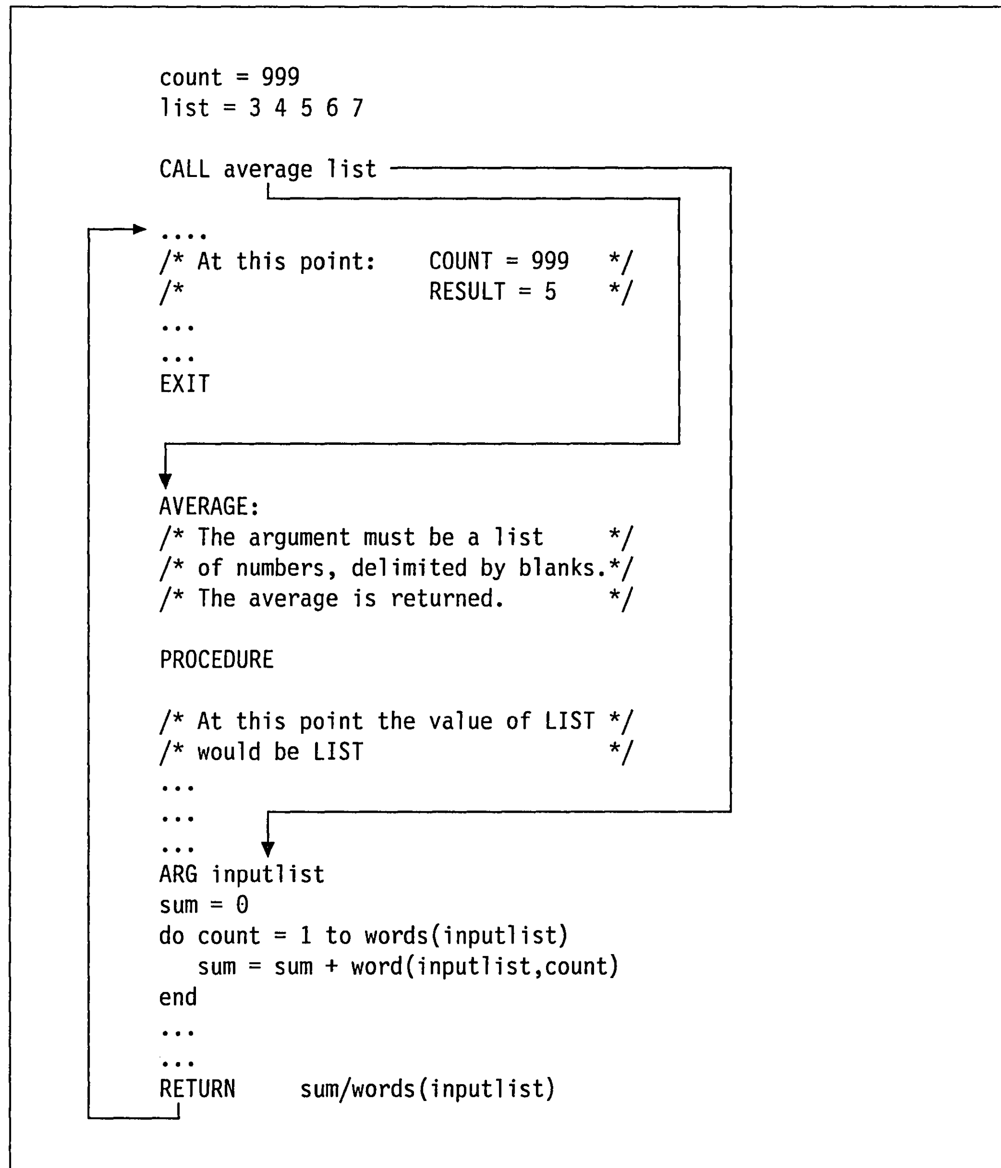
To make the interpreter forget, for the time being, all the variables it knows, use the PROCEDURE instruction.

After this instruction has been executed, 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 executed.

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 once. 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 *VM/SP System Product Interpreter Reference.*

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

```
PROCEDURE EXPOSE name [name] [name]...
```

where:

`name` is the name of a variable to be shared. For further details, see the PROCEDURE instruction in your *VM/SP System Product Interpreter Reference.*

The Existence of Variable Names

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 quotes.

```
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 quotes CASH is treated as a variable, and its value is substituted before the function is performed.

Finally, an example:

```
/* 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 10. TICKETS EXEC

The DROP Instruction

Usually, the place where you want the interpreter 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 interpreter to forget about a variable altogether. In this case, use the DROP instruction.

```
DROP name [name] [name]...
```

where:

name   is name of a variable to be dropped.

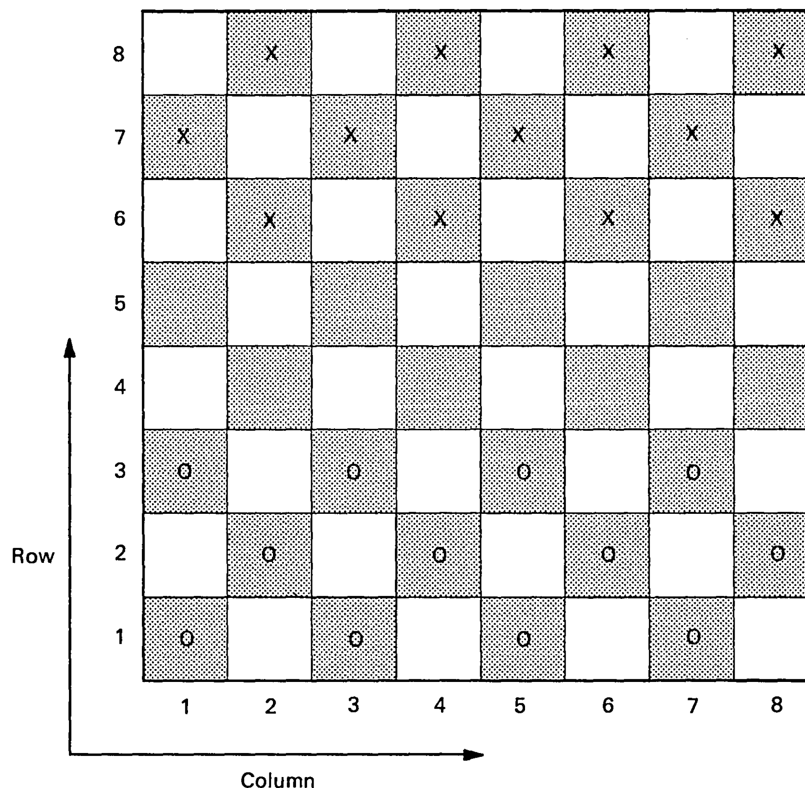
You can drop more than one variable using a single DROP instruction. You can also drop all the elements of an array by specifying the stem of the array. For example:

```
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 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.



```
/* This program simulates 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"
end
do col = 2 by 2 to 8
   board.2.col = "r"
end
do col = 1 by 2 to 7
   board.3.col = "r"
end
do col = 2 by 2 to 8
   board.6.col = "b"
end
do col = 1 by 2 to 7
   board.7.col = "b"
end
do col = 2 by 2 to 8
   board.8.col = "b"
end
```

Figure 11. CHECKERS EXEC

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

Reading 3

Chapter 4. Expressions

An *expression* is something that can be computed. In your *VM/SP System Product Interpreter 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 interpreter can compute. The rules that the interpreter 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[1]
- Arithmetic
- Groups of instructions[1]
- 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.

[1] 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 184, Groups of Instructions in "Selection" on page 172).

Operators

In this section:

Reading 1 **immediately following, describes:**

- Operators and Terms
- Order of evaluation
- Parentheses.

Reading 2 **on page 39, 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 49.

Operators and Terms

Reading 1

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

+	Add
*	Multiply
||	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 quotes, 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,

```
10 -  3  +  2
└─────┘
   │
   │
   7     +  2
  └──────────┘
        │
        │
        9
```

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 (-).

```
10 -  3  *  2
      └─────┘
         │
10 -     6
└────────┘
     │
     4
```

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

We shall discuss the rules of priority again in Reading 2 on page 41.

Parentheses

When the interpreter 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 49.

Did You Understand That?

1. 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'              */
```

a. 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"
```

b. 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.

   a. There are 5 people in this family.
   b. It is easy to count up to 10.

You have just completed Step 4.

Reading 1 continues in "True and False" on page 44.

Tracing

Reading 2

To find out how the interpreter 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:

```
/* 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 12. 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 12, 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:

```
/* 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 13. 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.

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.)

```
/* 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 14. 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 vice versa.

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 interpreter 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 your *VM/SP System Product Interpreter 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 your *VM/SP System Product Interpreter Reference.*

Did You Understand That?

1. What is the value of:

   a. 4 + 20 “tailors”
   b. 24 = 4 + 20
   c. “eggs” = “eggs” & 2*2 = 4
   d. 3 / 2*5
   e. 3 || 7+7
   f. 3(2+2)
   g. (2+2)3.

Answers:

1.

   a. 24 tailors (add before concatenate)
   b. 1 (add before comparison)
   c. 1 (comparison before AND, multiply before AND, comparison before AND)
   d. 7.5 (operators that have the same priority are processed left to right)
   e. 314 (add before concatenate)
   f. calls the function “3” with the argument “4” (or gives a syntax error if “3” does not exist)
   g. 43 (evaluate expression in parentheses first; then abut).

You have just completed Step 25.

Reading 2 continues in “True and False” on page 44.

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 47, 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 your *VM/SP System Product Interpreter 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."

Did You Understand That?

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'.

You have just completed Step 5.

Reading 1 continues in "Functions" on page 49.

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'         */
```

Did You Understand That?

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."

Answers:

1.
   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)
   ```

You have just completed Step 26.

Reading 2 continues in "Functions" on page 49.

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 function.

Reading 2 **on page 52, describes:**

- Writing your own functions
  - The ARG instruction and the ARG function
  - The RETURN instruction.

Reading 3 **on page 56, describes:**

- Including your own functions in the exec file of the program that uses them
- Functions written in S/370 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, on page 54.)

Quite generally, if the interpreter 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 interpreter 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,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 119.    */
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 *VM/SP System Product Interpreter 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.

Did You Understand That?

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.

You have just completed Step 6.

Reading 1 continues in "Loops" on page 59.

Reading 2

User-Written 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 myarg1, myarg2 ...
```

where:

`myarg1, myarg2, ...` 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 myarg1, myarg2 ...
```

The ARG( ) Function

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

```
ARG(n)
```

In this way you can refer to the nth argument.

RETURN Instruction

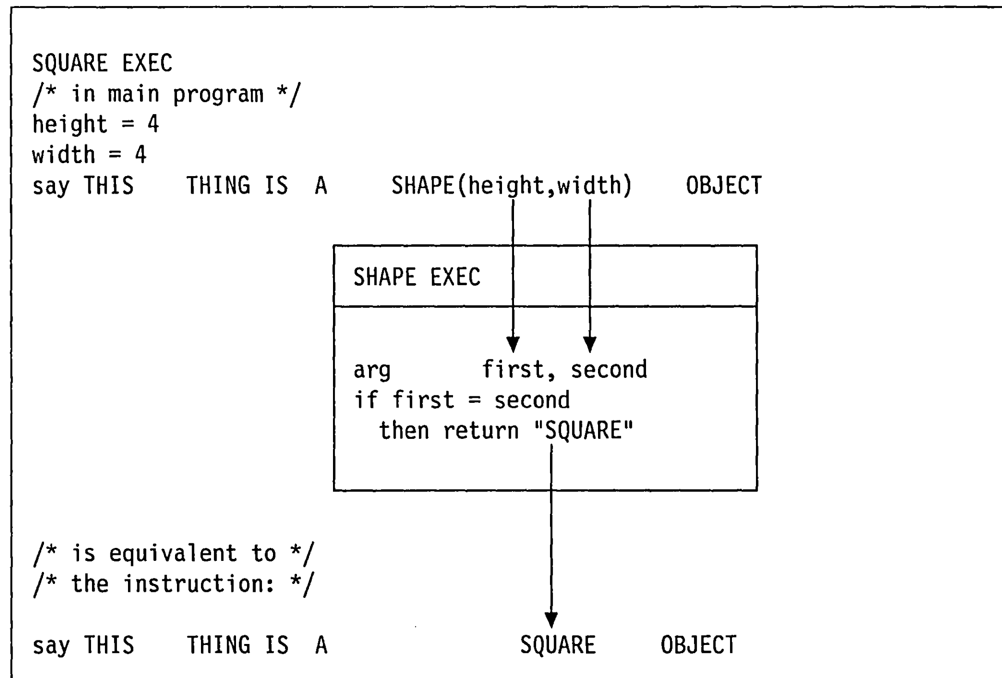
To use the result from a function call, the data must be returned from the function call to the main program. To return the result, use the following instruction:

```
RETURN expression
```

The interpreter 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.



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 invalid.

For example, you can write:

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

When the function is called with invalid 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 *VM/SP System Product Interpreter Reference* for other ways of using this function.

Did You Understand That?

Here is the specification and code for the HALF( ) function that we discussed on page 49.

```
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 */
```

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 */
Error 45 running HALF EXEC, line 18: No data specified on function RETURN
     6 +++ say half(4.5)
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 */
Error 45 running HALF2 EXEC, line 3: No data specified on function RETURN
    2 +++ say "Testing" half2(5,7)
Error 40 running TESTHAL2 EXEC, line 2: Incorrect call to routine
Ready(20040);
```

You have just completed Step 27.

Reading 2 continues in "Arithmetic" on page 63.

Reading 3

A Square Root Function

Reading 3

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

```
/* The SQUARE ROOT function                              */
/*                                                       */
/*                  SQRT(number)                         */
/*                                                       */
/* where "number" is a nonnegative REXX number,          */
/* returns the square root of "number".  Precision is 9 */
/* significant figures, independent of the setting of   */
/* NUMERIC DIGITS (explained on page 73).                */
/*                                                       */
/* Implementation details:  "number" is normalized to   */
/* give an even exponent (so that the exponent can be   */
/* dealt with separately later) and a mantissa between  */
/* 1 and 100.  The most significant digit of the result */
/* is found.                                             */
/*                                                       */
/* The mantissa is multiplied by 100, the exponent is   */
/* reduced by 2 to compensate, the partial result       */
/* (ROOT) is multiplied by 10, (leaving a zero in the   */
/* units position) and the units digit of this partial  */
/* result is then found.  And so on.                     */
/*                                                       */
/* Finally, the result is adjusted using the computed   */
/* exponent.                                             */

/*-----------------------------------------------------*/
/* Set precision                                         */
/*-----------------------------------------------------*/
numeric digits 10              /* for partial results   */
                               /* use one digit more    */
                               /* than final precision  */
/*-----------------------------------------------------*/
/* Check arguments                                       */
/*-----------------------------------------------------*/
if arg() ¬= 1
   then return                 /* wrong number arguments */
arg x
if ¬ datatype(x,number)
   then return                 /* argument not a number  */
if x < 0
   then return                 /* argument is negative   */

                                      /* continued ...   */
```

Figure 15 (Part 1 of 2). SQRT EXEC

```
/*-----------------------------------------------------*/
/* Normalize:                                          */
/* FROM                                                */
/*    x      the argument                              */
/* COMPUTE                                             */
/*    mant   the mantissa, where 0 < mant < 100        */
/*    exp    the exponent, where exp is even           */
/*-----------------------------------------------------*/
                        /* Format so that 0 < mant < 10 */
parse value format(x,,,,0) with  mant "E" exp

                        /* Modify so that exp is even   */
if exp = "" then exp = 0
if exp//2 ¬= 0 then do
   mant = mant * 10
   exp = exp - 1
end

/*-----------------------------------------------------*/
/* Find root by successive approximation               */
/*-----------------------------------------------------*/
root = 0
do 10
  do digit = 9 by -1 to 0,      /* find largest digit,  */
     while,                     /* such that            */
        (root + digit)**2 > mant /* (root+digit)squared  */
  end                           /*         is ¬>  mant */
  root = root + digit
  if root**2 = mant then leave
  root = root * 10
  mant = mant * 100
  exp = exp - 2
end

/*-----------------------------------------------------*/
/* Adjust for computed exponent                        */
/*-----------------------------------------------------*/
numeric digits 9
return root * 10**(exp/2)
```

Figure 15 (Part 2 of 2). 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 29).

```
/* 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 onwards, the code   */
           /* is the same as that shown in  */
           /* SQRT EXEC on page 56.         */
```

Figure 16. ROOTS EXEC

Functions Written in S/370 Assembler Language

A further improvement in performance can be obtained by writing your function in System/370 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 your *VM/SP System Product Interpreter Reference* for more information.

Reading 3 continues in "Arithmetic" on page 63.

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.

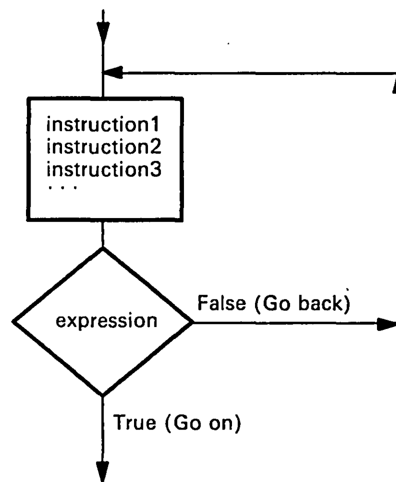
The DO Instruction

To build loops, you should use the REXX instruction DO. This is described fully in a later section, "Loops" on page 184.

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 interpreter reaches the keyword `END`; if the result is '0', the interpreter loops back to `instruction1`. Otherwise, execution continues with the instruction following the `END` instruction.

For example, the program in Figure 17 will go on asking the same question until the user answers "12."

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

Figure 17. DOZEN EXEC

Getting Out of Loops

This program will never finish.

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

Figure 18. NEVER EXEC

You can recognize this situation because, when you type in another command, CMS does not execute it. If by any chance you find that you are running such a program, 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 19 is nearly impossible to get out of if you do not know what the answer is.

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

Figure 19. 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.

Did You Understand That?

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!"
```

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
```

You have just completed Step 7.

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

Reading 1 continues in "Arithmetic" on page 63.

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 69, describes:

- Formatting numeric output
- Specifying conventional and exponential notation.

Reading 3

on page 73, describes:

- Using the ** operator to compute the nth 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 interpreter 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 wanted the user to keep trying until entering a valid number you could write:

```
/* 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 20. 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
```

(& is the AND operator. See "The AND (&) Operator" on page 45.)

By the way, the DATATYPE( ) function can test for other types of data, as well. See the DATATYPE function in your *VM/SP System Product Interpreter 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:

```
/* 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 21. SHARE EXEC

You should be careful not to try to divide by zero. If you do, a syntax error will result. That is why, in Figure 21, 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 interpreter works out the result correct to nine digits if necessary. This means nine significant digits, not counting the zeroes 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: '0.000000000233333333' */
```

The accuracy of computed results can be changed using the NUMERIC DIGITS instruction. This instruction is described in "The NUMERIC DIGITS Instruction" on page 73.

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 zeroes. 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 interpreter 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 interpreter 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.

Did You Understand That?

1. 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
```

Answers:

1. 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 2 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'.

You have just completed Step 8.

Reading 1 continues in "Groups of Instructions" on page 77.

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:

```
/* 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 22. INVOICE EXEC

It displays the data formatted like this:

```
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
```

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
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:

```
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 your *VM/SP System Product Interpreter 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.

Did You Understand That?

1.

   a. Write an exec called REFORMAT that expresses numbers entered by the user in both fixed point and exponential notation.

   b. 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. 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
```

b. The following table lists the results you should get when using the test numbers with 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	1200000000000	1.2E+10
1.2	1.2	1.2
1.2e+0	1.2	1.2

You have just completed Step 28.

Reading 2 continues in "Text" on page 78.

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:

```
/* examples of numbers with unusually high precision */

numeric digits 10
say "The largest signed number that can be held"
say "in a S/370 register is" 2**31 - 1 "exactly."
say

numeric digits 48
say "1/7 =" 1/7
```

Figure 23. ACCURATE EXEC

The sample program results in the display of:

```
accurate
The largest signed number that can be held
in a S/370 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 NUMBERIC DIGITS:

```
DIGITS()
```

would return "9" since 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 in your *VM/SP System Product Interpreter Reference.*

When your program performs a series of arithmetical operations, you may inadvertently introduce additional errors. Look at the fourth item in INVOICE EXEC on page 69. 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
```

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;
```

Did You Understand That?

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.
   a. True
   b. No. The ** operator has a higher priority than the / operator, so the interpreter 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 56.

Reading 3 continues in "Text" on page 78.

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 once (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" on page 78.

Reading 1

Text

In this section:

Reading 1 immediately following, describes:

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

Reading 2 on page 82, 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 90, describes:

- How to use the OVERLAY( ) function.

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 your *VM/SP System Product Interpreter 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 filename"
pull fn .                   /* The period ensures that   */
                            /* FN is assigned only one   */
                            /* word.                     */
if length(fn) > 8
then
   do                       /* A group.  See page 77.    */
      fn = substr(fn,1,8)
      say "The filename 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|'    */
```

Arranging Your Output in Columns

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

```
/* 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 24. 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;
```

Did You Understand That?

Given that:

C = "Continent"

1. What is the value of:
   a. C "of America"
   b. C || "al"
   c. C"al"
   d. LENGTH("Continent")
   e. LENGTH(C)
   f. LENGTH("C")
   g. Substr(c,1,4)substr(c,7,3)
   h. Substr(c,1,2)substr(c,5,2)
   i. LEFT("Q",8)"QUERY"
   j. LEFT("COPY",8)"COPYFILE"

Answers:

1.

   a. Continent of America
   b. Continental
   c. Continental
   d. 9
   e. 9
   f. 1
   g. Content
   h. Coin

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

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

   i. `Q       QUERY`
   j. `COPY    COPYFILE`

You have just completed Step 9.

Reading 1 continues in "Comparisons" on page 93.

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 25 shows how a subroutine can be used several times in order to create a table.

For example:

```
/* 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 25. TABLE2 EXEC

The output will be the same as TABLE1 on page 80.

For the CALL instructions in Figure 25, 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:

```
/* 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                      */
 ... (same as TABLE2 EXEC in Figure 25)
```

Figure 26. 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 your *VM/SP System Product Interpreter 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.

```
/* VALIDATE FILENAME                                    */
/* 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
   good = 1
   Say "Enter filename"
   pull fn .
   if length(fn) > 8 then do           /* length      */
      say "Filename must not be more",
          "than 8 characters long"
      good = 0
   end
   componentname = substr(fn,1,3)      /* 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
   end

                                     /*continued ...*/
```

Figure 27 (Part 1 of 2). VALIDFN EXEC

```
   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
   end
   author = substr(fn,6)                /* author      */
   if ¬ datatype(author,upper)
   then do
      say "Sixth and remaining characters",
          "must be alphabetic"
      good = 0
   end
   if good = 0 then say "Try again"
end
```

Figure 27 (Part 2 of 2). 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 your *VM/SP System Product Interpreter Reference.*

(Also consider the PULL, ARG and PARSE instructions, discussed later on pages 106 through 115).

The WORDPOS( ) Function

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

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

The arguments are:

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

The interpreter 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:

```
/* "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 28. 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 nth line of the source file. If n is omitted, SOURCELINE( ) returns the *line number* of the final line in the source file.

```
/*
This program processes the input file to give ...
 ... ...

Correct format is:

       MYPROG filename filetype [filemode]

Function performed is:
Rhubarb, rhubarb, rhubarb.
*/
say "Enter fileid 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 29. MYPROG EXEC

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

Did You Understand That?

1. 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:

1. 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 87)  */
/*-----------------------------------------------------*/
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;
```

You have just completed Step 29.

Reading 2 continues in "Comparisons" on page 93.

Reading 3

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 overlayed
- The `target` onto which it is to be overlayed
- The position in the `target` where overlaying is to start
- The number of bytes to be overlayed.

For example:

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

(For a complete definition, see your *VM/SP System Product Interpreter Reference.*)

Here is a useful example.

```
/* 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 key 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 30. 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 filetype) that matches one of a given list of words.

```
/* 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 filetype.  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 filetype, XEDIT will    */
/* use the filetype 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 31 (Part 1 of 2). XE EXEC

```
types = "SCRIPT EXEC PLIOPT DOC LISTING"
/*-----------------------------------------------------*/
/* check arguments                                     */
/*-----------------------------------------------------*/
arg filename filetype filemode "(" options
                                    /* Coding note:    */
                                    /* see page 113    */
if filename = "" | filename = "?"   /* Help needed     */
then do
   do line = 1 while substr(sourceline(line),1,2) = "/*"
      say sourceline(line)
   end
   exit
end
/*-----------------------------------------------------*/
/* compute filetype                                    */
/*-----------------------------------------------------*/
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 196    */
         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 31 (Part 2 of 2). XE EXEC

Reading 3 continues in "Comparisons" on page 93.

Comparisons

In this section:

Reading 1 immediately following, describes:

- Comparing numbers
- Comparing character strings.

Reading 2 on page 95, describes:

- Finding the first character that does not match
- Comparing data without regard to case
- Recognizing abbreviations.

Reading 3 on page 97, 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 your *VM/SP System Product Interpreter Reference*.)

Numbers

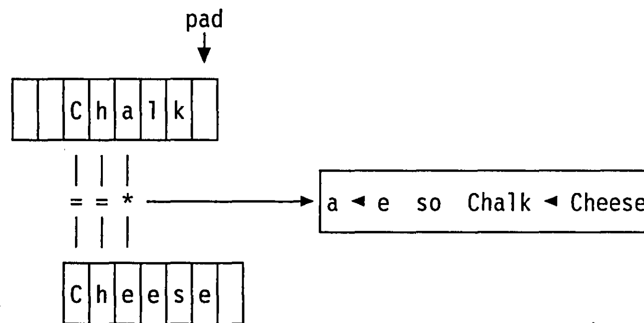
If both the terms being compared are numbers, comparison is numeric, rather than character by character.

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 are used to 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:

(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 on page 90 to discover the sequence of characters for your keyboard.

Did You Understand That?

1. What is the value of each of the following expressions?

   a. "3" > "five"
   b. "Kilogram" > "kilogram"
   c. "a" > "#"
   d. "q" > "?"
   e. "9a" > "9"
   f. "?" > " "

Answers:

1. All are "1" (true).

You have just completed Step 10.

Reading 1 continues in "Translation" on page 99.

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,

```
/* 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 32. YEP EXEC

Did You Understand That?

Given that:

Q2 = "COPY"
Q3 = "PRT"

1. What is the value of:
   a. COMPARE(SUBSTR(Q2,3),Q3)
   b. ABBREV("COPYFILE",Q2,4)
   c. ABBREV("PRINT",Q3,2).

Answers:

1.
   a. 2
   b. 1
   c. 0 ("PRT" is not equal to the first 3 letters of "PRINT".)

You have just completed Step 30.

Reading 2 continues in "Translation" on page 99.

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 since 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 94)

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 zeroes and so on.

For more information on exact comparison operators, see your *VM/SP System Product Interpreter Reference*

Fuzzy Arithmetical Comparisons

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

```
/* 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 33. NOFUZZ EXEC

To make comparisons less accurate than ordinary REXX arithmetic, use the NUMERIC FUZZ instruction. (For full details, see your *VM/SP System Product Interpreter Reference.*)

For example:

```
/* 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 34. FUZZ EXEC

To check the current setting of the NUMERIC FUZZ instruction use the ( ) 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" on page 99.

Translation

In System/370, 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 105.

Reading 2 **on page 100, describes:**

- Conversion between Character, Hexadecimal and Decimal.

Reading 3 **on page 103, describes:**

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

Reading 2

Hexadecimal

Reading 2

In System/370, 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 interpreter will accept strings expressed in either character or hexadecimal form. To indicate that a string is expressed in hex, write the letter "X" next to the closing quote.

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, lets 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.

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 100, 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 105.

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.

```
/* 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 35. 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 *VM/SP System Product Interpreter Reference*.

The VERIFY( ) Function

To find out whether a string contains only characters of a 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:

```
/* 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 "Invalid serial number.  Please start again"
```

Figure 36. DIGITS EXEC

Reading 3 continues in Chapter 5, "Conversations" on page 105.

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 112, 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 117, 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 quotes or the interpreter 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.

```
/* 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 37. 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 name1 name2 ...
```

For example, this might have been used in WHATDAY EXEC, on page 62, to let the user reply in mixed case.

```
/* 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" date(weekday)
end
say "Correct!"
```

Figure 38. WHATDAY2 EXEC

Did You Understand That?

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
```

   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 typing "quit". But you *can* get out by typing "QUIT".

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:

```
noan
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 interpreter displays the line that caused the error and the return code.

You have just completed Step 11.

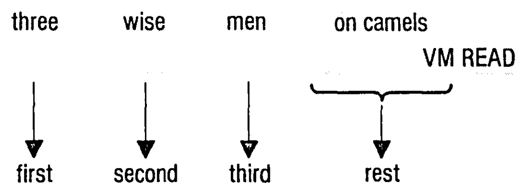
Parsing Words

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.

```
say "Please enter three or more words:"
pull first second third rest
...
```

The user's screen: Please enter three or more words:

command line = = >



As usual, the program pauses and the user can type something on the command line. When the user presses ENTER, the program continues.

The variable `FIRST` is given the value "THREE."
The variable `SECOND` is given the value "WISE."
The variable `THIRD` is given the value "MEN."
The variable `REST` is given the value "   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.

```
/* 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 39. 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.

Did You Understand That?

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
```

You have just completed Step 12.

Reading 1 continues in Chapter 6, "Commands" on page 119.

Getting Data from the Command Line

Reading 2

When you want to run your exec, type its filename on the command line. This can be followed by more data, called arguments.

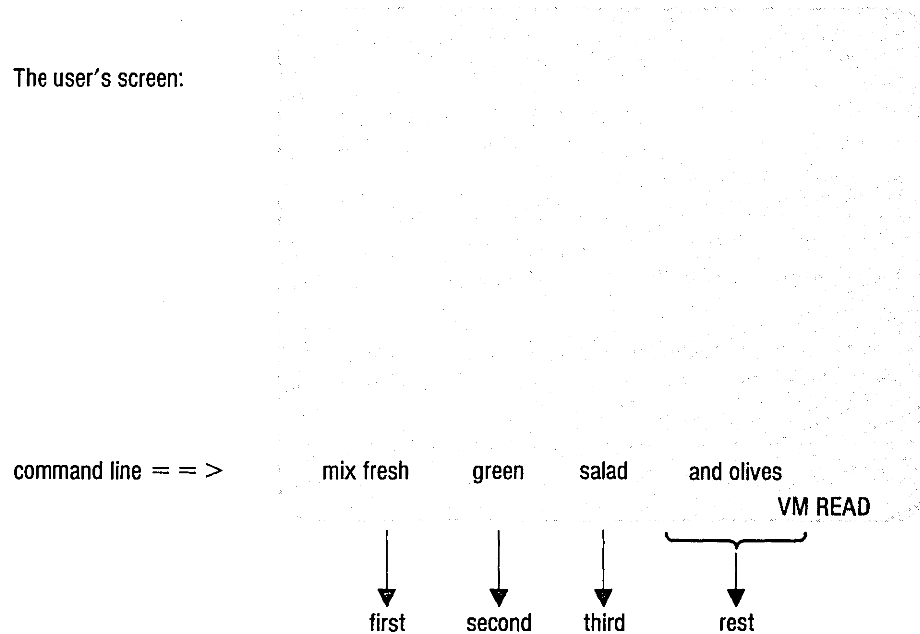
To obtain the data that the user entered on the command line when starting your program, use the ARG instruction. ARG will parse the arguments in the same way that PULL parses data from the keyboard, except that the first word entered on the command line (the name of the exec) is not parsed. (The ARG instruction gives the same results as the PARSE UPPER ARG instruction.)

If there is a program called MIX EXEC, the command shown on the following command line will start it. For example,

```
/* 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 40. MIX EXEC



When the ARG instruction is executed:

The variable `FIRST` is given the value "FRESH."
The variable `SECOND` is given the value "GREEN."
The variable `THIRD` is given the value "SALAD."
The variable `REST` is given the value " 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 *literal patterns.*

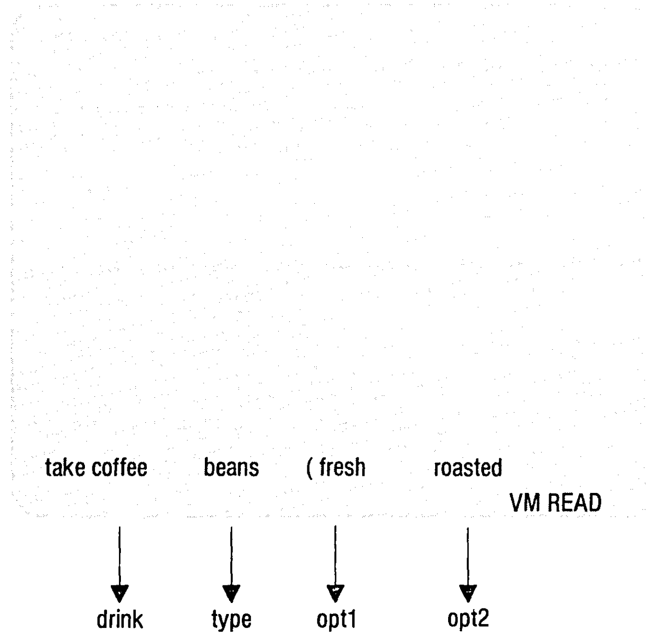
Literal Patterns

To split up the data being parsed, use literal patterns. If your PARSE instruction specifies a string (that is, one or more characters enclosed in quotes) 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 is used to parse the data from the command line. If there is a program called TAKE EXEC, the command shown on the following command line will start it.

```
/* Example: recognizing options */

arg drink type shelf  "("  opt1 opt2 opt3 ")" rest
say drink type shelf opt1 opt2 opt3 rest
```

Figure 41. TAKE EXEC



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 literal 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 VAR symbol arg1 arg2 arg3 ...
PARSE VALUE expression WITH arg1 arg2 arg3 ...
```

For example:

```
/* 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 42. PARSING EXEC

Did You Understand That?

1.

a. Modify MYPROG EXEC on page 87 to use the ARG instruction.

b. Make a further modification to test for a CONTINUE option. Allow any abbreviation of COntinue that is two or more letters long. Test for invalid options.

Answers:

1. 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"
```

When run, the following is displayed:

```
myprog2
This program processes the input file to give ...
 ... ...

Correct format is:

   MYPROG2 filename filetype [filemode] [(COntinue [)]]

Function performed is:
Rhubarb, rhubarb, rhubarb.
Ready;
```

You have just completed Step 31.

Reading 2 continues in Chapter 6, "Commands" on page 119.

Parsing Using Patterns

Reading 3

The idea of parsing using patterns is fully explained in your *VM/SP System Product Interpreter 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 quotes, like "(" and ")" in the MYPROG2 EXEC on page 116
- 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.

```
/* 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 prefixed      */
/* to "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 43. CHANGE EXEC

Reading 3 continues in Chapter 6, "Commands" on page 119.

Reading 3

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 126, 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 138, 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 interpreter can operate in a number of *environments* (for example, CMS or XEDIT). The way the interpreter 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 interpreter 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 interpreter 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.

```
/* Example: to erase a number of SCRIPT files.     */

do until fn = ""
   say "Enter filename 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 44. ERASER EXEC

- If you want to use a REXX operator or special character as an ordinary character, then you must put it in quotes. 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 quotes.

  For example:

```
/* Example: to erase all the files on filemode A    */
/* that have a filetype of LIST                     */

"ERASE * LIST"        /* This clause is treated as  */
                      /* an expression.  The result */
                      /*   'ERASE * LIST'           */
                      /* is passed to CMS           */
```

Figure 45. ELIST EXEC

  In Figure 45, if the asterisk was not in quotes, the interpreter would attempt to multiply `ERASE` by `LIST`!

  Note: Remember to put quotes around all operators and parentheses unless already enclosed in quotes. Either of the following examples is correct:

  `"COPY" MYFILE SCRIPT A "=" BACKUP A "(REPLACE"`

  `"COPY MYFILE SCRIPT A = BACKUP A (REPLACE"`

- Another difficulty is the use of duplicate names. In Figure 46, the programmer has chosen A as the name of a variable. In the COPYFILE instruction, A is used as the filemode and must be enclosed in quotes; otherwise, the current value of A would be substituted.

```
/* Example: to save copies of a number of SCRIPT    */
/* files.  Each copy is given the same filename     */
/* as the original, and a filetype of BACKUP.       */

do until a = ""
   say "Enter filename 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
   "COPY" a "SCRIPT A = BACKUP A (REP"
             /* This clause is treated as an        */
             /* expression.  The result, which in   */
             /* this example is                     */
             /* COPY MYFILE SCRIPT A = BACKUP A (REP */
             /* is passed to CMS                    */
end
```

Figure 46. BACKUP EXEC

This example leads on to a more general question.

When to Use Quotes

The syntax for REXX expressions is very flexible. If a symbol, that is not the name of a variable, is written without quotes, no error is signalled. 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 46, A is the name of a variable, so it must not be used as the literal name of a filemode without putting quotes 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 quotes. In the example BACKUP EXEC in Figure 46, the COPYFILE command would be written:

```
"COPY" a "SCRIPT A = BACKUP A (REP"

/* Here, everything is in quotes 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 [TO] userid hisdisk mydisk [mode] [password]
```

where:

userid	is the user ID of the person the disk belongs to.
hisdisk	is the 3-digit number of his disk.
mydisk	is a 3-digit number 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 47.

(For an introduction to this subject, see "LINK" in the *VM/SP CMS Primer*. For full details, see the *VM/SP CP General User Command 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.

```
/* For linking to Disk 196 belonging to HELPDESK       */

"LINK HELPDESK 196 200"       /* a CP command          */
"ACCESS 200 B"                /* a CMS command         */
```

Figure 47. 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.

You have just completed Step 13.

Return Codes

When you write a CMS or CP command in your exec, you should consider what would happen if the command failed to execute correctly. For example, a COPYFILE command might fail because the user's disk was full. After such a failure, 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 a failure. 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:

```
copy profile exec a profile backup a

  Ready;

link fred 591 591

  FRED 591 not linked; not in CP directory
  Ready(00053);

access 591 b

  DMSACC113S B (591) not attached
  Ready(00100);

copy profile exec a = = b (for luck

  Invalid parameter LUCK in the FOR option Field.
  Ready(00024);

erase junk exec

  File JUNK EXEC not found.
  Ready(00028);
```

The first command worked correctly so the return code was zero and CMS displayed the "Ready" message:

```
Ready;
```

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.

```
Ready(00053);
```

Now that you understand how CMS handles commands and return codes, let us see how the interpreter handles them.

Any command that would be valid on the CMS command line is valid as a clause in a REXX program. The interpreter treats the clause like any other expression, substituting the values of variables, and so on. The interpreter 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 your *VM/SP System Product Interpreter Reference*.)

When the interpreter has issued a command and CMS or CP has finished executing it, the interpreter 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:

```
"COPY 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 interpreter gives control back to CMS. This will be explained later in "The EXIT Instruction" on page 200.

To find out what return codes can be expected from a CMS command, look up the command in the *VM/SP CMS Command Reference*. Return codes are listed in the last paragraph of the description of each command.

To find out what return codes can be expected from CP commands, look in the *VM/SP System Messages Cross-Reference*. The commands themselves are described in the the *VM/SP CP General User Command 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 interpreter. For example, the special variable RC is assigned a new

value when a command has been executed. (For full details, see your *VM/SP System Product Interpreter Reference*.)

Did You Understand That?

1. 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 filemode A (a directory in your file space or a R/W minidisk) whose names begin with "PR".

   a. Study the CMS command LISTFILE. You will find it in the *VM/SP CMS Command Reference*, or you can get a short description displayed on your screen by typing in "HELP LISTFILE". Use the LISTFILE command to display the required list of files on your screen.

   b. Study the EXEC option of the LISTFILE command. Write a REXX program that issues a command to generate the required file.

   c. At the end of the description of LISTFILE in the *VM/SP CMS Command Reference*, you will find a list of possible return codes. Modify your program to handle all possible errors.

   d. Add to your program a command that RENAMEs the file that has been created as "PR ALL A".

   e. Test your program by running it twice.

Answers:

1.

```
LISTPR EXEC


/* Lists all the files on filemode A whose filenames    */
/* 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"
```

You have just completed Step 14.

Reading 1 continues in Chapter 7, "File Processing" on page 143.

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 afterwards 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 interpreter 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 interpreter that tests the return code from every command. If a nonzero return code is detected, the usual sequence of clauses is abandoned. Instead, the interpreter 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.

You can use SIGL to tell the user which command caused normal processing to be interrupted:

```
...
signal on error
COPY ... ...

"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 one of these messages from a REXX EXEC. See the *VM/SP CMS Command Reference* for a complete description of XMITMSG.

When using XMITMSG in a REXX EXEC, variables are enclosed in quotes. 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 quotes  */
                         /* 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 quotes.        */
end
```

Note: This is not a complete program and can not 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 normal output, use the Resume Typing command.

```
SET CMSTYPE RT
```

Be sure that your program executes SET CMSTYPE RT before you need to execute 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).

```
"SET CMSTYPE HT"
"STATE" fn ft fm          /* Does the file exist?          */
rcsave = rc
"SET CMSTYPE RT"          /* (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
"SET CMSTYPE HT"
''arg(1)                 /* Coding note: the null string */
                         /* prevents ARG from being      */
                         /* 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 "at line" sigl"."
```

Note: This is not a complete program and can not be executed by itself.

Did You Understand That?

1. Review the following program. Make sure that you understand what it is supposed to do. Will it always work correctly?

```
/* This program requests the user to supply a list of    */
/* files (filename filetype only) and replies, for       */
/* each file:                                            */
/*                                                       */
/*  * whether it is on the user's directory or minidisk */
/*    accessed as filemode A.                            */
/*                                                       */
/*  * whether it is on the directory or minidisk         */
/*    accessed as filemode L.                            */
/*                                                       */
/*  * if there is a copy on each filemode, 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 substr(line,1,2) ¬= "/*"
   line = sourceline(n)
   say line
end
else do forever
   do until ft ¬= "" & rest = ""    /* Get fn ft         */
      say "Enter filename and filetype",
          "(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 filemodes*/
   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 filemode "home")"
      otherwise
      call quiet compare fn ft A fn ft L

                                    /* continued ...     */
```

Figure 48 (Part 1 of 2). PAIRS EXEC

```
      select
         when result = 0
         then say "Same file found on both filemodes",
                  "(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 filemodes A and L",
                  "are not the same"
         otherwise say "Unexpected return code" result,
                       "from COMPARE command"
      end
   end
end
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 48 (Part 2 of 2). PAIRS EXEC

Answers:

1. The program will run correctly.

You have just completed Step 32.

Using the Program Stack

The *program stack* is used to pass 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:

  EXECIO — to read files from a directory or minidisk and to execute CP commands and execute CP commands
  IDENTIFY — to obtain the nodeid, rscsid, and so on

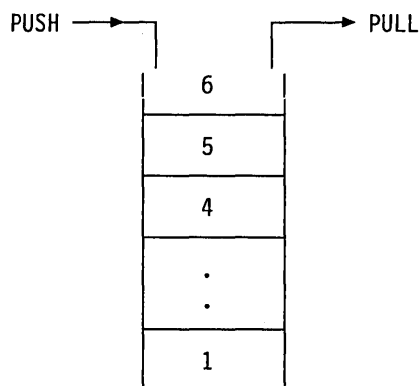
LISTDIR	to find out about directories
LISTFILE	to find out about files
NAMEFIND	to obtain information from a NAMES file
QUERY	to find out about your CMS virtual machine
RECEIVE	to read in files and notes
RDR	to find out what files are in your reader.

- And finally, an example of a command that takes data from the program stack. Some commands that can do this are:

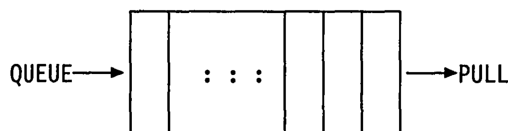
EXECIO	to write files to a directory or minidisk
COPYFILE	to copy files (using the SPECS option)
FORMAT	to format a minidisk
SORT	to sort a file.

Definitions

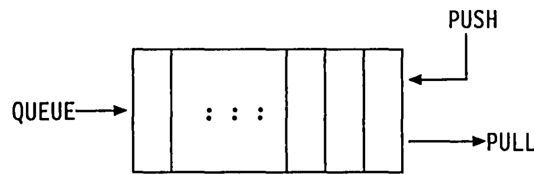
In computer science, a *stack* is a list of items that you can work with from only one end, the top. You can PUSH an item onto the stack or PULL an item off from it. The item you PULL off will always be the last item you (or somebody else) PUSHed on.



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.



The CMS program stack can be used both as a stack and as a queue.



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, 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.

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
	CMSSTACK FIFO	CMSSTACK LIFO	LINERD
Assembler macro	(none)	(none)	RDTERM

where:

FIFO means First In, First Out (as in a queue).

LIFO means Last In, First Out (as in a stack).

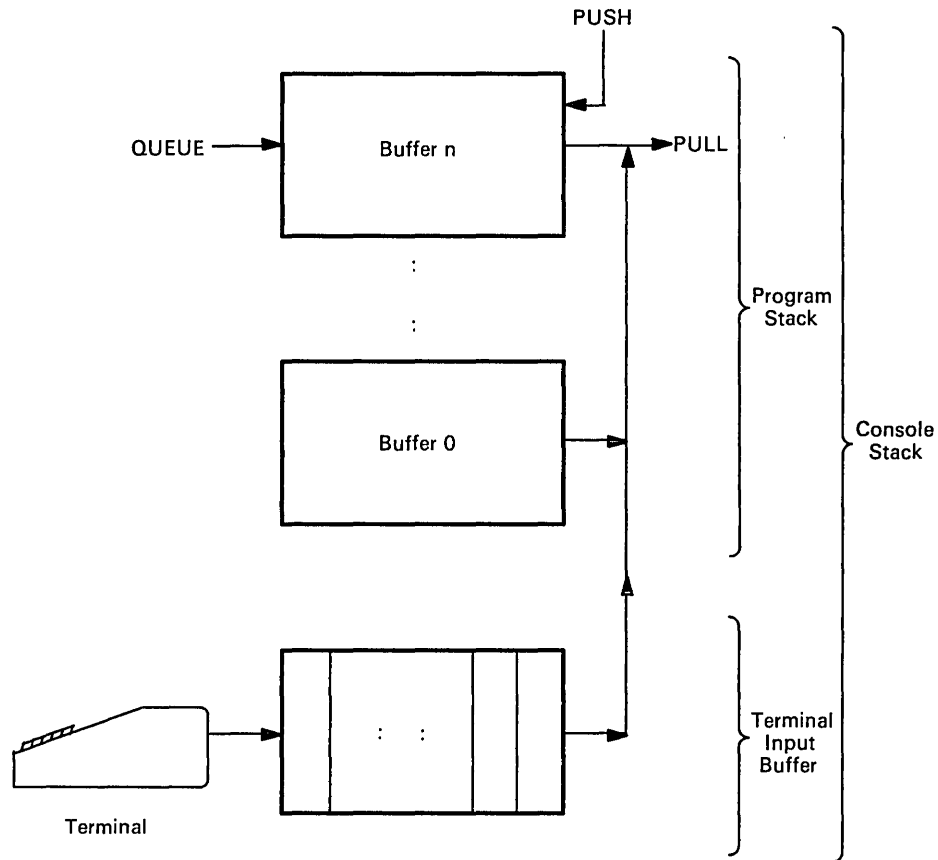
Buffers

A *buffer* is a general term for a part of the computer's memory 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 the Enter key, 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, use 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, the user must press ENTER.

   Note: Lines originating from SFS and non-SFS sources might contain different information. For example, the LISTFILE command displays information about the contents of a specific filemode, which can be an accessed minidisk or an accessed directory. If the filemode is a minidisk, the command displays lines that describe files. However, if the filemode is a directory, the command might also display information about subdirectories. A line that describes a subdirectory is different from the line that describes a file.

5. It is important to avoid removing items that do not belong to you from 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, filemode A, is more than 80% full. This means that it's time to get a bigger minidisk, or else erase some files you'll 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.

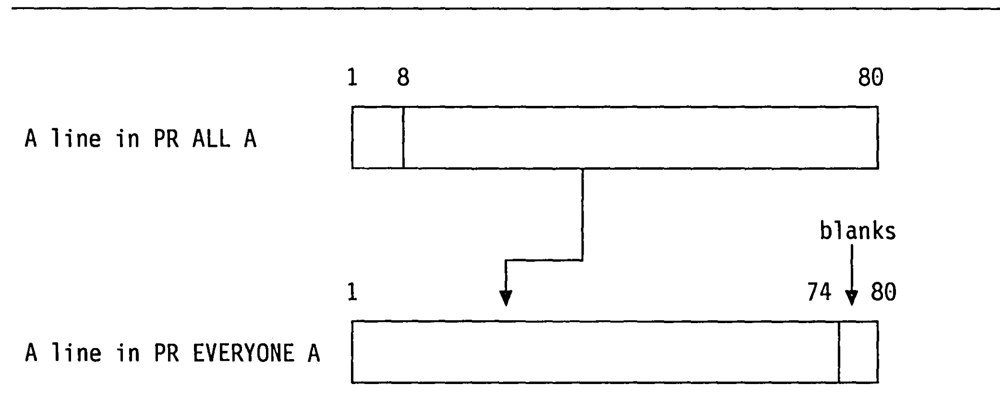
```
/* Gives a warning when the user's primary minidisk     */
/* (filemode A) is more than eighty per cent full       */
"MAKEBUF"
"QUERY DISK A ( STACK"
if rc = 0 then do
   pull                      /* Discard header          */
   parse pull "-" percentage .
   if percentage > 80
   then say "Warning:  Your disk
   is" percentage"% full"
end
else say "NEARFULL EXEC: unexpected return code" rc
"DROPBUF"
```

Figure 49. 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 *VM/SP CMS Command Reference.*

In this program, the answer is provided by the interpreter; it is QUEUEd onto the program stack before COPYFILE command is issued.

```
/* 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"
"COPY PR ALL A PR EVERYONE A ( SPECS NOPROMPT REP"
if rc ¬= 0 then say "Unexpected return code",
                    rc "from COPYFILE command."
"DROPBUF"
```

Figure 50. LEFT7 EXEC

You have just completed Step 33.

Reading 2 continues in Chapter 7, "File Processing" on page 143.

CP Commands

How to Suppress Messages Issued by CP Commands

Reading 3

To issue a command to CP, suppressing messages and obtaining only the return code, use the CMS command EXECIO:

```
"EXECIO 0 CP (STRING" cp command
```

where:

`EXECIO` is a CMS command.

`0` specifies that no data is to be returned.

`CP` is a keyword of EXECIO, specifying that a command is to be passed to CP.

`STRING` is an option of EXECIO, specifying that the remainder of the command line 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 (filemode 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 a failure, your files could be lost forever.) To obtain a temporary minidisk, with the physical characteristics of an IBM 3330, a virtual address (cuu) of 192 and an extent of 5 cylinders, you could type on the CMS command line:

```
define t3330 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:

```
"EXECIO 0 CP (STRING DEFINE T3330 AS 192 CYL 5"
if rc ¬= 0 ...
```

How to Obtain the Reply from a CP Command

To obtain the reply from a CP command in a REXX program, use:

```
"EXECIO * CP (STRING" cp_command
```

Here, the asterisk (*) specifies that all the lines in the reply are to be returned. EXECIO will put the reply in the program stack. There will be as many items in the program stack as there would have been lines on the screen.

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 *filemode* that are **not** on the list, and which can, therefore, be used as the vaddr and filemode of a temporary minidisk.

```
/* This program obtains a temporary minidisk, using      */
/* a virtual address (vaddr) and a filemode 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 filemode 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 filemodes are in    */
/* use, the return code is 27.  Otherwise, the return    */
/* code is that of the CMS or CP command that prevented */
/* success.                                              */
/*-------------------------------------------------------*/
/* 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 until substr(line,1,2) ¬= "/*"
         line = sourceline(n)
         say line
      end
      return 16
   end
end
/*-------------------------------------------------------*/
/* How was this program called                           */
/*-------------------------------------------------------*/
parse source . howcalled .        /* See the Reference  */
                                  /* manual.            */
/*-------------------------------------------------------*/
/* Find unused CUU                                       */
/*-------------------------------------------------------*/

                                  /* continued ...      */
```

Figure 51 (Part 1 of 3). TDISK EXEC

```
"MAKEBUF"
signal on error
theirs = queued()
"EXECIO * CP (STRING Q DASD"
used = ""
do while queued() > theirs
   pull  . cuu .
   used = used cuu
end
do newcuu = 200 while pos(newcuu,used) ¬= 0
end
/*----------------------------------------------------*/
/* Find unused filemode                               */
/*----------------------------------------------------*/
alphabet = "ABCDEFGHIJKLMNOPQRSTUVWXYZ"
do letter = 1 to 26 until response = "NOT ACCESSED."

   "Q DISK" substr(alphabet,letter,1) "(LIFO"  /* PUSH */

   pull . . response     /* get last line of last reply */
end

signal off error; "DROPBUF"   /* clear our buffer      */
"MAKEBUF"; signal on error
if letter = 27 then do
   if howcalled = "COMMAND"
      then say "All filemodes in use"
   return 27
end
newfm = substr(alphabet,letter,1)

/*----------------------------------------------------*/
/* Obtain and format minidisk                         */
/*----------------------------------------------------*/
"EXECIO 0 CP (STRING DEFINE T3330 AS" newcuu,
                           "CYL" cylinders

push "TEMP"
push "YES"
"SET CMSTYPE HT"
"FORMAT" newcuu newfm
"SET CMSTYPE RT"
signal off error
"DROPBUF"
exit

                                  /* continued ...     */
```

Figure 51 (Part 2 of 3). TDISK EXEC

```
/*---------------------------------------------------*/
/* 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 51 (Part 3 of 3). TDISK EXEC

The COMMAND Environment

So far, we have said that the interpreter 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 interpreter to execute 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 filetype 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 128 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 filetype 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 were prefixed with ADDRESS COMMAND. If you have done this, and you want to switch back to the CMS environment, use:

```
ADDRESS CMS
```

Reading 3 continues in Chapter 7, "File Processing" on page 143.

Chapter 7. File Processing

Files are where you keep your data. You can store files in SFS directories and/or on minidisks.

In this chapter:

Reading 1 **immediately following, describes:**

- Writing files using the EXECIO command.

Reading 2 **on page 145, describes:**

- Reading files using the EXECIO command and the program stack
- The FINIS command
- Using data from an existing file to create a new file
- Precautions to be taken when modifying an existing file.

Reading 3 **on page 156, describes:**

- Other ways of processing files
- Processing files in the CMS shared file system.

Note: The newer commands refer to the component parts of a file as *lines*. The older commands (such as COPYFILE) use the term *records*. They are the same thing. In this book we use the term lines.

For details on the EXECIO command, see the *VM/SP CMS Command Reference*.

Writing Files

Reading 1

To write one line to a file, use the following command:

```
"EXECIO 1 DISKW" fname ftype fmode "(STRING" expression
```

where:

EXECIO	is a CMS command.
1	is the number of lines to be written. (For simplicity, write one line at a time.)
DISKW	is a keyword of EXECIO that specifies "Write to a file" on a minidisk or in a directory.
fname	is the filename of the file to be written to.
ftype	is the filetype of the file to be written to.
fmode	is the filemode, a letter specifies the minidisk or directory where the file is to be written.
STRING	is an option of EXECIO, specifying that the remainder of the command is to be written as a line in the file. When used in a REXX program, this can be followed by an expression.
expression	is a REXX expression. The result of this expression becomes a line in the file.

This line is added at the end of the file. If the file does not exist, this line becomes the first and only line.

Example: An Editor

Our first example is a rather simple editor for adding one or more lines to a file.

```
/* World's smallest editor */
say "Enter filename and filetype"
pull fn ft .
say "Enter data:  as many lines as you like.",
    " To finish, press ENTER without entering any data."
do until rc ¬= 0
   parse pull line


   if line = ""                     /* empty line?         */
   then exit                        /* If so, return to CMS */


   "EXECIO 1 DISKW" fn ft "A (STRING" line
end
say "Unexpected return code" rc "from EXECIO 1 DISKW"
```

Figure 52. EDDY EXEC

Notice that EXECIO, like other CMS commands, can indicate errors by giving nonzero return codes. In Figure 52, the DO UNTIL instruction causes the interpreter to leave the loop when a nonzero return code is encountered.

You have just completed Step 15.

Reading 1 continues in Chapter 8, "XEDIT" on page 161.

Reading Files

Reading 2

To read one line of a file from a minidisk or a directory, use:

```
"EXECIO 1 DISKR" fname ftype [fmode]
```

EXECIO puts the line in the program stack at the back of the queue. You can get the line off the program stack with a PULL instruction.

To avoid clashes with data from other commands, use MAKEBUF to create a separate buffer that you will use only for transferring file lines. On your way out, get rid of leftover stacked lines by using the DROPBUF command.

You can also read a line of a file and assign it to a variable. Use:

```
"EXECIO 1 DISKR" fname ftype [fmode] "(VAR" symbol
```

EXECIO puts the line from the specified file into the variable `symbol`. (See the *VM/SP CMS Command Reference* for more details on the EXECIO command.)

End of File

Usually, the first time you execute EXECIO for a particular file, you get the first record; the second time, you get the second record; and so on. Nothing special happens when you read the last record, but the next time, EXECIO responds with a return code of 2, indicating *end of file*

Example: To Display a File on the Screen

```
/* Example: reads and displays a file  (like the CMS    */
/* command TYPE).  Default filemode is "A".             */
/*                                                      */
/* Command format:  MYTYPE filename filetype [filemode] */

arg fn ft fm .
if ft = "" then do
   say "Command format is:  MYTYPE fn ft [fm]"
   exit
end
if fm = "" then fm = "A"
"MAKEBUF"

do forever
   "EXECIO 1 DISKR" fn ft fm "(VAR LINE"
   if rc ¬= 0 then leave
   say line
end

if rc ¬= 2                      /* End of file?          */
then say "Unexpected return code" rc,
         "from EXECIO 1 DISKR" fn ft fm '(VAR LINE'
"DROPBUF"
```

Figure 53. MYTYPE EXEC

The FINIS Command

Once you have started reading or writing a file, it is said to be *open*. CMS "remembers" which record you have written or read, so that when you ask to process the next record, CMS can quickly find it.

When you have finished writing or reading a file, you should use the FINIS command to tell CMS to *close* it.

```
FINIS fn ft [fm]
```

The return code from this command is:

0 if the file was open, and the close was successful

6 if the file could not be closed because it was already closed

31 if the close failed on an SFS file. Changes are rolled back.

Here are two reasons why you should use FINIS:

1. After you have closed your file, any other program that attempts to read the file will have to open it again, and will thus start processing in the usual place. This removes the risk of the wrong record being processed.
2. When your program writes a file and then calls another program to process it, the other program may give an error message "File already open." In this case, you **must** close your file (using FINIS) after you have written it and before calling the other program.

Example: A Time Recording Program

Here is a practical example you can use to record the amount of time you and your colleagues spend at work. There are three separate REXX programs:

CLOCKIN records the time the user arrived today in a file called 'date CLOCKIN A'.

CLOCKOUT picks up the time the user arrived from 'date CLOCKIN A', appends the time the user departed, and writes all this as a single line in the file 'month CLOCKUP A'.

CLOCKUP reads the records in 'month CLOCKUP A' and computes the total time the user has been present this month.

Further details:

- A user who has executed "CLOCKIN" is considered to be present until "CLOCKOUT" is executed. If the user executes "CLOCKIN" when already considered present, a message is displayed.
- A user can "CLOCKIN" and "CLOCKOUT" more than once in a day.
- Midnight is an important time in this system. Users should "CLOCKOUT" before midnight. If this is not done, "month CLOCKUP A" will have to be manually updated.

```
/* Part of a suite of programs for time recording.      */
/* This program records the current time in the file    */
/* "date CLOCKIN A".  There is just one line in the     */
/* file.  It reads:                                     */
/*                                                      */
/*     dd mon yyyy arrived at hh:mm:ss                  */
/*                                                      */
/* But if this file already exists, a message is issued */
/* instead.  The message reads:                         */
/*                                                      */
/*     Arrived today at hh:mm:ss                        */
/*                                                      */

parse value date() with day month year
filename = day||translate(month)||substr(year,3,2)

"SET CMSTYPE HT"
"STATE" filename "CLOCKIN A"
rcsaved = rc
"SET CMSTYPE RT"

"MAKEBUF"
signal on error
select
   when rcsaved = 28 then do
      "EXECIO 1 DISKW" filename "CLOCKIN A (STRING",
               date() "arrived at" time()
   end
   when rcsaved = 0 then do
      "EXECIO 1 DISKR" filename "CLOCKIN A"
      pull . . . . . time
      say "Arrived today at" time
   end
   otherwise
   say "Unexpected return code" rcsaved
end
signal off error
"FINIS" filename "CLOCKIN A"
"DROPBUF"
exit
/*----------------------------------------------------*/
/* Error handler:  common exit for nonzero return codes */
/*----------------------------------------------------*/

                                   /continued*/
ERROR:
say "Unexpected Return Code" rc "from command:"
say "      " sourceline(sigl)
say "at line" sigl
"DROPBUF"
```

Figure 54. CLOCKIN EXEC

```
/* Timekeeping program.  Processes file created by      */
/* CLOCKIN EXEC.                                        */
/*                                                      */
/* If the file "date CLOCKIN A" does not exist, the     */
/* message "FILE NOT FOUND" is displayed and no other   */
/* action is taken.  Otherwise, the file is read, the   */
/* words                                                */
/*                                                      */
/*      departed at hh:mm:ss                            */
/*                                                      */
/* are added, and the line is appended to the file      */
/* "month CLOCKUP A".  When this has been done, the     */
/* file "date CLOCKIN A" is erased.                     */

parse value date() with day month year
filename = day||translate(month)||substr(year,3,2)

"STATE" filename "CLOCKIN"
if rc ¬= 0            /* File not found.  STATE command */
then exit rc          /* has provided error message     */

"MAKEBUF"
signal on error
"EXECIO 1 DISKR" filename "CLOCKIN A"
parse pull line

"EXECIO 1 DISKW" filename "CLOCKOUT A (STRING",
         line "departed at" time()
"FINIS" filename "CLOCKOUT A"

"COPY" filename "CLOCKOUT A MONTH" "CLOCKUP A (APPEND"
"ERASE" filename "CLOCKOUT A"
"ERASE" filename "CLOCKIN A"
say "Operation complete."
signal off error
"DROPBUF"
exit

/*------------------------------------------------------*/
/* Error return code from CMS command                   */
/*------------------------------------------------------*/
ERROR:
say "Unexpected return code" rc "from command at line",
    sigl "in CLOCKOUT EXEC."
"DROPBUF"
```

Figure 55. CLOCKOUT EXEC

Did You Understand That?

1. Copy CLOCKIN and CLOCKOUT to your primary directory or minidisk (filemode A). Test them to see if they will perform as specified.
2. Arrange that every day, the first time you log on, your arrival time is automatically recorded.

3. Write the CLOCKUP program, previously specified, which computes how much time you have worked this month.

Answers:

1. Did everything work correctly? If not, study the error messages and check that you typed everything in correctly.
2. This can be done by adding the line

   ```
   CLOCKIN
   ```

   to your PROFILE EXEC A. If your PROFILE EXEC is not written in REXX, you should write "EXEC CLOCKIN".
3. A possible answer is:

```
CLOCKUP EXEC


/* This program computes total time worked in any   */
/* month, using data obtained from the file         */
/* "month CLOCKUP A".  Command format is:           */
/*                                                  */
/* CLOCKUP [month]                                  */
/*                                                  */
/* Default is this month.                           */
monthnames = "JAN FEB MAR APR MAY JUN JUL AUG SEP",
             "OCT NOV DEC"
/*--------------------------------------------------*/
/* Compute MONTH using argument from command line   */
/*--------------------------------------------------*/
arg month .
if month = ""
then do
   month = substr(date("month"),1,3)
   upper month
end
else do
   if pos(month,monthnames) = 0
   then do
      say "This program computes time worked",
          "in any month."
      say "COMMAND FORMAT:  CLOCKUP [month]"
      say "   (default is this month)"
      say "Names of months are" monthnames
      exit
   end
end
                                  /* continued ...   */
```

```
/*-----------------------------------------------*/
/* Compute total time worked.                    */
/* Input is in file MONTH CLOCKUP A              */
/*-----------------------------------------------*/
"MAKEBUF"
total = 0
do forever
   "EXECIO 1 DISKR" month "CLOCKUP A"
   if rc ¬= 0 then leave
   pull . . . . . timein . . timeout .
   total = total + (c2s(timeout) - c2s(timein))
                        /* Note: C2S is an internal */
                        /* function (See below)     */
end
if rc = 2                        /* end of file     */
then do
   minutes = total % 60
   if total // 60 >= 30          /* round to the    */
   then minutes = minutes + 1    /* nearest minute  */
   hours = minutes % 60
   minutes = minutes // 60
   say "Total time worked in" month "was",
       hours "hours" minutes "minutes"
end
else say "Unexpected return code" rc,
         "from EXECIO 1 DISKR" month "CLOCKUP A"
"DROPBUF"
exit
/*-----------------------------------------------*/
/* C2S: Convert to seconds                       */
/* ======================                        */
/* converts time in the format                   */
/*    hh:mm:ss                                   */
/* to seconds                                    */
/*-----------------------------------------------*/
C2S:
arg hh ":" mm ":" ss
return ss + 60*(mm + 60*hh)
```

You have just completed Step 34.

Using Data From an Existing File to Create a New File

We can now consider file processing in general. The simplest file processing programs involve taking data from one input file and using it to produce one output file; this kind of program will be discussed now. More complicated programs, involving multiple input or output files are dealt with in programming textbooks.

A simple file processing program reads an input file a line at a time, performs some computation on the data, and stores the result as a line in an output file. Consider two cases:

1. If the output file does not have the same "filename filetype filemode" as the input file, the only difficulties are: specifying the computations to be performed, and coding them accurately. The STARS EXEC example, shown in Figure 56, is an example of this type of program.
2. If the output file is required to have the same "filename filetype filemode" as the input file, some care is needed in avoiding the effects of a possible hardware error. SORT8 EXEC on page 157 is an example of this type of program.

Example: Processing a File to Produce a New File

You can use this program as a model for any program that reads (but does not alter) an input file and produces a new output file.

```
/* COMMAND FORMAT:   STARS filename                    */
/*                                                     */
/* FUNCTION:                                           */
/* This program reads records from "filename COPY A"   */
/* Those that do not have a asterisk in column 1 are   */
/* discarded.  The rest are written into a new file    */
/* called "filename STARS A".   Any previous version   */
/* of "filename STARS A" is overwritten.               */

/*-----------------------------------------------------*/
/* CHECK argument                                      */
/*-----------------------------------------------------*/
arg filename .

if filename = ? | filename = ""  /* Help needed?       */
then do recno = 1                /* Display prolog     */
   line = sourceline(recno)
   if left(line,2) ¬= "/*" then exit
   say line
end

"STATE" filename "COPY A"
if rc ¬= 0 then do
   say "File" filename "COPY A not found"
   exit
end

/*-----------------------------------------------------*/
/* Erase old output file, if any                       */
/*-----------------------------------------------------*/
"ERASE" filename "STARS A"
if rc = 0 | rc = 28 then nop
else do
   say "Unexpected return code" rc "from ERASE" filename,
       "STARS A"
   exit
end

                                 /* continued ....     */
```

Figure 56 (Part 1 of 2). STARS EXEC

```
/*-----------------------------------------------------*/
/* Process file                                        */
/*-----------------------------------------------------*/
"MAKEBUF"                  /* See page 134.   */
do forever
                             /* Read a record        */
   subcmd = "R"
   "EXECIO 1 DISKR" filename "COPY A (VAR RECORD"
   if rc ¬= 0 then leave
                             /* If it fits           */
   if substr(record,1,1) = "*"
   then do                   /* Copy it to fn STARS  */
      subcmd = "W"
      "EXECIO 1 DISKW" filename "STARS A (VAR RECORD"
      if rc ¬= 0 then leave
   end
end

if subcmd = "R" then do      /* Reading ?            */
   if rc = 2                 /*    End of file ?     */
   then rc = 0               /*    That's OK         */
end
if rc ¬= 0 then say "Unexpected return code" rc,
                    "from EXECIO DISK"subcmd
rcsaved = rc
"DROPBUF"
"FINIS" filename "STARS A"
exit rcsaved
```

Figure 56 (Part 2 of 2). STARS EXEC

Precautions To Be Taken When Modifying an Existing File

If a program is supposed to modify an existing file (that is, produce a new file with the same fileid), what happens if VM/SP has a failure during the processing? What happens if the program has a failure?

What usually happens is that the data already filed is safe, but the data being processed is lost. After a program fails, you might have to run the program again, either from the beginning or from the point of the failure, depending on the recovery capabilities of the program. If the input data is still available, there is no problem. But how does the failure affect the completion of the program? Has the original file been completely altered, partially altered, or not altered at all?

If the file is stored in an SFS file space and either VM/SP or the program itself fails, SFS automatically rolls back to the end of the last commit or rollback. Rollback means that changes made to a file are ignored and are not saved for that file. Commit, on the other hand, means that changes are saved in the permanent copy of the file. Therefore, SFS ensures that you have a definite recovery point, and no data is partially changed. For more information on shared file systems, see your *VM/SP Application Development Guide for CMS*.

If the file is stored on a minidisk and VM/SP fails, in most cases you can recover from the point of the last successful close (except if the file is in CMS 800−byte format, in which case the file might be partially changed). If the file is stored on a

minidisk and the program fails, CMS immediately tries to close the file. Because the program did not complete its processing, the file probably contains partially changed data.

If you know that your program will be used to process only SFS files, you might not need to build in any special precautions, because of the safeguards built into SFS. However, users might have created *aliases* of those files and/or granted *authorities* on the files to other users. You must be careful in your program not to use functions (such as the RENAME command) that would destroy these aliases and authorities.

If your program will be used to process minidisk files, or if you want your program to be able to process all types of files, you can ensure that the program can handle these special cases by using the temporary file technique.

For example, if MY FILE A is to be processed by a program called PROCESS EXEC to produce a new MY FILE A, the PROCESS program should:

1. Read MY FILE A without changing it and generate a processed temporary file called PROCESS CMSUT1 A.
2. COPYFILE PROCESS CMSUT1 A MY FILE A (REPLACE.

   Note: COPYFILE (REPLACE will not destroy SFS authorities that the owner of the file might have granted or aliases that users might have created).
3. ERASE PROCESS CMSUT1 A.

If VM/SP or the program fails, you can usually determine where the failure occurred from the messages that are generated. You can recover your data as follows:

1. If the failure occurred before PROCESS started, MY FILE A is unchanged. Start the program again.
2. If the failure occurred while processing, PROCESS CMSUT1 A might contain incomplete data, but MY FILE A is unchanged. Erase PROCESS CMSUT1 A and restart the program.
3. If the failure occurred while COPYFILE was working, MY FILE A might contain incomplete data, but PROCESS CMSUT1 A contains the completely processed data. Reissue the COPYFILE command.
4. If the failure occurred after COPYFILE completed, MY FILE A was updated successfully. If PROCESS CMSUT1 A still exists, you can erase it.

By convention, the filetype CMSUT1 is reserved for temporary files. Thus, XEDIT CMSUT1 A contains an updated file while it is actually being filed, and COPYFILE CMSUT1 A contains the intermediate result of a COPYFILE command. You will not see a CMSUT1 file generated by a command very often, because it is left in your directory or on your primary minidisk, filemode A, only if there is a failure at the instant that the command is working.

Some of the precautions we have just discussed are applied in an example called SORT8 EXEC, which you can find on page 157.

You have just completed Step 35.

Reading 2 continues in Chapter 8, "XEDIT" on page 161.

Other Techniques

Reading 3

The technique previously described in this chapter is the most general. It works just as well when the file you are modifying is too big to fit into the storage available. And once you have designed one program, it is very easy to design another one.

But in special situations you might consider:

- For searching a file, when only a small proportion of the records need to be processed, use any of the FIND, LOCATE, AVOID or ZONE options of the EXECIO command. The time taken to execute your program will be significantly reduced.
- For changing the format of a file, use the COPYFILE command.
- For rearranging fields within each line of a file, use the COPYFILE command.
- For sorting files according to the contents of a field or fields that occupy fixed position(s) within every line, use the SORT command. This command will "respect" case, that is, "z" is taken to be less than "A".
- For processing files that are small enough to be edited by XEDIT, use XEDIT. For example, to sort a file "ignoring" case, use the XEDIT subcommand SORT. Remember to take care of nonzero return codes from XEDIT commands. (It may take you a little time to learn how to use XEDIT for file processing.)

An Example: Sorting a File

This program contains examples of:

- Using the LISTFILE command to obtain details about an existing file
- Using the SORT command
- Precautions to be taken when modifying an existing file
- Suppressing messages from CMS commands.

```
/* This program sorts according to the contents of     */
/* columns 1 through 8 of each record.  All lowercase   */
/* letters, 'a', are presumed to come before           */
/* uppercase 'A'.                                      */
/*                                                      */
/* COMMAND FORMAT:  SORT8 filename filetype [filemode]  */
/*                                                      */
/* If the operation is completed without error, the     */
/* sorted file replaces the original file.  If there    */
/* is an error (for example, not enough room in your    */
/* file space or on the minidisk you have accessed      */
/* as A), the original file is unchanged.               */
/* Exceptionally, the original file may be destroyed,   */
/* in this case the file SORT8 CMSUT1 A contains the    */
/* true result.                                         */

/*----------------------------------------------------*/
/* Check arguments                                      */
/*----------------------------------------------------*/
arg fname ftype fmode .
if ftype = "" then do
   say "Command format is: SORT8 fn ft [fm]"
   exit
end
if fmode = "*" then do
   say "Filemode '*' is not allowed"
   exit
end
if fmode = "" then fmode = a

/*----------------------------------------------------*/
/* Check fileid, record format, record length           */
/*----------------------------------------------------*/
"MAKEBUF"
"LISTFILE" fname ftype fmode "(STACK FORMAT"
rcsave = rc
if rc = 0                          /* Something stacked */
then pull . . . recfm lrecl .
"DROPBUF"

if rcsave = 28 then say "File not found"
if rcsave ¬= 0 then exit rc
if recfm ¬= "F" then do
   say "File is not fixed format"
   exit 32
end

                                   /* continued ...      */
```

Figure 57 (Part 1 of 3). SORT8 EXEC

```
if lrecl < 8 then do
   say "Record length less than 8"
   exit 24
end
/*-----------------------------------------------------*/
/* If workfile still exists, warn user (For full       */
/* explanation, see page 154).                         */
/*-----------------------------------------------------*/
address command "STATE SORT8 CMSUT1 A"    /* see page 141 */
select
   when rc = 28 then nop /* does not exist. All's well  */

   when rc = 0 then do
      say "Work file SORT8 CMSUT1 A from previous SORT8",
          "command still exists."
      say "Check that previous operation did not destroy",
          "any files."
      say "Then ERASE SORT8 CMSUT1 A and repeat your",
          "SORT8 command."
      exit
   end

   otherwise
   say "Unexpected return code" rc,
       "from STATE command"
   exit rc
end

/*-----------------------------------------------------*/
/* Sort into SORT8 CMSUT1 A                            */
/*-----------------------------------------------------*/
"MAKEBUF"
"SET CMSTYPE HT"
   queue 1 8                /* sort field              */
"SORT" fname ftype fmode "SORT8 CMSUT1 A"
rcsave = rc
"SET CMSTYPE RT"
"DROPBUF"

                                  /* continued ...      */
```

Figure 57 (Part 2 of 3). SORT8 EXEC

```
if rcsave = 0 then nop      /* All's well                 */
else do
   select
      when rcsave = 40 then
         say "Maximum number of records exceeded"
      when rcsave = 100 then
         say "Error reading/writing file."
      otherwise
      say "Unexpected return code" rcsave,
          "from SORT command."
   end
   say "Original file unchanged."
   "ERASE SORT8 CMSUT1 A"
   exit rcsave
end
/*-----------------------------------------------------*/
/* Replace old file with sorted file                   */
/*-----------------------------------------------------*/
"COPYFILE SORT8 CMSUT1 A" fname ftype fmode "(REPLACE"
if rc ¬= 0 then do
   say "Unexpected return code" rc,
       "from COPYFILE command.  Possible disaster."
   exit
"ERASE SORT8 CMSUT1 A"
```

Figure 57 (Part 3 of 3). SORT8 EXEC

Processing Files in the CMS Shared File System

To obtain the additional file management and file sharing functions of the CMS Shared File System (SFS) and process SFS Files, you can use:

- CMS commands that support these functions
- Callable services library (CSL) routines.

The CSL supplied with VM/SP is named VMLIB. VMLIB contains routines that you can call from your REXX program to perform SFS functions, such as:

- Opening and closing files and directories
- Reading and writing records and blocks
- Creating directories and aliases
- Granting and revoking authorities
- Committing or rolling back changes.

For information about how to code the REXX CSL external function call, see the *VM/SP System Product Interpreter Reference.* CSL routines are described in the *VM/SP Application Development Guide for CMS* and the *VM/SP Application Development Reference for CMS.*

Reading 3 continues in Chapter 8, "XEDIT" on page 161.

Reading 3

Chapter 8. XEDIT

XEDIT is the editor supplied with VM/SP. 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 invoked 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 165, 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 168, 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 *VM/SP System Product Editor Command and Macro 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 filetype of XEDIT and execute that. This type of file is called a *macro*.

For example, if the file TEN XEDIT, shown in Figure 58, 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 fail with 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 filetype of XEDIT. To indicate that your program is written in the REXX language, it must begin with a REXX comment, as usual.

Because the filetype of your program is XEDIT, the interpreter will assume that the environment is XEDIT. And therefore any clause in the program that the interpreter 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 filenames from one-to-eight characters long. The filenames 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.

```
/* This program changes the settings of PF Keys 7 and 8 */
/* so that you scroll backwards or forwards 10 lines    */
/* at a time.                                           */
"SET PF7 UP 10"
"SET PF8 NEXT 10"
```

Figure 58. 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.

```
/* 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 59. 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 *VM/SP System Product Editor Command and Macro 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:

```
/* Example: tracing a syntax error */
trace errors
"EXTRACT" tooth      /* EXTRACT is a valid command, but */
                     /* 'tooth' is not a valid operand  */
```

Figure 60. DENTAL XEDIT

Reading 1

Executing the command DENTAL from the XEDIT command line would cause the following to be displayed:

```
dental
3 *-* extract tooth /* EXTRACT is valid command, but */
  +++ RC(5) ++++
```

Messages

To display messages in the XEDIT message area, use:

```
MSG text_of_message
```

For example:

```
NEXT
if rc = 1 then MSG "End of file reached"
```

You have just completed Step 16.

Reading 1 continues in Chapter 9, "Control" on page 171.

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, use:

```
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 use:

```
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 of screen:

```
/* This program changes the settings of PF Keys 7 and 8 */
/* so that you scroll backwards or forwards half a      */
/* screen at a time.                                    */
"EXTRACT /SCREEN"
amount = (substr(screen.1,5) - 4) % 2
"SET PF7 UP" amount
"SET PF8 NEXT" amount
```

Figure 61. 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 *VM/SP System Product Editor Command and Macro 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.

```
/* 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 62. 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 *VM/SP System Product Editor User's Guide*.

```
/* 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 /TRUNC /FTYPE"
if (ftype.1='DIR-UPDT') | (ftype.1='DIRECT')
   then SET TRUNC 72
if (recfm.1='F') & (trunc.1<=72) then SET SERIAL ALL
RETURN:
   return
ERROR:
   "SOS ALARM"
   msg "Unexpected return code" rc "from line" sigl,
       "of XEDIT profile"
   return
```

Figure 63. PROFILE XEDIT

Reading 2 continues in Chapter 9, "Control" on page 171.

Menus Using XEDIT

Reading 3

The System Product Editor (XEDIT) can be used with the System Product Interpreter to generate full-screen menus. A short example of a full-screen menu is shown in Figure 65. It shows the user the name of the last file edited, lets the user select this file or another, and then invokes 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 *VM/SP System Product Editor Command and Macro Reference*, for information on the XEDIT subcommands and macros.

The TESTMENU program, following, invokes XEDIT using SAMPMENU as a profile. To try this, create the TESTMENU program and then enter `testmenu` on the CMS command line.

```
/* sample exec to show use of an XEDIT full screen menu */
"XEDIT" lastfile edited "(PROF" sampmenu
```

Figure 64. TESTMENU EXEC

To try this, enter `testmenu` on the CMS command line. The TESTMENU program, following, simply invokes XEDIT using SAMPMENU as a profile.

```
/* 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'
"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  */
  "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, re-set 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,               */
                      cms desbuf           /*    clear stack     */
                      command quit         /*    and quit        */
                      exit
                    end
        else message = "Unsupported PF key"  /* bad PF key pressed */
      otherwise message = "Unsupported function" /* unknown func.  */
    end
  end
  if (message = "") & (words(oldname oldtype)=2) then leave
end

                                               /* continued...     */
```

Figure 65 (Part 1 of 2). SAMPMENU XEDIT

```
/* 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"
  end
  if message ¬= "" then do; emsg message; message=''; end;
  "CURSOR SCREEN 8 26"
  return
```

Figure 65 (Part 2 of 2). SAMPMENU XEDIT

Reading 3 continues in Chapter 9, "Control" on page 171.

Chapter 9. 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 interpreter to select for execution one of a number of lists of instructions, use the IF instruction or the SELECT instruction.

Loops To tell the interpreter 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 interpreter to finish executing your program, use the EXIT instruction.

Calls to subroutines To tell the interpreter 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 interpreter 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 interpreter how to decide which instructions are to be executed next, you can use the IF instruction or the SELECT instruction.

Reading 1 **covers the entire "Selection" section starting on page 173 It 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.**

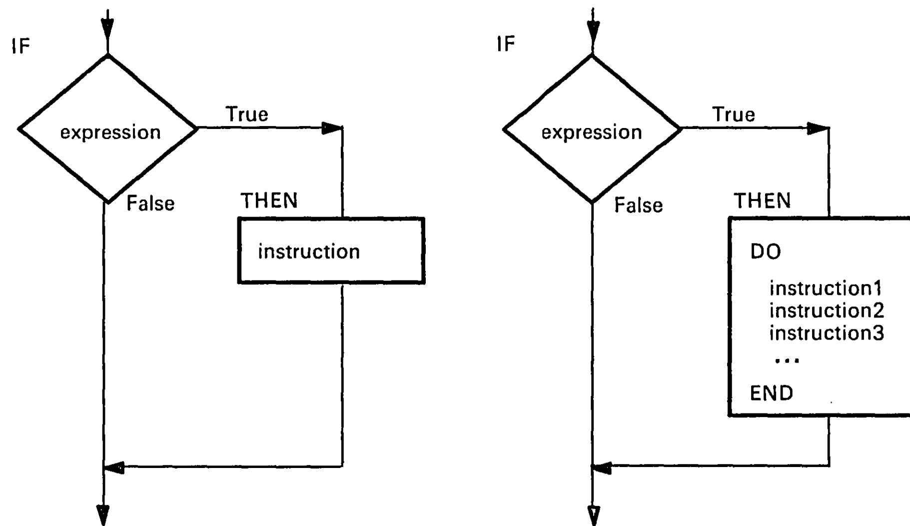
- Continue **Reading 2** in "Loops" on page 184.

Reading 3 **skips this section.**

- Continue **Reading 3** in "Loops" on page 184.

The IF Instruction

Reading 1



To tell the interpreter how to make a decision about a single instruction use:

```
IF expression
THEN instruction
```

The interpreter 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 interpreter 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 interpreter 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.

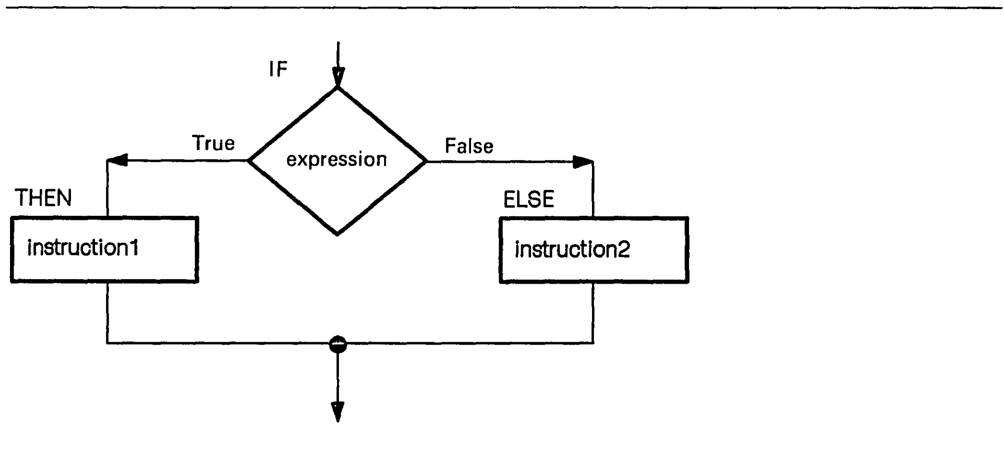
For example:

```
if answer = "YES"
then do
   say "OK.  Please enter the filename and filetype",
       "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.

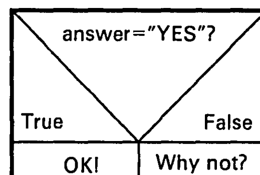
The ELSE Keyword



When you want the interpreter to select from one of two possible instructions use:

```
IF expression
THEN instruction1
ELSE instruction2
```

The interpreter will execute instruction2 only if expression is false. For example, if you wanted:



you could code:

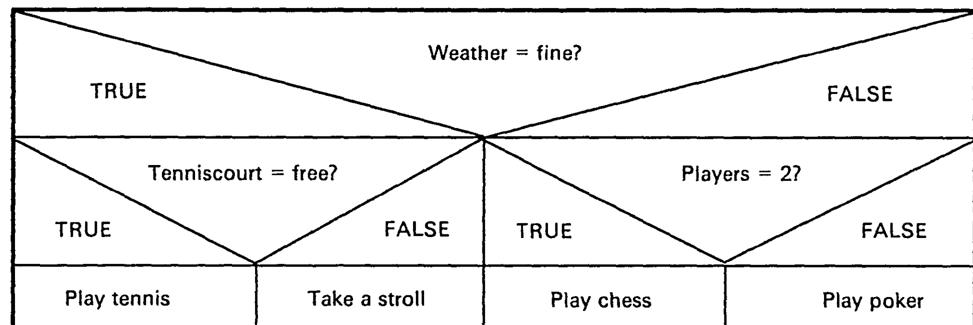
```
if answer = "YES"
then say "OK!"
else say "Why not?"
```

The interpreter 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 83).       */
   then say "I see"
   else say "I just don't understand you"
end
```

More complicated situations can be handled using a succession 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 the human reader 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 ...; ...; END"s also help the interpreter 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 interpreter 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 ...
```

Did You Understand That?

1. 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:

1. 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.

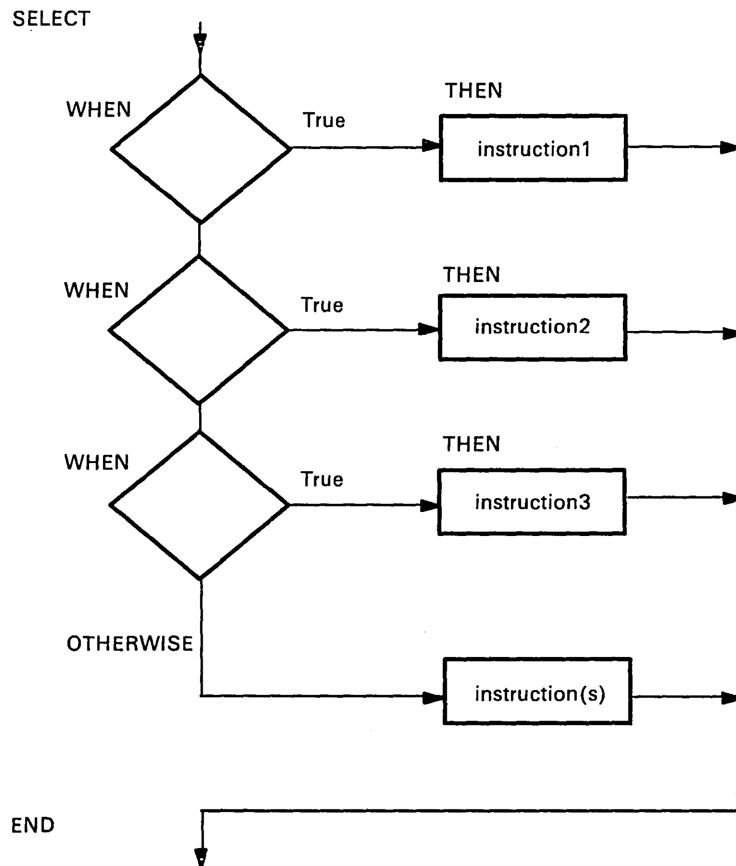
Weather = fine?
TRUE
FALSE
Tenniscourt = free?
TRUE
FALSE
Play tennis
Players = 2?
TRUE
FALSE
Play chess
Play poker

For the values given in the WHATODO EXEC, the following will result:

```
whatodo
     7 *-* if weather = fine
       >>>   "1"
     8 *-*  then
     9 *-*  if tenniscourt = free
       >>>   "1"
    10 *-*   then
       *-*   say "Shall we play tennis?"
       >>>     "Shall we play tennis?"
Shall we play tennis?
    11 *-*   /* else say "Shall we take a stroll?" */
Ready;
```

You have just completed Step 17.

The SELECT Instruction



If you want the interpreter 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 expression1 is true, instruction1 is executed. After this, processing continues with the instruction following the END.
- But if expression1 is false, expression2 is tested. Then, if expression2 is true, instruction2 is executed and processing continues with the instruction following the END.
- If all of expression1, expression2, 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 interpreter 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 interpreter 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:

```
/* 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 66. 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:

```
/* 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 67. PILOT 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 68. TRUCKER EXEC

Did You Understand That?

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
```

You have just completed Step 18.

Reading 1 continues in “Loops” on page 184.

Loops

A *loop* is a group of instructions that may have to be executed more than once.

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 196, describes:

- Compound DO instructions
- Leaving a specified loop.

Reading 3 on page 198, 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. The first is about preparing for a meeting. Each person attending will require three documents. The program that prints the documents is:

```
/* 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.                                      */

"SPOOL PRINT CONT"            /* See the following note  */
do 5
   "PRINT AGENDA DOCUMENT"
   "PRINT MINUTES DOCUMENT"
   "PRINT ACCOUNTS DOCUMENT"
   "SPOOL PEINT CLOSE*sdq.        /* See the following note  */
end
"SPOOL PRINT NOCONT"
```

Figure 69. HANDOUTS EXEC

The program in Figure 69 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 your *VM/SP CP General User Command Reference* for details.

In this next program, the instruction between the DO and the END will be executed HEIGHT times.

```
/* 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 70. 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
```

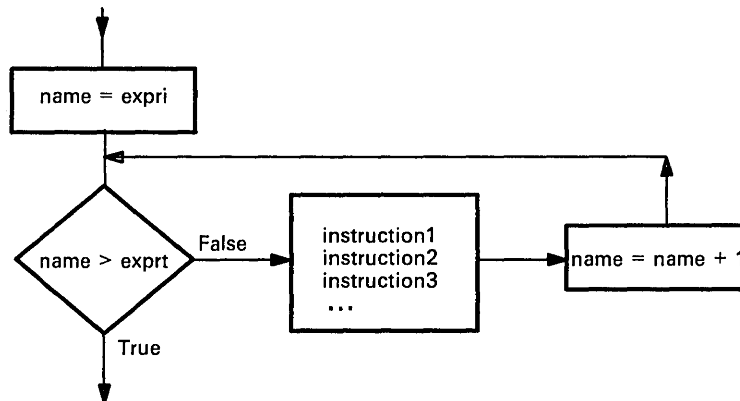
where:

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 is used to compute the width of each row of stars.

```
/* 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 71. 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 [BY exprb] [TO exprt]
   ...
   ...
END
```

where:

exprb (the **expr**ession for **BY**) gives the number that is to be added to *name* at the bottom of the loop.

Did You Understand That?

1. If you carefully study the flowchart on page 186, 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").

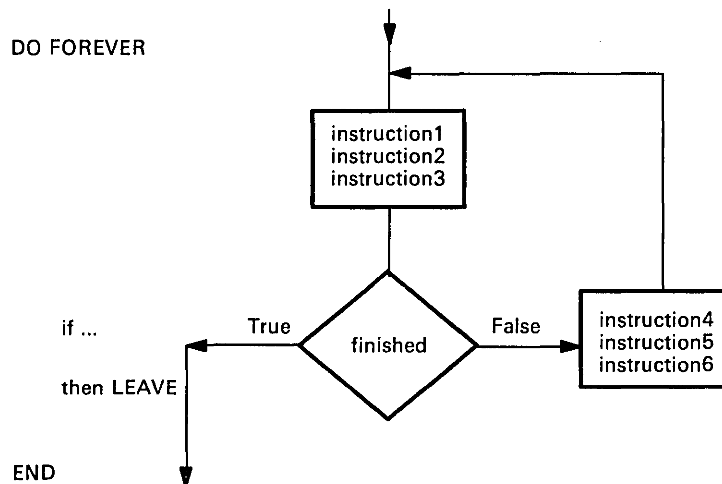
4. Eight, (on the ninth pass, the EXIT instruction ends the program before the SAY instruction is reached).

You have just completed Step 19.

Reading 1

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

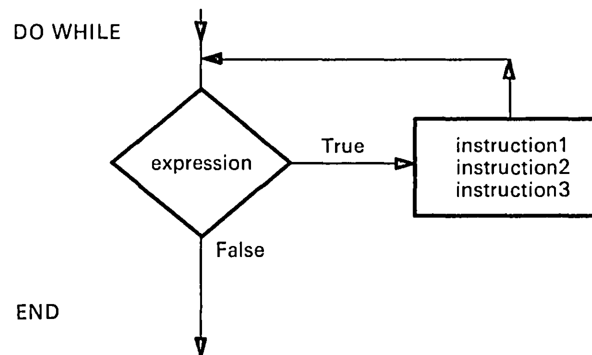
LEAVE

causes processing to continue with the instruction following the END keyword. For example, the SUM EXEC in Figure 72 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.

```
/* 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 72. 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 (**expression** for **while**) 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.

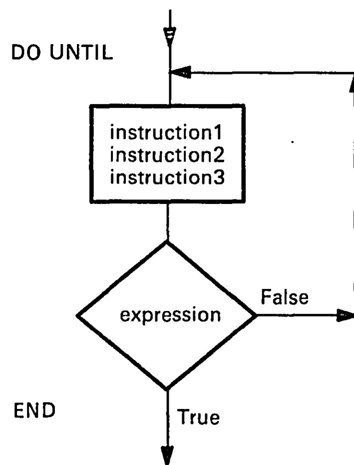
These two fragments will produce the same results.

```
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".

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 instruction
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.

Did You Understand That?

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.)

You have just completed Step 20.

Reading 1 continues in "The EXIT Instruction" on page 200.

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 *VM/SP System Product Interpreter 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.

```
/* 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 73. 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 interpreter 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 201.

The ITERATE Instruction

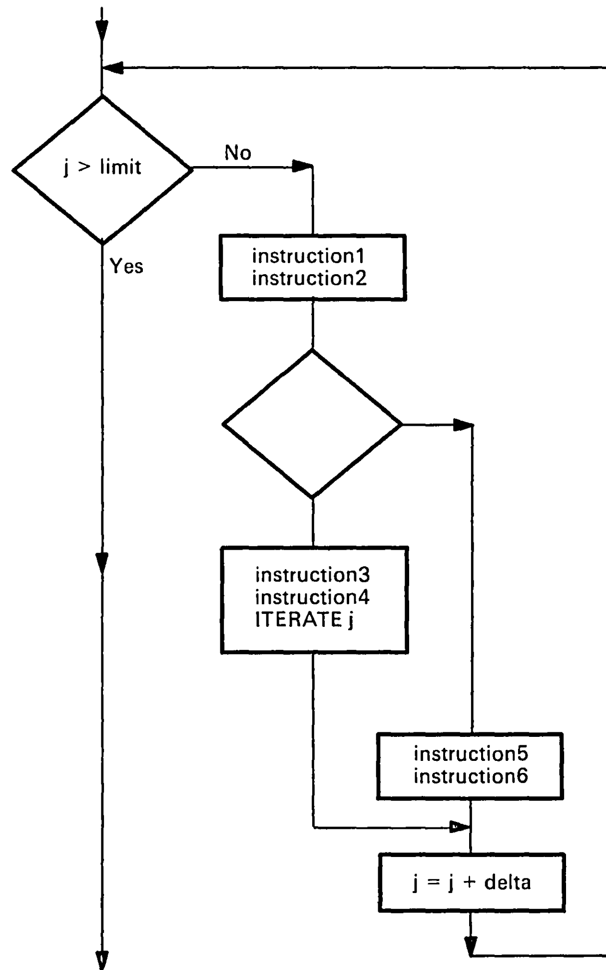
Reading 3

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 interpreter 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, normal 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 212.

The EXIT Instruction

Reading 1 **covers the entire "EXIT Instruction" section. It describes:**

- How to leave your program by using the EXIT instruction.

Reading 1

To tell the interpreter 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:

```
/* Example: using EXIT with a return code */
say "Returning to CMS"
exit 22
```

Figure 74. FADE EXEC

When run, the program in Figure 74 will cause this to be displayed:

```
fade
Returning to CMS
Ready(00022);
```

Reading 1 continues in "Subroutines" on page 201.

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 209, describes:

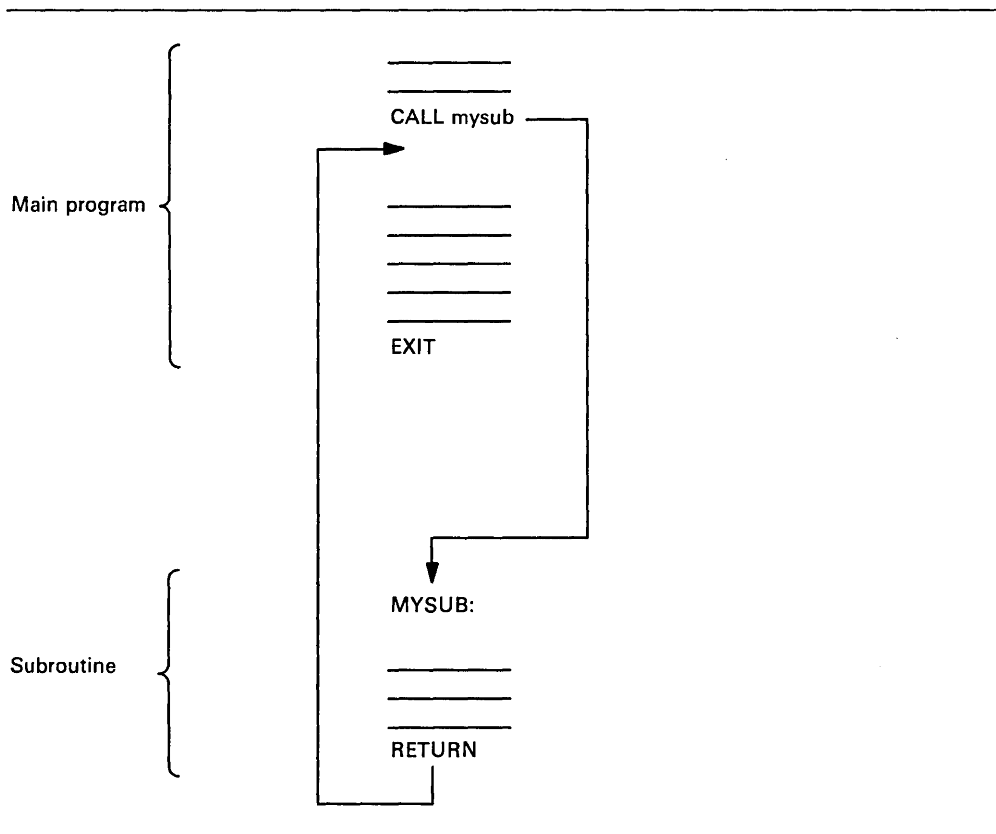
- Subroutines and functions
  - What are the differences
  - What are the similarities
- Parsing the arguments
- External subroutines.

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 interpreter to look through your program until it finds the *label* that marks the start of the subroutine. Processing continues from there until the interpreter finds a RETURN instruction that causes the interpreter to return to the main program.

A subroutine can be called from more than one place in a program. The interpreter 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 interpreter to execute a subroutine use:

```
CALL subname [argument1, argument2 ...]
```

where:

`subname`
: is the name of the subroutine. The interpreter will first search for the corresponding label in your program. A label consists of a symbol followed by a colon (:), for example:

  subname:

  If no such label is found, the interpreter looks for a built-in function, exec file, or module file of that name. (To be discussed later, on page 213.)

`argument1, etc.`
: are expressions. 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 10 arguments on a CALL instruction.

  You can also obtain the arguments by using the ARG or PARSE ARG instructions, discussed later.

For example:

```
/* 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 75. CHEER EXEC

This is what appears on the screen is:

```
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 75, 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

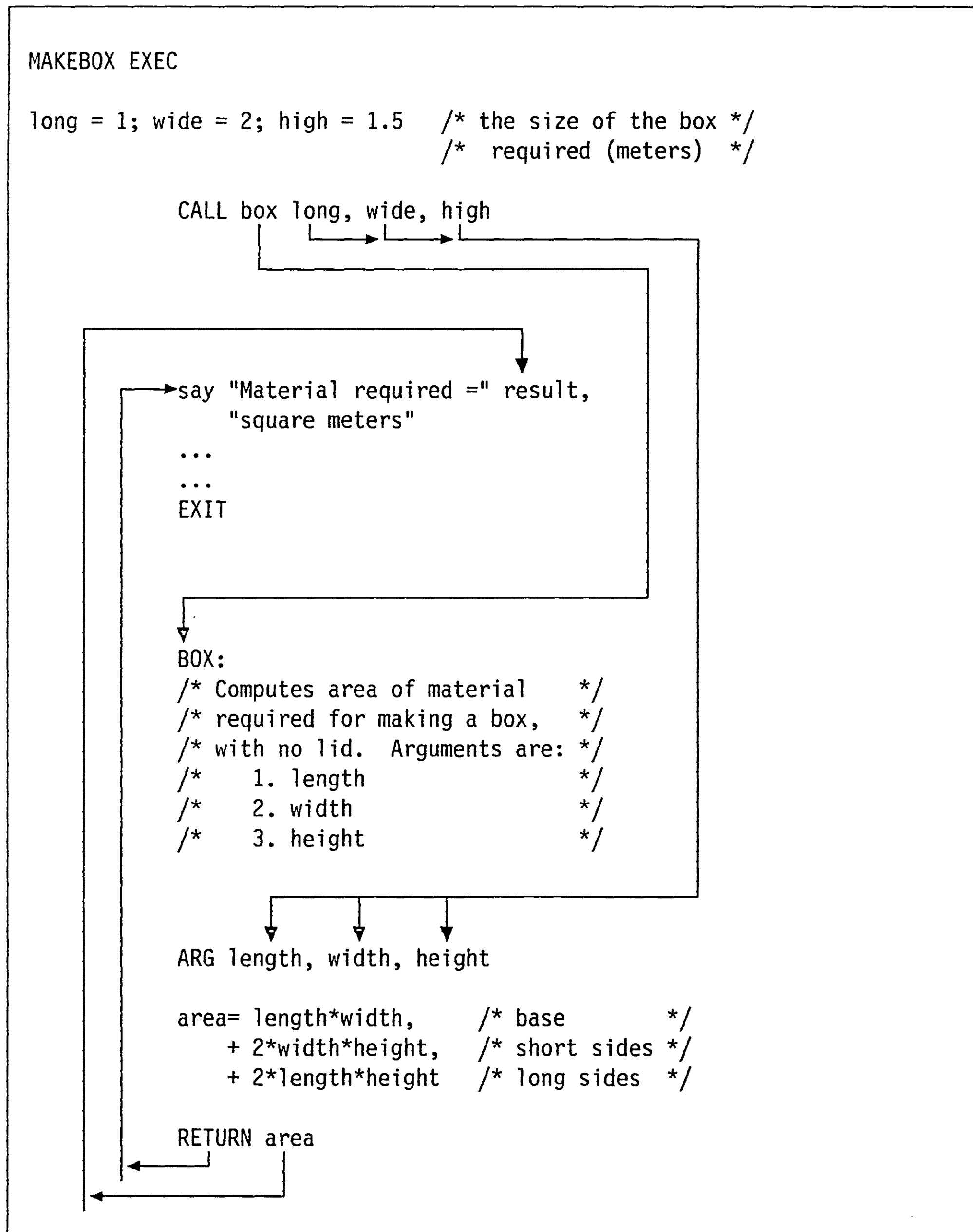
```
RETURN [expression]
```

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.



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.

Did You Understand That?

1. 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  */
   call tell              /* Who's ahead now         */
   b = b + random(1,6)    /* Now it's Barry's turn   */
   call tell              /* Who's ahead now         */
end
exit                      /* End of main program     */
```

2. Copy the main program and your subroutine into an exec file and test your program.

Answers:

1. A possible solution is:

```
/*-----------------------------------------------*/
/* 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.

You have just completed Step 21.

Reading 1 continues in "Jumps" on page 212.

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:

  ```
  CALL routine [argument1, ... ]
  ```

  But to call a function, you use a function call:

  ```
  routine([argument1, ... ])
  ```

- 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 both can be *external* (that is, in a different file).
- Both have the same *search order*. When a call to `routine` is recognized, the interpreter searches for:

  1. The label `routine`: in the same file
  2. A REXX function called `routine`
  3. An external routine.

  (For full details, see your *VM/SP System Product Interpreter Reference*.)

- Both, when they are *internal*, can use the PROCEDURE instruction (described in Reading 3, page 29).
- 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 on page 129, 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 your *VM/SP System Product Interpreter 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:

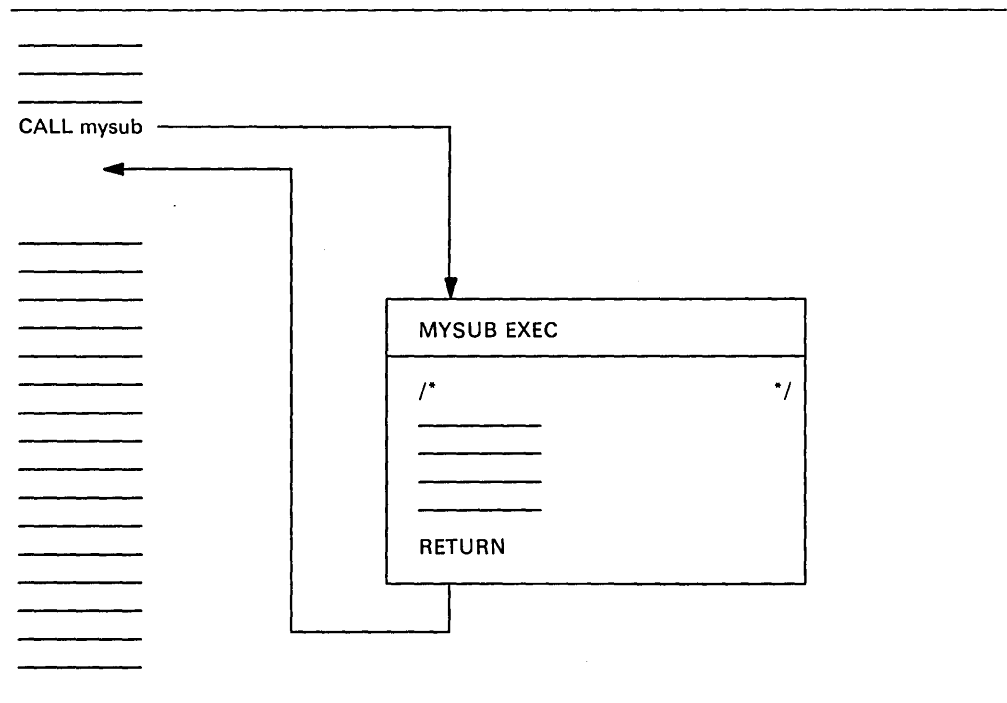
```
WORDS:
PARSE ARG first second third fourth rest, number
```

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.)

You have just completed Step 36.

Reading 2 continues in "Jumps" on page 212.

Jumps

In this section we discuss instructions that cause the interpreter 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 213, describes:**

- How to use the SIGNAL instruction for abnormal changes of control.

Reading 3 **on page 214, describes:**

- How to use the SIGNAL instruction to set "ON-conditions."
  - SIGNAL ON FAILURE
  - SIGNAL ON HALT
  - SIGNAL ON NOVALUE
  - SIGNAL ON SYNTAX.

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 interpreter 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 10, "Programming Style and Techniques" on page 215.

The SIGNAL Instruction

Reading 2

To tell the interpreter 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 interpreter 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 signalled 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 10, "Programming Style and Techniques" on page 215.

SIGNAL ON Condition

Reading 3

To set up a "trap" so that the interpreter, whenever a specified condition is detected, will jump to the corresponding label, use:

```
SIGNAL ON  { ERROR    }
           { FAILURE  }
           { HALT     }
           { NOVALUE  }
           { SYNTAX   }
```

where :

`SIGNAL ON ERROR` has already been discussed. Refer to page 126.

The others work in a similar way, jumping to their own label when the condition is detected, and setting the REXX special variable SIGL.

`SIGNAL ON FAILURE` sets up a "trap" to the label FAILURE that is taken whenever a failure condition from a host command is detected (that is, a negative return code is returned). For example, a failure condition would occur if a command could not be found. This can be useful for debugging commands that would probably cause the program to stop running.

`SIGNAL ON HALT` sets up a "trap" to the label HALT that is taken whenever the CMS immediate command HI (Halt Interpretation) is issued from the CMS command line. This command works in a similar way to HX, but it does not force a return to CMS. This can be useful when debugging a REXX program under XEDIT. You can halt the REXX program without aborting the whole XEDIT session.

`SIGNAL ON NOVALUE` sets up a "trap" to the label NOVALUE that is taken whenever a symbol that could be the name of a variable is encountered, and the variable does not exist. This can be useful for checking a program that is coded in the more reliable style, with all strings that are not numbers in quotes.

`SIGNAL ON SYNTAX` sets up a "trap" to the label SYNTAX that is taken whenever a syntax error is detected. This might be useful for debugging a system program written in REXX when a user complains that the program gives syntax errors, but is not able to produce an accurate description of the problem.

Congratulations! You have successfully completed Reading 3. Now you can try putting 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 215.

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 220, 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.

Did You Understand That?

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

Answers:

1.

```
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
```

You have just completed Step 22.

Happy Hour

As this is the end of Reading 1, here is a chance to relax.

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.)

```
/* 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 ------>   0"
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 76 (Part 1 of 2). CATMOUSE EXEC

```
      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
/*----------------------------------------------------------*/
/* 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 76 (Part 2 of 2). CATMOUSE EXEC

You have just completed Step 23.

Congratulations! You have also 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, "How Your Program Is Interpreted" on page 9.

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:

1. **Main program**: calculate their positions, and
2. **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 terminate?
- 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?

Reading 2

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
```

Figure 77. PULLOVER EXEC

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.

Consider 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

You have just completed Step 37.

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 39).

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"
 ...
```

Tracing Your Program

Or you can use the TRACE instruction, described in your *VM/SP System Product Interpreter 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.

```
/* 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 78. 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 interpreter 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 218), */
/* 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 218),   */
/* 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 218),   */
/* 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

                               /*continued*/
```

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

You have just completed Step 38.

Reading 3 begins in Chapter 2, "How Your Program Is Interpreted" on page 9.

Summary of Changes

Summary of Changes
for SC24-5238-04
for VM/SP Release 6

How to Obtain the Release 5 Edition of this Publication

To obtain the edition of this publication that pertains to Release 5 of VM/SP, order ST00-1593.

New Built-in Functions for Release 6 of VM/SP

DIGITS	Returns the current setting of NUMERIC DIGITS.
FUZZ	Returns the current setting of NUMERIC FUZZ.
WORDPOS	Returns the word number of the first word of a given phrase found in a given string.

New Option Added to an Instruction for Release 6 of VM/SP

- SIGNAL instruction added the Failure option.

New Comparison Operators Added for Release 6 of VM/SP that include:

< <	Strictly less than
> >	Strictly greater than
\< <, ¬< <	Strictly not less than
\> >, ¬> >	Strictly not greater than
< < =	Strictly less than or equal to
> > =	Strictly greater than or equal to

Note: The backslash (\) is synonymous with the NOT symbol (¬). The two may be used interchangeably.

Other Changes

- Restriction on the placement of the PROCEDURE statement is enforced. The PROCEDURE instruction, if used, must be the first instruction executed after the CALL or function invocation.
- New section added to the **File Processing** chapter, 'Processing Files in the CMS Shared File System'.
- The backslash character(\) is supported as a synonym

Miscellaneous

- Minor changes to accommodate the CMS Shared File System (SFS) and VM/XA.
- Minor technical and editorial changes have been made throughout this publication.

Summary of Changes
for SC24-5238-03
for VM/SP Release 5

Miscellaneous

Minor technical and editorial changes have been made throughout this publication. This edition supersedes the previous edition, SC24-5238-02.

Summary of Changes
for SC24-5238-02
for VM/SP Release 4

Miscellaneous

Minor technical and editorial changes have been made throughout this publication. This edition supersedes the previous edition, SC24-5238-01.

Summary of Changes
for SC24-5238-01
for VM/SP Release 3

Miscellaneous

Minor editorial changes have been made throughout this publication. This edition does not supersede the previous edition, SC24-5238-00.

Glossary of Terms and Abbreviations

A

argument. (1) A string passed between a calling routine and a called routine. (2) A parameter provided to a program.

argument list. A complete set of arguments, separated by commas, that is passed between a calling routine and a called routine.

arithmetic operator. Performs arithmetic operations on character strings that are valid numbers. The arithmetic operators include addition (+), subtraction (-), multiplication (*), exponentiation (**), division (/), integer division (%), remainder (//), prefix +, and prefix -.

assembler language. A source language that includes symbolic machine language statements in which there is a one-to-one correspondence with instruction formats and data formats of the computer.

assignment. A single clause with the form **symbol = expression**. An assignment gives a variable a new value.

B

buffer. An area of storage, temporarily reserved for performing input or output, into which data is read, or from which data is written.

built-in function. A function that is supplied by a language. For REXX, these are functions defined as part of the REXX language and include character manipulation, conversion, and information functions.

byte. A unit of storage, consisting of eight adjacent binary digits operated on as a unit and constitute the smallest addressable unit in the system.

C

callable services library (CSL). A package of CMS assembler routines that can be stored as an entity and made available to application programs.

CC. Condition code.

character. A member of a set of elements that can represent, organize, or control data. Characters may be letters, digits, punctuation marks, or other symbols.

clause. An element of a REXX program consisting of zero or more blanks (which are ignored), a sequence of tokens, zero or more blanks (again ignored), and a semicolon delimiter (which may be implied). Clauses can be subdivided into five types: null clauses, labels, assignments, instructions, and commands.

CMS. Conversational Monitor System.

CMS EXEC language. A general-purpose, high-level programming language, particularly suitable for EXEC procedures and EDIT macros. The CMS EXEC processor executes procedures and macros (programs) written in this language. Contrast with *EXEC 2 language* and *Restructured Extended Executor (REXX) language*.

comma. (1) A token that represents the continuation character. (2) A separator of arguments in an argument list. (3) A separator in a parsing template.

command. Single clauses consisting of just an expression. The expression is evaluated and the result is passed as a command string to the default or specified host environment.

comment. A token consisting of characters (on one or more lines) delimited by /* and */. Comments can be written anywhere in a REXX program.

comparison operator. Compares two terms and returns the value '1' if the result of the comparison is true, or '0' otherwise.

compound variable. A symbol that contains at least one period, one character before the period, and one character after the period. A compound symbol cannot start with a digit or period.

concatenate. An operation to combine two strings into one by appending the second string to the right-hand end of the first string.

conditional loop. A loop that allows a set of instructions to be repeated either WHILE or UNTIL a specified condition is met.

conditional phrase. An expression evaluated as either true or false and determines whether an instruction or set of instructions are executed.

condition code (CC). A code that reflects the result of a previous I/O, arithmetic, or logical operation.

console stack. Refers collectively to the program stack and the terminal input buffer.

constant symbol. A symbol that starts with a digit (0-9) or a period and whose value cannot be changed.

continuation character. A character represented by the comma and lets a clause be extended onto more than one line. This character is functionally replaced by a blank and cannot be used in the middle of a string or comment.

control program. A computer program that schedules and supervises the program execution in a computer system. See *Control Program (CP)*.

Control Program (CP). A component of VM/SP that manages the resources of a single computer so multiple computing systems appear to exist. Each virtual machine is the functional equivalent of an IBM System/370.

Conversational Monitor System (CMS). A virtual machine operating system and component of VM/SP that provides general interactive time sharing, problem solving, program development capabilities, and operates only under the control of the VM Control Program (CP).

CP. Control Program.

CP command. A command available to all VM users. Class G CP commands let the general user reconfigure their virtual machine, control devices attached to their virtual machine, do input and output spooling functions, and simulate many other functions of a real computer console. Other CP commands let system operators, system programmers, system analysts, and service representatives manage the resources of the system.

D

DBCS. Double-byte character set.

double-byte character set (DBCS). A character set that requires 2 bytes to uniquely define each character. This contrasts with SBCS, in which each printed character is represented by 1 byte.

E

EBCDIC. Extended binary-coded decimal interchange code.

entry point. An address or label of an instruction performed upon entering a computer program, a routine, or a subroutine. A program can have several different entry points, each corresponding to a different function or purpose.

environment. A collection of logical and physical resources. Examples of environments in VM/SP are CP and CMS.

exec. In REXX, a file with a filetype of EXEC that contains REXX language instructions that are directly interpreted. Execs can also contain commands executed by the host environment. For an exec to be recognized as a REXX exec, the program must start with a comment.

EXEC 2 language. A general-purpose, high-level programming language, particularly suitable for EXEC procedures and XEDIT macros. The EXEC 2 processor runs procedures and XEDIT macros (programs) written in this language. Contrast with *CMS EXEC language* and *Restructured Extended Executor (REXX) language*.

expression. In REXX, a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data. Expressions consist of terms (literal strings, function calls and symbols), and zero or more operators.

extended binary-coded decimal interchange code (EBCDIC). A set of 256 characters, with each character represented by 8 bits.

extended PLIST (untokenized parameter list). Four addresses that indicate the extended form of a command as it was entered at a terminal.

external routine. A program external to the user's program, language processor, or both. These routines can be written in any language (including REXX) that supports the system dependent interfaces used by REXX to invoke it.

F

fblock. File block.

FIFO (first-in-first-out). A queuing technique in which the next item to be retrieved is the item that has been on the queue for the longest time. Contrast with *LIFO (last-in-first-out)*.

file block. The block pointed to by word 4 of the extended parameter list. The file block can be used to execute a program with a file type other than EXEC. It can also be used to execute a program that is already is storage or to override the default address environment.

file ID. A CMS file identifier that consists of a file name, file type, and file mode. The file ID is associated with a particular file when the file is created, defined, or renamed under CMS. See *file name*, *file type*, and *file mode*.

file mode. A two-character CMS file identifier field comprised of the file mode letter (A through Z) followed by the file mode number (0 through 6). The file mode letter indicates the minidisk or SFS directory on which the file resides. The file mode number indicates the access mode of the file.

file name. A one-to-eight character alphanumeric field, comprised of A through Z, 0 through 9, and special characters $ # @ + - (hyphen) : (colon) _ (underscore), that is part of the CMS file identifier and serves to identify the file for the user.

file type. A one-to-eight character alphanumeric field, comprised of A through Z, 0 through 9, and special characters $ # @ + - (hyphen) : (colon) _ (underscore), that is used as a descriptor or as a qualifier of the file name field in the CMS file identifier.

free storage. Storage not allocated. The blocks of memory available for temporary use by programs or by the system.

function. A series of instructions that an exec invokes to perform a specific task and return a value. Three types of routines can be called as functions: internal, built-in, and external.

function call. An invocation of a set of instructions that must return a result. Function calls can be included in an expression anywhere that a term would be valid.

function package. A set of external functions and/or subroutines which can be loaded into virtual storage. The functions then seem like ordinary built-in functions to users. Three supported function packages in REXX are: RXUSERFN, RXLOCRN, RXSYSFN.

G

GCS. Group Control System.

Group Control System (GCS). A component of VM/SP, consisting of a shared segment that the user can IPL and run in a virtual machine. It provides simulated MVS services and unique supervisor services to help support a native SNA network.

H

HELP. An online tool for supplying reference information on commands and messages for VM components.

hexadecimal string. Any sequence of zero or more hexadecimal digits (0-9, a-f, A-F), optionally separated by blanks, delimited by single or double quotes, and immediately followed by the symbol x or X.

I

immediate command. A type of CMS command that, when entered after an attention interruption, causes program execution, tracing, or terminal display to stop. Another immediate command can be entered to resume tracing or terminal display. The immediate commands are HB (halt batch execution), HI (halt all System Product Interpreter or EXEC 2 programs or macros), HO (halt tracing), HT (halt typing), HX (halt execution), RO (resume tracing), RT (resume typing), SO (suspend tracing), TE (trace end), and TS (trace start). They are called immediate commands because they are executed as soon as they are entered; they are not stacked in the console stack. Within an exec, immediate commands can be established or cancelled by the CMS command IMMCMD.

instruction. One or more clauses, the first of which starts with a keyword that identifies the instruction. Instructions affect the flow of control, provides services to the programmer, or both.

interface. A shared boundary between two or more entities. An interface might be a hardware or software component that links two devices or programs together.

internal label. A label that is inside the user's program. In the search order, internal labels take precedence over built-in and external functions.

internal routine. A routine that exists inside the user's program and identified by a label.

invoke. To start a command, procedure, or program.

K

keyword. Symbols reserved for use by the language processor in certain contexts. Keywords must be the first token in a clause and cannot be followed by an equal character or colon. Keywords include the names of the instructions and ELSE, END, OTHERWISE, THEN, and WHEN.

L

label. A clause that consists of a single symbol followed by a colon.

LIFO (last-in-first-out). A queuing technique in which the next item to be retrieved is the item most recently placed in the queue. Contrast with *FIFO (first-in-first-out)*.

literal pattern. In REXX, quoted stings used in a parsing template to specify how a sequence of characters is split up.

literal string. A sequence including any characters and delimited by single quotes (') or double quotes (").

logical operator. Performs logical operations on one or two terms. In REXX, the logical operators include: AND, Inclusive OR Exclusive OR, or Logical NOT. A value of '1' is returned if the expression is true and '0' if the expression is false.

M

macro. A program that performs certain operations in applications such as editors and assemblers. A REXX program that issues subcommands to XEDIT is a macro.

module. (1) A unit of a software product that is discretely and separately identifiable with respect to modifying, compiling, and merging with other units, or with respect to loading and execution. For example, the input to, or output from, a compiler, the assembler, the linkage editor, or an exec routine. (2) A nonrelocatable file whose external references have been resolved.

N

null clause. A clause consisting of only blanks, comments, or both. A null clause is ignored unless it includes a comment, in which case it will be traced, if appropriate.

null string. A string with no characters and has a length of zero.

number. A character string consisting of one or more decimal digits optionally prefixed by a plus or minus sign, and optionally including a single period that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase) followed optionally by a plus or minus sign then followed by one or more decimal digits defining the power of 10.

numeric pattern. A pattern that specifies, by column number, how input data is to be parsed.

O

operand. Information entered with a command name to define the data on which a command processor operates and to control the execution of the command processor.

operator. A token that specifies what type of action to be done on one or two terms. There are four types of operators: concatenation, arithmetic, comparison, and logical.

P

parameter. A variable that is given a constant value for a specified application and that may denote the application.

parameter list (PLIST). In CMS, a string of 8-byte arguments that call a CMS command or function. The first argument must be the name of the command or function to be called. General register 1 points to the beginning of the parameter list.

parse. A method of splitting up a sequence of characters with a template and assigning the resultant values to variables.

pattern. In parsing, specifies where a sequence of characters are split up based on the matching of strings or based on the position within the string. Patterns can be specified as a variable or string and are usually removed from the data during parsing.

PLIST. Parameter list.

positional pattern. Patterns that cause parsing to occur on the basis of position within the input string. They take the form of signed or unsigned whole numbers.

Procedures Language/VM.. A component of VM/SP. It contains the VM/SP System Product Interpreter, which processes the REXX language. This component contains the VM/SP implementation of the Systems Application Architecture Procedures Language in addition to the VM/SP System Product Interpreter function available in VM/SP Releases 3, 4, and 5. Procedures Language/VM provides a single source base for the VM/SP System Product Interpreter in both the CMS and GCS environment.

PROFILE EXEC. A special EXEC procedure with a file name of PROFILE that a user can create. The procedure is usually executed immediately after CMS is loaded into a virtual machine (also known as IPL CMS).

program stack. Temporary storage for lines (or files) being exchanged by programs that execute under CMS. See *console stack.*

R

RC. A REXX special variable set to the return code from any executed host command or subcommand. It is also set to the return code when the conditions ERROR, FAILURE, and SYNTAX are trapped.

relative pattern. Patterns that cause parsing to occur on the basis of the relative position in the input string.

They take the format of signed or unsigned whole numbers.

repetitive loop. A loop that allows a set of instructions to be performed a certain number of times. This number can be specified as: none, a whole number, by the subkeyword, FOREVER, or by a control variable.

RESULT. A REXX special variable that is set by the RETURN instruction in a CALLed routine. RESULT is dropped if the called routine does not return a value.

Restructured Extended Executor (REXX) language. A general-purpose programming language, particularly suitable for EXEC procedures, XEDIT macros, or programs for personal computing. Procedures, XEDIT macros, and programs written in this language can be interpreted by the System Product Interpreter. Contrast with *CMS EXEC language* and *EXEC 2 language*.

REXX EXEC. An EXEC procedure or XEDIT macro written in the REXX language and processed by the System Product Interpreter. Synonymous with *REXX program*.

REXX language. Restructured Extended Executor language.

REXX program. Synonym for *REXX EXEC*.

routine. A series of instructions invoked with the CALL instruction or as a function. It can be either internal or external to the user's program.

S

semicolon. A token that indicates the end of a clause and implied by the interpreter in three cases: by a line-end, by certain keywords, and by a colon if it follows a single symbol.

SIGL. A REXX special variable that contains the line number of the clause currently executing when the last transfer of control to a label took place.

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

special character. Tokens that act as delimiters when found outside a literal string. They include the following characters: , : ;) (and the individual characters from the operators.

special variable. Variables set automatically by the language processor and include: RC, RESULT, and SIGL.

SPOOL. Simultaneous peripheral operations online.

spool file block. A 4096-byte buffer that contains control information, in addition to records. Synonymous with *spool file buffer linkage block*.

spool file buffer linkage block. Synonym for *spool file block*.

stack. See *console stack* and *program stack*.

stem. Any combination of characters where the last character is a period and the first character cannot be a digit (0-9) or a period.

subcommand. The commands of processors such as EDIT or System Product Editor (XEDIT) that run under CMS.

subkeyword. Symbols reserved by the language processor within the clause of individual instructions. For example, the symbol FOREVER is a subkeyword of the DO instruction.

supervisor call instruction (SVC). An instruction that interrupts a program being executed and passes control to the supervisor so that it can do a specific service indicated by the instruction.

SVC. Supervisor call instruction.

symbol. Any combination of alphabetic or numeric characters (A-Z, a-z, 0-9) and the characters @ # $ ¢ . ! ? and underscore.

synonym. In CMS, an alternative command name defined by the user as equivalent to an existing CMS command name. Synonyms are entries in a CMS file with a file type of SYNONYM. Entering the SYNONYM command allows use of those synonyms until that terminal session ends or until the use of synonyms is revoked by entering the SYNONYM command with no operands.

syntax. The rules for the construction of a command or program.

System Product Editor. The CMS facility, comprising the XEDIT command and XEDIT subcommands and macros, that lets a user create, change, and manipulate CMS files.

System Product Interpreter. The language processor of the VM/SP operating system that processes procedures, XEDIT macros, and programs written in the REXX language.

Systems Application Architecture. A defined set of interfaces, conventions, and protocols that can be used across various IBM systems.

T

template. A guide that allows strings to be parsed by words (delimited by blanks), by explicit matching of strings, or by specifying numeric positions.

term. An element of an expression that consists of literal strings, symbols, or function calls.

terminal input buffer. Holds lines entered at the user's terminal until CMS processes them.

tokenized PLIST (parameter list). A string of doubleword aligned parameters occupying successive doublewords.

trace. In REXX, a means of tracking the interpretation of a program. Tracing is primarily used for debugging.

U

user. Anyone who requests the services of a computing system.

user-written CMS command. Any CMS file created by a user that has a file type of MODULE or EXEC. Such a file can be executed as if it were a CMS command by issuing its file name, followed by any operands or options expected by the program or EXEC procedure.

user ID. A one-to eight character alphanumeric symbol identifying each virtual machine.

V

variable pattern. A parsing pattern that uses variables to specify where a string of characters are parsed. The value of the variable can be set by the user and can change during execution.

variable symbol. In an EXEC procedure, a symbol that is assigned a value by the user, or in some cases by the System Product Interpreter. The value of a variable symbol can be tested and changed using control statements.

virtual console. A console simulated by CP on a terminal such as a 3270. The virtual device type and I/O address are defined in the VM/SP directory entry for that virtual machine.

virtual machine (VM). A functional equivalent of a real machine.

Virtual Machine/System Product (VM/SP). An IBM licensed program that manages the resources of a single computer so that multiple computing systems appear to exist. Each virtual machine is the functional equivalent of a *real* machine.

VM/SP. Virtual Machine/System Product.

VM READ screen status. For a display terminal used as a virtual console under VM/SP, an indicator located in the lower right of the screen that displays when the user's virtual machine is not executing, but is waiting for a response or a request for work from the user.

W

whole number. An integer or a number that has a zero decimal part. Whole numbers are not usually expressed by the language processor in exponential notation.

window. An area on the physical screen where virtual screen data can be displayed. Windowing lets the user do such functions as defining, positioning, and overlaying windows; scrolling backward and forward through data; and writing data into virtual screens.

X

XEDIT. See *System Product Editor*.

XEDIT macro. (1) A procedure defined by a frequently used command sequence to do a commonly required editing function. A user creates the macro to save repetitious rekeying of the sequence, and invokes the entire procedure by entering a command (that is, the macro file's file name). The procedure can consist of a long sequence of XEDIT subcommands, CMS and CP commands or both, along with REXX or EXEC 2 control statements to control processing within the procedure. (2) A CMS file with a file type of *XEDIT*.

Bibliography

Related Publications

You should have a copy of:

VM/SP System Product Interpreter Reference, SC24-5239.

This manual is a reference manual. It lists the REXX error messages and describes instructions, functions, debugging aids, and parsing. It is suitable for experienced programmers, particularly those who have used another high-level language (such as, PL/I, Algol, and Pascal).

You may also need to refer to:

Common Programming Interface Procedures Language Reference, SC26-4358

VM/SP Application Development Reference for CMS, SC24-5284

VM/SP CMS Command Reference, SC19-6209

VM/SP CP General User Command Reference, SC19-6211

VM/SP CP System Command Reference, SC24-5402

VM/SP System Product Editor Command and Macro Reference, SC24-5221

VM/SP System Messages and Codes, SC19-6204

VM/SP System Messages Cross-Reference, SC24-5264.

Other tutorials and user's guides that may be useful are:

VM/SP Application Development Guide for CMS, SC24-5286

VM/SP CMS Primer, SC24-5236

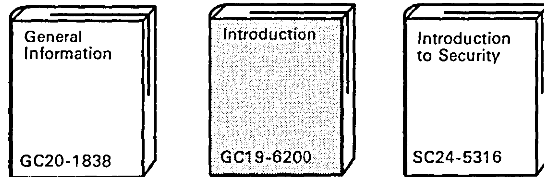
VM/SP CMS Primer for Line-Oriented Terminals, SC24-5242

VM/SP CMS User's Guide, SC19-6210

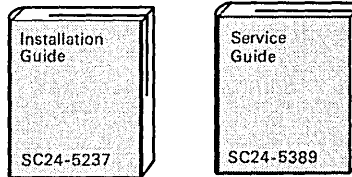
VM/IS Writing Simple Programs with REXX, SC24-5357.

VM/SP RELEASE 6 LIBRARY

Evaluation

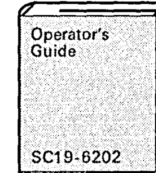


Installation and Service

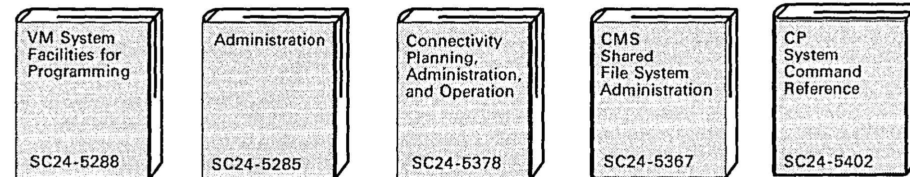


Planning

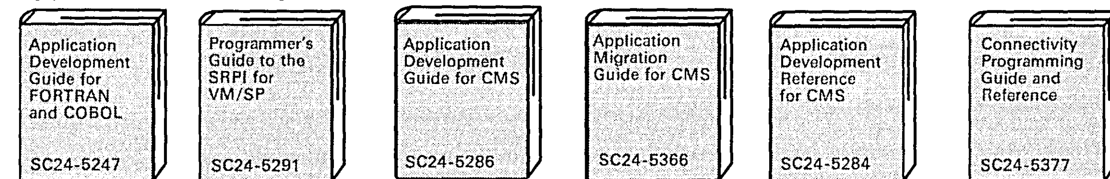
Operation

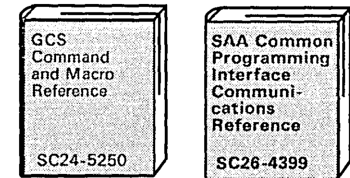


Administration

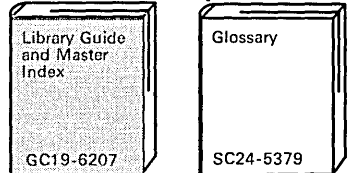


Application Development

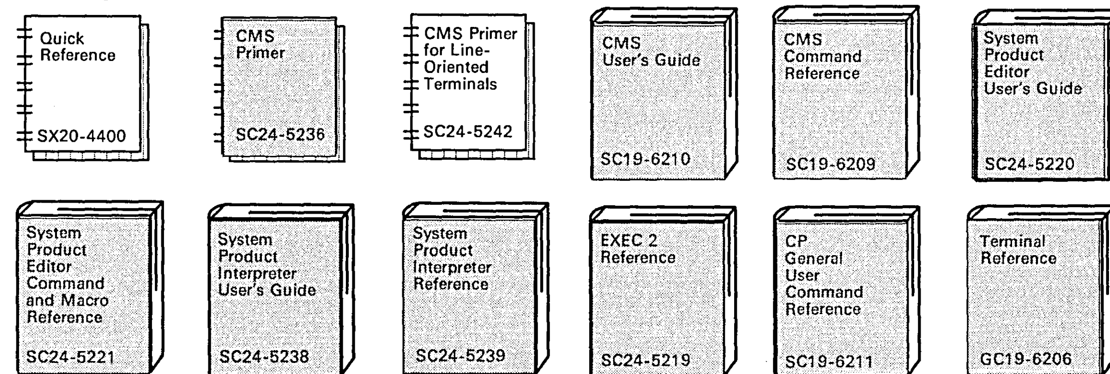




Index/Glossary



End Use



one copy of each shaded manual received with product tape

VM/SP RELEASE 6 LIBRARY

Diagnosis

- System Messages and Codes — SC19-6204
- System Messages Cross-Reference — SC24-5264
- Service Routines Program Logic — LY20-0890
- Interactive Problem Control System Guide and Reference — SC24-5260
- Diagnosis Guide — LY24-5241
- CP Diagnosis Reference — LY20-0892
- CP Data Areas and Control Blocks — LY24-5220
- CMS Diagnosis Reference — LY20-0893
- CMS Data Areas and Control Blocks — LY24-5221
- VM CP Trace Table (Poster) — SX24-5225
- Problem Determination Summary — SX24-5224

Reference Summaries

- SP Editor Command Language Reference Summary — SX24-5122
- EXEC 2 Language Reference Summary — SX24-5124
- CMS Primer Summary of Commands — SX24-5151
- SP Interpreter Reference Summary — SX24-5126
- CMS Primer for Line-Oriented Terminals Summary of Commands — SX24-5159
- HELP Facility Introduction — SX24-5221
- CP General User Command Reference Summary — SX24-5219
- CP System Command Reference Summary — SX24-5222
- CMS Command Reference Summary — SX24-5220
- VM Summary of End Use Tasks and Commands (Poster) — SX24-5173

Auxiliary Communication Support

- VTAM Installation and Resource Definition — SC23-0111
- VTAM Customization — SC23-0112
- VTAM Operation — SC23-0113
- VTAM Messages and Codes — SC23-0114
- VTAM Programming — SC23-0115
- VTAM Diagnosis Guide — LY30-5601
- VTAM Programming for LU 6.2 — SC30-3400
- VTAM Data Areas (VM) — LY30-5593
- VTAM Reference Summary — LY30-5600
- VM/Pass-Through Facility Overview — GC24-5373
- VM/Pass-Through Facility: Managing and Using — SC24-5374
- RSCS Exit Customization — SH24-5197
- RSCS General Information — GH24-5055
- RSCS Planning and Installation — SH24-5057
- RSCS Messages and Codes — SH24-5196
- RSCS Operation and Use — SH24-5058
- RSCS Diagnosis Reference — LY24-5228
- RSCS Reference Summary — SX24-5135

Index

A

ABBREV function 95
abbreviation of information 95
abuttal 41, 78
accuracy in calculations, changing 73
addition 65
addition operator 41, 65
ADDRESS instruction 141
AND operator 45, 47
ARG function 53
ARG instruction 52, 105, 201, 205
  description of 112
  example of 112
  using literal patterns 113
arguments
  of a CALL instruction 201
  of a function 50, 52
  of a subroutine 202
  parsing 52, 112, 210
arithmetic
  checking data 64
  description 63
array
  description of 22
  using compound symbols 22
  with more than one dimension 32
assembler language functions 58
assignment
  description of 10, 19
  examples of 19

B

blank (concatenation operator) 41, 78
blanks removed 12
buffers 133
built-in functions 1, 50
BY expression 187

C

CALL instruction 201
Callable Services Library(CSL) 159
character
  comparing 93
  conversion of 100
  priority of 37
  sorting 90
character priority when comparing 94
character strings 10, 40
checking data 64
clause
  description of 10

clause *(continued)*
  null 11
  separating 11
  spanning more than one line 11
clause delimiter 11
closing a file 146
CMS environment 119, 141
CMS Primer 2
CMS (Conversational Monitor System) commands
  DROFBUF 136
  EXECIO 138
  FINIS 146
  HT 158
  LISTFILE 156
  MAKEBUF 136
  putting data onto a program stack 131, 136
  RT 158
  SORT 158
  suppressing messages issued by 128
  taking data from a program stack 132, 137
  using 119
column formatting 80
comma to indicate continuation of a clause 11
COMMAND environment 141
comments 9, 10
COMPARE function 83, 95
comparison operators 41, 44, 93
comparisons
  allowing approximation 97
  characters 93
  exact 97
  fuzzy arithmetical 97
  numbers 93
  priority of characters 94
  strings 95
compound symbols
  description of 22
  for repeated substitution 16
  use of a period 22
concatenation 78
concatenation operator 41, 78
conditional loops
  description of 190
  DO FOREVER instruction 190, 194
  DO UNTIL instruction 193
  DO WHILE instruction 191, 193
  LEAVE instruction 190, 197
console stack 133
continuation
  of a clause 11
  of expression in SAY instruction 105
Control Program (CP) commands
  obtaining a reply from 138

Control Program (CP) commands *(continued)*
  suppressing messages issued by 138
  using 122
control variable 186, 196
Conversational Monitor System (CMS) commands
  DROFBUF 136
  EXECIO 138
  FINIS 146
  HT 158
  LISTFILE 156
  MAKEBUF 136
  putting data onto a program stack 131, 136
  RT 158
  SORT 158
  suppressing messages issued by 128
  taking data from a program stack 132
  using 119
conversion between hexadecimal and decimal characters 100
COPIES function 79
copying a string 79
COPY, XEDIT subcommand 137
correcting your program 223
CP environment 122
CP (Control Program) 61
CP (Control Program) commands
  obtaining a reply from 138
  suppressing messages issued by 138
  using 122
creating a new file from an existing file 152

D

dangling ELSE 176
data
  prompting user for 106
  putting onto a program stack 131
  taking from a program stack 132
data types, checking 64
DATATYPE function 64
debugging 126
decimal number
  converting 100
  description of 63
decision making 173
delimiters
  clause 11
  comment 9
DELWORD function 85
derived name 22
designing a program 220
DIGITS 74
DIGITS option of NUMERIC instruction 73
displaying a file on your screen 146
division 65
division operator 41, 65
DO FOREVER instruction 190, 194
DO instruction
  BY expression 187
  conditional loop 190
  control variable 186, 196
  DO FOREVER instruction 190, 194
  DO UNTIL instruction 193
  DO WHILE instruction 191, 193
  END keyword 60
  ITERATE instruction 198
  LEAVE instruction 190, 197
  non-looping 77, 173, 179
  repetitive loops 184
DO UNTIL instruction 59, 153, 193
DO WHILE instruction 191, 193
DROP instruction 31
DROPBUF command 137, 153
dropping variables 31
duplicate names 27
duplicating strings 79

E

E (exponent symbol) 67
ELSE keyword
  dangling 176
  NOP instruction 181
  of IF instruction 174
END keyword
  of DO instruction 60
  of SELECT instruction 178
environments
  CMS 119, 141
  COMMAND 141
  CP 122
equal operator 41, 97
evaluating expressions
  order of 37, 41
  using parentheses 37, 42
exact comparison operators 41, 97
exactly equal operator 41, 97
exclusive OR operator 41
EXEC
EXECIO command 138, 143, 145, 153
EXIT instruction 124, 171, 200
exponent 67
exponential notation
  description of 63, 67
  NUMERIC DIGITS instruction 73
  significant digits 73
  specifying 71
exponentiation 73
exponentiation operator 41, 73
EXPOSE keyword of PROCEDURE instruction 29, 30
expressions
  description of 35

expressions *(continued)*
evaluating
order of 37, 41
using parentheses 37, 42
using the TRACE instruction 39
in an assignment 19
in IF clause 173
parsing 114
text 78
using parentheses 37
using TRUE and FALSE 44
external functions
See external routines
external routines
functions 50, 53, 209
subroutines 209, 211
EXTRACT, XEDIT subcommand 165

F

FALSE expression 44
Features of REXX 1
FIFO (first-in/first-out) 133
file
closing 146
creating 152
displaying 146
modifying 154
reading 145
sorting 156
writing 143
file processing 143, 156
creating a file 152
reading a file 145
writing a file 143
finding
phrase in a string 85
string in another string 83
FINIS command 146, 154
fixed point number
description of 67
specifying 70
floating-point number
description of 67
specifying 71
FORMAT function 69, 74
formatting output
lining up numbers 69
putting in columns 80
full screen menus 168
function call 49
functions 86
ABBREV 95
ARG 53
arguments for 50, 52
built-in 50
COMPARE 95
functions *(continued)*
COPIES 79
DATATYPE 64
DELWORD 85
description of 49
differences with subroutines 209
DIGITS 74
example of 52
external 50, 53, 209
FORMAT 69
FUZZ 98
HALF 49
internal 50, 53, 57, 209
LASTPOS 83
LEFT 79
LENGTH 79
MAX 50
OVERLAY 90
POS 83
RANDOM 50
returning from 52
search order 209
SIGN 74
similarities with subroutines 209
SOURCELINE 86
SUBSTR 78
SUBWORD 85
SYMBOL 31
TRANSLATE 103
TRUNC 74
user-written 50, 52
using the ARG instruction 52
VALUE 16
VERIFY 103
WORD 85
WORDINDEX 85
WORDLENGTH 85
WORDPOS 85, 103
WORDS 85
written in Assembler language 58
FUZZ 98
FUZZ option
of NUMERIC instruction 97
fuzzy arithmetical comparison 97

G

getting arguments for a function or routine 50, 52
getting data from the command line 52, 112
getting data when you are prompted 106
getting out of loops 60, 196
GOTO considered harmful 171
greater than operator 41, 44, 93
greater than or equal to operator 41
groups of instructions 77

H

HALF function 49
Halt Interpretation (HI) immediate command 60, 214
help, providing, to explain a program 86
hexadecimal
  converting 100
  description of 100
HI (Halt Interpretation) immediate command 60, 214
HT (Halt Type) immediate command
  SET CMSTYPE HT command 157

I

IF instruction
  description of 173
  ELSE keyword 174
  THEN keyword 173
increasing accuracy in calculations 73
information abbreviated 95
instructions
  ADDRESS 141
  ARG 52
  CALL 201
  description of 10
  DO 77
  DO FOREVER 190
  DO WHILE 191
  DROP 31
  EXIT 200
  groups of 77
  IF 173
  INTERPRET 17
  ITERATE 198
  LEAVE 190
  NOP 181
  NUMERIC DIGITS 73
  NUMERIC FUZZ 97
  PARSE ARG 113
  PARSE PULL 5, 106
  PARSE VALUE 114
  PARSE VAR 114
  PROCEDURE 29
  PULL 106
  PUSH 132
  QUEUE 132
  RETURN 52
  SAY 105
  SELECT 178
  SIGNAL 212
  TRACE 39
  UPPER 106
integer 63
integer division operator 41, 65
internal functions
  See internal routines
internal routines
  functions 50, 53, 57, 209
internal routines *(continued)*
  subroutines 29, 82, 206, 209
INTERPRET instruction 17
interpreter 2, 9
issuing commands
  to CMS 119
  to CP 122
ITERATE instruction 198

J

jumping through your program 198, 212
justified left 79

K

keywords
  of DO instruction
    END 60
  of IF instruction
    ELSE 174
    THEN 173
  of SELECT instruction
    END 178
    OTHERWISE 178
    THEN 178
    WHEN 178
  to manipulate program stack 133

L

label
  description of 11
  in a CALL instruction 203
LASTPOS function 83
LEAVE instruction 154, 190, 196
leaving loops 60, 196
leaving your program 200
LEFT function 79
left justified 79
LENGTH function 79, 85, 90
less than operator 42, 44, 93
less than or equal to operator 42
LIFO (last-in/first-out) 133
LISTFILE command 157
literal patterns in parsing 113
logical operator 47
loops
  conditional 190
  control variable 186, 196
  description of 59, 184
  DO FOREVER instruction 190, 194
  DO UNTIL instruction 59, 193
  DO WHILE instruction 191, 193
  ITERATE instruction 198
  LEAVE instruction 190, 197
  leaving 60, 196
  repetitive 184

loops *(continued)*
  skipping instructions 198, 212

M

macros 161
MAKEBUF command 137, 153
manipulating the program stack 133
mantissa 67
MAX function 50
menu, full screen 168
messages
  suppressing
    FILE NOT FOUND 141
    from CMS commands 128
    from CP commands 138
  XEDIT, displaying 164
minus operator 41, 65
modifying a file, precautions when 154
multiple clauses on a line 11
multiplication 65
multiplication operator 41, 65

N

naming variables 20
NOP instruction 181
not equal operator 41, 97
not exactly equal operator 41, 97
not greater than operator 41
not less than operator 42
NOT operator 41, 47
null clauses 11
numbers
  comparing 93
  determining the sign 74
  exponential notation 67
  fixed point 67
  floating point 67
  power of 73
  range of 67
  rounding 74
  truncating 74
  types of 63
  whole 63
NUMERIC DIGITS instruction 73
NUMERIC FUZZ instruction 97

O

operator
  comparison 42, 44, 93
  list of 41
  logical 47
  prefix 41
  priority of 37, 41
  using parentheses 37, 42
OR operator 45, 47
order of evaluation 37, 41, 42
order of precedence 37
OTHERWISE keyword 178
output format 69, 80
OVERLAY function 90
overlaying one string onto another 90

P

parentheses 37, 42
PARSE ARG instruction 113
PARSE instruction 5
PARSE PULL instruction 5, 106
PARSE VALUE instruction 114
PARSE VAR instruction 114
parsing
  arguments 52, 112, 210
  data when you are prompted 106
  expressions 114
  use of a period 110
  using literal patterns 113
  using patterns 117
  variables 114
  words 109
patterns used in parsing 117
period
  as a placeholder in parsing 110
  in compound symbols 22
phrase 85
placeholder, period, in parsing 110
plus operator 41, 65
POS function 83
power of a number 73
precedence
  of characters 94
  operators 37, 41
prefix operators 41
Primer for CMS 2
priority of characters 94
priority of operators 37, 41
PROCEDURE instruction
  description of 29
  EXPOSE keyword 29
PROFILE XEDIT 166
program stack
  as a queue 132
  as a stack 132
  description of 131
  extensions (buffers) 133
  putting data onto 131, 136
  taking data from 132, 137
  using 134
  with SFS sources 135
programs
  correcting 223
  description of 171

programs *(continued)*
designing 220
leaving 200
prompting user for data 106
PULL instruction
description of 106
using 6, 109, 132, 133
PUSH instruction 132, 133
putting data onto the program stack 131, 136
putting words into variables 109

Q

queue described 132
QUEUE instruction 132, 133, 137
quotation marks 10, 121
quotes 10, 121

R

RANDOM function 50
range of numbers 67
RC special variable 124
reading files 145
reading plan 3
recursive calls
See CALL instruction
remainder 65
remainder operator 41, 65
repeated substitution 16
repetitive loops 184
RESULT reserved symbol 205
return codes
CMS and CP 123
REXX 13
RETURN instruction 52, 201, 205
returning from a function or routine 52, 205
rounding numbers 40, 74
RT Immediate command
SET CMSTYPE RT command 157

S

Say 105
search order for subroutines and functions 209
SELECT instruction
description of 172, 178
END keyword 179
example of 20, 180
OTHERWISE keyword 179
THEN keyword 179
WHEN keyword 179
separating clauses 11
SFS
See Shared File System(SFS)
Shared File System(SFS)
CSL routines, use with 159
modifying SFS files 154
Shared File System(SFS) *(continued)*
processing files in 159
program stack, use with 135
writing programs with 5
SIGL
special variable 124
storing line numbers 126, 213
SIGN function 74, 103
SIGNAL instruction
description of 126, 213
example of 129
restrictions 212
usage 212
signed number 63
significant digits 73
skipping instructions in a loop 198, 212
SORT command 157
sorting
a file 156
characters 90
SOURCELINE function 86, 153
special variables
RC 124
Result 124
SIGL 124
splitting
clauses 11
data 109
stack described 132
stem
description of 24
example of 24
strictly greater than operator 41
strictly greater than or equal to operator 41
strictly less than operator 41
strictly less than or equal to operator 41
strictly not greater than operator 41
strictly not less operator 41
string
comparing 95
copying 79
description of 10
duplicating 79
examples of 10
overlaying 90
subcommands in XEDIT 161
subroutines
ARG instruction 205
arguments for 202
description of 201
differences with functions 209
example of 29, 82
external 209, 211
formatting output 82
internal 30, 82, 206, 209
PROCEDURE instruction 29
protecting variables 29

subroutines *(continued)*
RETURN instruction 205
search order 209
sharing variables 30
similarities with functions 209
substituting
compound symbols 16
symbols 15
using the INTERPRET instruction 17
using the VALUE function 16
variables 16
SUBSTR function 78, 85, 90
substring 78
subtraction 65
subtraction operator 41, 65
SUBWORD function 85
symbol
compound 22
description of 19
determining if it is a variable 31
duplicate names of 27
substituting 16
SYMBOL function 31
syntax error
example of 12
FORMAT function 69
syntax, description of 12
Systems Application Architecture (SAA) 2

T

tables 80
tabulating text output 80, 82
taking data from a program stack 132, 137
term 36
terminal input buffer 133
text expressions 78
THEN keyword
NOP instruction 181
of IF instruction 173
of SELECT instruction 178
TRACE
Errors 126
Intermediate results 39
Normal 39
Results 39, 40, 126
TRACE instruction 39
tracing
description of 39
example 39
TRANSLATE function 103
translating
between character, hexadecimal, decimal 100
character sets 99, 103
examples of 100
to uppercase 5, 12, 106
TRANSLATE function 103

translating *(continued)*
VERIFY function 103
TRUE expression 44
TRUNC function 74
truncating numbers 74
types of data 40

U

UPPER instruction 95, 106
uppercase 5, 12, 106
user-written functions 50, 52

V

VALUE function 16
variables
description of 11, 19
dropping 31
example of 20
length of 79
naming conventions 20
parsing 114
protecting 29
setting of 20, 109
sharing between routines 30
substituting 16
XEDIT, known to 165
variables, special
RC 124
Result 124
SIGL 124
VERIFY function 103

W

WHEN keyword 178
whole numbers 63
word
description of 91
functions using 91
parsing 109
WORD function 85, 91
WORDINDEX function 85
WORDLENGTH function 85
WORDPOS function 85, 86, 103
WORDS function 85, 91
writing files 143
writing lines to the screen 105

X

XEDIT (System Product Editor)
EXTRACT subcommand 165
generating full screen menus 168
macros
description of 162
examples of 162
naming 162

XEDIT (System Product Editor) *(continued)*
  macros *(continued)*
    return codes 162
  messages 164
  profile 166
  subcommands 161

Special Characters

. (as a placeholder) 110
. (in compound symbols) 22
< (less than operator) 42, 44, 93
<< (strictly less than operator) 41
<<= (strictly less than or equal to operator) 41
<= (less than or equal to operator) 42
+ (addition operator) 41, 65
+ (prefix operator) 41
| (inclusive OR operator) 45, 47
|| (concatenation operator) 41, 78
& (AND operator) 45, 47
&& (exclusive OR operator) 41
* (multiplication operator) 41, 65
** (exponentiation operator) 41, 73
*/ comment delimiter 9
¬ (NOT operator) 41
¬< (not less than operator) 42
¬<< (strictly not less than operator) 41
¬> (not greater than operator) 41
¬>> (strictly not greater than operator) 41
¬= (not equal operator) 41, 97
¬== (not exactly equal operator) 41, 97
/ (division operator) 41, 65
/* comment delimiter 9
// (remainder operator) 41, 65
/= (not equal operator) 41, 97
/== (not exactly equal operator) 41, 97
% (integer division operator) 41, 65
> (greater than operator) 41, 44, 93
>> (strictly greater than operator) 41
>>= (strictly greater than or equal to operator) 41
>= (greater than or equal to operator) 41
#CP I CMS 61
= (equal operator) 41, 45, 93, 97
== (exactly equal operator) 41, 97
- (prefix operator) 41
- (subtraction operator) 41, 65
\< (not less than operator) 42
\<< (strictly not less than operator) 41
\> (not greater than operator) 41
\>> (strictly not greater than operator) 41
\= (not equal operator) 41, 97
\== (not exactly equal operator) 41, 97

Program Number
5664-167

File Number
S370/4300-39

Printed in U.S.A.

VM/SP
System Product Interpreter User's Guide
Order No. SC24-5238-04

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? ___YES ___NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ______________________________

Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

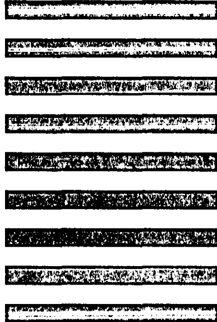
SC24-5238-04

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE
NECESSARY
IF MAILED
IN THE
UNITED STATES



BUSINESS REPLY MAIL
FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape | Please Do Not Staple | Fold and tape

®

VM/SP
System Product Interpreter User's Guide
Order No. SC24-5238-04

READER'S COMMENT FORM

Is there anything you especially like or dislike about this book? Feel free to comment on specific errors or omissions, accuracy, organization, or completeness of this book.

IBM may use or distribute whatever information you supply in any way it believes appropriate without incurring any obligation to you, and all such information will be considered nonconfidential.

Note: Do not use this form to report system problems or to request copies of publications. Instead, contact your IBM representative or the IBM branch office serving you.

Would you like a reply? __YES __NO

Please print your name, company name, and address:

____

____

____

____

IBM Branch Office serving you: ________________________________

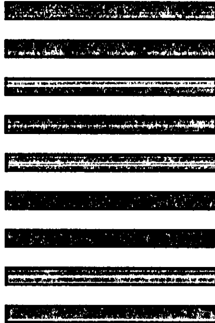
Thank you for your cooperation. You can either mail this form directly to us or give this form to an IBM representative who will forward it to us.

SC24-5238-04

Reader's Comment Form

CUT
OR
FOLD
ALONG
LINE

Fold and tape — Please Do Not Staple — Fold and tape



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NY

POSTAGE WILL BE PAID BY ADDRESSEE:

INTERNATIONAL BUSINESS MACHINES CORPORATION
DEPARTMENT G60
PO BOX 6
ENDICOTT NY 13760-9987

Fold and tape — Please Do Not Staple — Fold and tape

Program Number
5664-167

File Number
S370/4300-39

Printed in U.S.A.