User's Guide to the REXX Language

REXX Language

Document revision 5

4 October 2023

Mindus SARL

NetPhantom®

Mindus SARL

1 Rue du Gabian
MC-98000 Monaco
MONACO

Telephone: **+377 99 90 32 66**
Web: **https://netphantom.com**
E-mail: **info@netphantom.com**

Support

Phone: **+377 99 90 32 66**
E-mail: **support@netphantom.com**

Contents

1 Getting Started in REXX

1.1 Introduction to REXX

A REXX procedure is a program that consists of a series of tasks in one common processing file or batch file. Many languages can be used to write programs. BASIC, which is widely used in home computing, has very few rules, but it requires writing many lines of code for an intricate program. Languages such as PL/1, COBOL, C, APL, and PASCAL have more rules, but allow you to write more functions in fewer lines.

REXX combines the simplicity of a programming language such as BASIC with features that exist in more powerful languages such as writing fewer lines. It is easier to learn, because it uses familiar words and concepts. REXX allows you to do simple tasks, yet has the ability to handle complex tasks.

1.2 Writing a REXX Procedure

We shall write the following REXX procedure using a text editor. To write a REXX procedure named HELLO.CMD while using the text editor follow these instructions:

1. Create a text file named HELLO.CMD.

2. Type the procedure HELLO.CMD, as follows:

```
/* An introduction to REXX */
SAY "Hello! I am REXX"
SAY "What is your name?"
PULL who
 IF who = ""
  THEN
  SAY "Hello Stranger"
 ELSE
  SAY "Hello" who
EXIT
```

3. Save the file and exit from the text editor.

Now you are ready to run the REXX procedure you have written. Type the name of the procedure at the command prompt and press Enter.

```
hello
```

When the procedure pauses, you can either type your name or press Enter to see the other response. A brief description of each part of HELLO.CMD follows:

```
/* An introduction to REXX */
```

A comment explains what the procedure is about. A comment starts with a /* and ends with a */. All REXX procedures must start with a comment on line one and column one of the file. The comment tells the command processor that the procedure being run is a REXX procedure and distinguishes it from simple batch files.

```
SAY "Hello! I am REXX."
SAY "What is your name?"
```

These instructions cause the words between the quotation marks to be displayed on your screen.

```
PULL who
```

The PULL instruction reads the response entered from the keyboard and puts it into the system's memory. Who is the name of the place in memory where the user's response is put. Any name can be used with the PULL instruction.

```
IF who = " "
```

The IF instruction tests a condition. The test in this example determines if who is empty. It is empty if the user types a space and presses Enter or just presses Enter.

```
THEN
```

Specifies that the instruction that follows is to be run, if the tested condition is true.

```
SAY "Hello Stranger"
```

Displays Hello Stranger on the screen.

```
ELSE
```

Specifies that the instruction that follows is to be run if the tested condition is not true.

```
SAY "Hello" who
```

Displays Hello on the screen, followed by whatever is in who.

```
EXIT
```

This instruction causes the procedure to stop.

Here is what would happen if a person named Bill tried the HELLO program:

```
Hello! I am REXX.
What is your name?
Bill
Hello BILL
```

If Bill does not type his name, but types a blank space, this happens:

```
Hello! I am REXX.
What is your name?
Hello Stranger
```

2 Using Fundamental REXX Elements

2.1 Basic Elements of REXX

This section gives you a chance to try some of the elements used in writing REXX procedures. Do not worry about making mistakes because you will be guided through the steps.

> **Note:** When writing a REXX procedure, it is best to use one line for each element. If you want an element to span more than one line, you must put a comma (,) at the end of the line to indicate that the element continues on the next line. If you want to put more than one element on a line, you must use a semicolon (;) to separate the elements.

A REXX procedure can contain any or all of the following elements:

- Comments
- Strings
- Instructions
- Operating System Commands
- Assignments
- Labels
- Internal Functions

2.2 Comments

All REXX procedures must begin with a comment starting in column one of line one. The comment tells the command interpreter it is about to read and run a REXX procedure. The symbols for a comment are:

/*	To mark the start of a comment
*/	To mark the end of a comment

When the interpreter finds a /*, it stops interpreting; when it encounters a */, it begins interpreting again with the information following the symbol. The comment can be a few words, no words, or several lines, as in the following examples:

```
/* This is a comment. */
```

or,

```
SAY "'Be Prepared!'" /* This comment is on the same line as
the instruction and continues on to the next line */
```

You can use only /* */ to start a REXX procedure, but it is better to put a brief description of the procedure between the comment symbols.

The comment can indicate the purpose of the procedure, the kind of input it can handle, and the kind of output it produces. Comments help you understand the procedure when you read it later, perhaps to add to it, improve it, or use it elsewhere in the same procedure or another procedure.

When you write procedures, remember that others may need to use or modify them. It is a good idea to add comments to the instructions so that anyone can understand each step. If you do not use a procedure often, it is helpful to have reminders to aid your memory. In general, explain your procedure well enough so that others can understand it.

2.3 Strings

A string is any group of characters inside single or double quotation marks. Either type of quotation marks can be used, but the beginning and the ending mark must match. The interpreter stops interpreting when it sees a quotation mark

and the characters that follow remain as they were typed, with uppercase and lowercase letters. The interpreter resumes interpreting when it sees a matching quotation mark. For example:

```
'The Greatest Show on Earth'
"The President leads his country"
```

are both strings.

To use an apostrophe (single quotation mark) or double quotation marks within a string, use the other quotation mark around the whole string. For example:

```
"Don't count your chickens before they hatch."
```

or

```
'Do not count your "chickens" before they hatch.'
```

You also can use a pair of quotation marks (the same as those used to mark the string) as follows:

```
SAY "Mary said  ""He's here."""
```

This is interpreted by REXX as: Mary said "He's here."

2.4 Instructions

An instruction tells the system to do something. Instructions can contain one or more assignments, labels, or commands and they usually start on a new line. The following are explanations of some of the more common instructions.

SAY Instruction

The format for the SAY instruction is:

```
SAY expression
```

The expression can be something you want displayed on the screen or something to be computed, such as an equation:

```
SAY 5 + 6 "= eleven"
```

This displays 11 = eleven

With the SAY instruction, anything not in quotation marks is changed to uppercase or is processed. If you want something to appear exactly as it is typed, enclose it in quotation marks.

PULL and PARSE PULL Instructions

In a procedure, the usual sequence of instructions is to use SAY to ask a question and PULL to receive the answer. The response typed by the user is put into system memory.

The following procedure does not work correctly if the PULL instruction comes before the SAY instruction.

What happens when the following procedure, NAME.CMD, is run?

```
/* Using the PULL Instruction */
SAY "Enter your name"
PULL name          /* Puts response
from user into memory */
SAY "Hello" name
EXIT
```

NAME.CMD puts a name in memory and then displays that name anywhere in the file that the word name appears without the protection of single or double quotation marks. If you tried the NAME procedure, you probably noticed that your name was changed to uppercase. To keep the characters as you type them, use the PARSE PULL instruction. Here is an example called CHITCHAT.CMD that uses the PARSE PULL instruction:

```
/* Using the PARSE PULL Instruction */
SAY "Hello! Are you still there?"
SAY "I forgot your name. What is it?"
PARSE PULL name
SAY name "Are you going to Richard's seminar?"
PULL answer
IF answer = "YES"
 THEN
 SAY "Good. See you there!"
IF answer = "NO"
 THEN
 SAY "Sorry,  We will miss your input."
EXIT
```

The PARSE PULL instruction reads everything from the keyboard exactly as it is typed, in uppercase or lowercase. In this procedure, the name is displayed just as you type it. However, answer is changed to uppercase characters because the PULL instruction was used. This ensures that if yes, Yes, or YES is typed, the same action is taken.

EXIT Instruction

The EXIT instruction tells the procedure to end. The EXIT instruction should be used in a procedure that contains subroutines. Although the EXIT instruction is optional in some procedures, it is good programming practice to use it at the end of every procedure.

2.5 Assignments

An assignment tells the system that a string should be put in a special place in system memory. In the example:

```
Work = "Building 021"
```

The string Building 021 is stored as the value Work in system memory. Because Work can have different values (be reassigned to mean different things) in different parts of the procedure, it is called a Variable.

2.6 Labels

A label is any word that is followed by a colon (with no space between the word and the colon) and is not in quotation marks. For example:

```
MYNAME:
```

A label marks the start of a subroutine. The following example shows one use of a label (called error) within a procedure:

```
   .
   .
   .
IF problem  = 'yes' then SIGNAL error
   .
   .
   .
error:
   SAY 'Problem in your data'
   EXIT
```

3 Working with Variables and Arithmetic

3.1 Introduction

In this section, you will see how to use variables and arithmetic and how to add comments throughout a procedure to describe how it works. Some of the topics in this section include the following:

Variable	A piece of data given a unique name
Value	Contents of a variable
Operators	Symbols used for arithmetic functions
Addition	+ operator
Subtraction	- operator
Multiplication	* operator
Division	/, //, % operators

3.2 Variables

A variable is a piece of data with a varying value. Within a procedure, each variable is known by a unique name and is always referred to by that name. When you choose a name for a variable, the first character must be one of:

```
A B C...Z ! ? _
```

Lowercase letters are also allowed as a first letter. The interpreter changes them to uppercase. The rest of the characters can be any of the preceding characters and also 0 through 9.

3.3 Value

The value of a variable can change, but the name cannot. When you name a variable (give it a value), it is an assignment. For example, any statement of the form,

```
symbol = expression
```

is an assignment statement. You are telling the interpreter to compute what the expression is and put the result into a variable called a symbol. It is the same as saying, "Let symbol be made equal to the result of expression" or every time symbol appears in the text of a SAY string unprotected by single or double quotation marks, display expression in its place. The relationship between a variable and a value is like that between a post-office box and its contents; The box number does not change, but the contents of the box may be changed at any time. Another example of an assignment is:

```
num1 = 10
```

The num1 assignment has the same meaning as the word symbol in the previous example, and the value 10 has the same meaning as the word expression.

One way to give the variable num1 a new value is by adding to the old value in the assignment:

```
num1 = num1 + 3
```

The value of num1 has now been changed from 10 to 13.

A special concept in REXX is that any variable that is not assigned a value assumes the uppercase version of the variable as its initial value. For example, if you write in a procedure,

```
list = 2 20 40
SAY list
```

you see this on your screen:

```
2 20 40
```

As you can see, list receives the values it is assigned. But if you do not assign any value to list and only write, SAY list you see this on your screen:

```
LIST
```

Here is a simple procedure called VARIABLE.CMD that assigns values to variables:

```
/* Assigning a value to a variable */
a = 'abc'
SAY a
b = 'def'
SAY a b
EXIT
```

When you run the VARIABLE procedure, it looks like this on your screen:

```
abc
abc def
```

Assigning values is easy, but you have to make sure a variable is not used unintentionally, as in this example named MEETING.CMD:

```
/* Unintentional interpretation of a variable */
the='no'
SAY Here is the person I want to meet
EXIT
When the procedure is run, it looks like this:
Running the porgram
HERE IS no PERSON I WANT TO MEET
```

To avoid unintentionally substituting a variable for the word, put the sentence in quotation marks as shown in this example of MEETING.CMD, which assigns a variable correctly:

```
/* Correct interpretation of a variable the*/
the= 'no'
SAY "Here is the person I want to meet"
EXIT
```

3.4 Working with Arithmetic

Your REXX procedures may need to include arithmetic operations of addition, subtraction, multiplication, and division. For example, you may want to assign a numeric value to two variables and then add the variables.

Arithmetic operations are performed the usual way. You can use whole numbers and decimal fractions. A whole number is an integer, or any number that is a natural number, either positive, negative, or zero, that does not contain a decimal part (for example, 1, 25, or 50). A decimal fraction contains a decimal point (for example, 1.45 or 0.6).

Before you see how these four operations are handled in a procedure, here is an explanation of what the operations look like and the symbols used. These are just a few of the arithmetic operations used in REXX.

Note: The examples contain a blank space between numbers and operators so that you can see the equations better, but the blank is optional.

Operators

The symbols used for arithmetic (+ , -, *, /) are called operators because they operate on the adjacent terms. In the following example, the operators act on the numbers (terms) 4 and 2:

```
SAY 4 + 2  /* says "6" */
SAY 4 * 2  /* says "8" */
SAY 4 / 2  /* says "2" */
```

Addition

The operator for addition is the plus sign (+). An instruction to add two numbers is:

```
SAY 4 + 2
```

The answer you see on your screen is 6.

Subtraction

The operator for subtraction is the minus sign (-). An instruction to subtract two numbers is:

```
SAY 8 - 3
```

The answer on your screen is 5.

Multiplication

The operator for multiplication is the asterisk (*). An instruction to multiply two numbers is:

```
SAY 2 * 2
```

The answer on your screen is 4.

Division

For division, there are several operators you can use, depending on whether or not you want the answer expressed as a whole number. For example, for a simple division, the symbol is one slash (/). An instruction to divide is:

```
SAY 7 / 2
```

The answer on your screen is 3.5.

To divide and return just a remainder, the operator is two slashes (//). To divide, and return only the whole number portion of an answer and no remainder, the operator is the percent sign (%).

For examples showing you how to perform four arithmetic operations on variables, select the Examples push button.

Evaluating Expressions

Expressions are normally evaluated from left to right. An equation helps to illustrate this point. Until now, you have seen equations with only one operator and two terms, such as 4 + 2. Suppose you had this equation:

```
9 - 5 + 4 =
```

The 9 - 5 would be computed first. The answer, 4, would be added to 4 for a final value: 8.

Some operations are given priority over others. In general, the rules of algebra apply to equations. In this equation, the division is handled before the addition:

```
10 + 8 / 2 =
```

The value is 14.

If you use parentheses in an equation, the interpreter evaluates what is in the parentheses first. For example:

```
(10 + 8) / 2 =
```

The value is 9.

4 REXX Features

4.1 Introduction to REXX Features

Some features of REXX that you can use to write more intricate procedures will be discussed in this section. You will see how to have a procedure make decisions by testing a value with the IF instruction. You will also see how to compare values and determine if an expression is true or false. A brief description of the terms covered in this section follows below:

IF	Used with THEN. Checks if the expression is true. Makes a decision about a single instruction.
THEN	Identifies the instruction to be run if the expression is true.
ELSE	Identifies the instruction to be run if the expression is false.
SELECT	Tells the interpreter to select one of a number of instructions.
WHEN	Used with SELECT. Identifies an expression to be tested.
OTHERWISE	Used with SELECT. Indicates the instruction to be run if expressions tested are false.
DO-END	Indicates that a group of instructions should be run.
NOP	Indicates that nothing is to happen for one expression.
Comparisons > < =	Indicates greater than, less than, equal to.
**NOT Operator ¬ or **	Changes the value of a term from true to false, or from false to true.
AND Operator &	Gives the value of true if both terms are true.
OR Operator |	Gives the value of true unless both terms are false.

4.2 Making Decisions (IF THEN)

In the procedures discussed in earlier sections, instructions were run sequentially. In this section, you will see how you can control the order in which instructions are run. Depending upon the user's interaction with your procedure, you may choose not to run some of your lines of code.

Two instructions that let you make decisions in your procedures are the IF and SELECT instructions. The IF instruction lets you control whether the next instruction is run or skipped. The SELECT instruction lets you choose one instruction to run from a group of instructions.

The IF instruction is used with a THEN instruction to make a decision. The interpreter runs the instruction if the expression is true; for example:

```
IF answer = "YES"
 THEN
 SAY "OK!"
```

In the previous example, the SAY instruction is run only if answer has the value of YES.

Grouping Instructions Using DO and END

To tell the interpreter to run a list of instructions after the THEN instruction, use:

```
DO
   Instruction1
   Instruction2
   Instruction3
END
```

The DO instruction and its END instruction tell the interpreter to treat any instructions between them as a single instruction.

4.3 The ELSE Instruction

ELSE identifies the instruction to be run if the expression is false. To tell the interpreter to select from one of two possible instructions, use:

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

You could include the IF-THEN-ELSE format in a procedure like this:

```
IF answer = 'YES'
 THEN SAY 'OK!'
ELSE SAY 'why not?'
```

Try the next example, GOING.CMD, to see how choosing between two instructions works.

```
/* Using IF-THEN-ELSE */
SAY "Are you going to the meeting?"
PULL answer
IF answer = "YES"
 THEN
 SAY "I'll look for you."
ELSE
 SAY "I'll take notes for you."
EXIT
```

When this procedure is run, this is what you will see on your screen:

```
Are you going to the meeting?

yes

I'll look for you.
```

4.4 SELECT, END, WHEN, OTHERWISE, and NOP Instructions

SELECT tells the interpreter to select one of a number of instructions. It is used only with WHEN, THEN, END, and sometimes, OTHERWISE. The END instruction marks the end of every SELECT group. The SELECT instruction looks like this:

```
SELECT
   WHEN expression1
      THEN instruction1
   WHEN expression2
      THEN instruction2
   WHEN expression3
      THEN instruction3
...
OTHERWISE
   instruction
   instruction
   instruction
END
```

Note: An IF-THEN instruction cannot be used with a SELECT instruction unless it follows a WHEN or OTHERWISE instruction. You can read this format as follows:

- If expression1 is true, instruction1 is run. After this, processing continues with the instruction following the END. The END instruction signals the end of the SELECT instruction.
- If expression1 is false, expression2 is tested. Then, if expression2 is true, instruction2 is run and processing continues with the instruction following the END.
- If, and only if, all of the specified expressions are false, then processing continues with the instruction following OTHERWISE.

A DO-END instruction could be included inside a SELECT instruction as follows:

```
SELECT
   WHEN expression1 THEN
      DO
        instruction1
        instruction2
        instruction3
     END
.
.
.
```

You can use the SELECT instruction when you are looking at one variable that can have several different values associated with it. With each different value, you can set a different condition.

For example, suppose you wanted a reminder of weekday activities. For the variable day, you can have a value of Monday through Friday. Depending on the day of the week (the value of the variable), you can list a different activity (instruction). You could use a procedure such as the following, SELECT.CMD, which chooses from several instructions.

Note: A THEN or ELSE instruction must be followed by an instruction.

```
/* Selecting weekday activities */
SAY 'What day is it today?'
Pull day
SELECT
   WHEN day = 'MONDAY'
      THEN
      SAY 'Model A board meeting'
   WHEN day = 'TUESDAY'
      THEN
      SAY "My Team Meeting"
   WHEN day = 'WEDNESDAY'
      THEN NOP   /* Nothing happens here */
```

```
   WHEN day = 'THURSDAY'
      THEN
      SAY "My Seminar"
   WHEN day = 'FRIDAY'
      THEN
      SAY "My Book Review"
OTHERWISE
     SAY  "It is the weekend, anything can happen!"
END
EXIT
```

NOP Instruction: If you want nothing to happen for one expression, use the NOP (No Operation) instruction, as shown in the previous example for Wednesday.

4.5 True and False Operators

Determining if an expression is true or false is useful in your procedures. If an expression is true, the computed result is 1. If an expression is false, the computed result is 0. The following shows some ways to check for true or false operators.

Comparisons - Some operators you can use for comparisons are:

>	Greater than
<	Less than
=	Equal to

Comparisons can be made with numbers or can be character-to-character. Some numeric comparisons are:

The value of 5 > 3 is 1	This result is true.
The value of 2.0 = 002 is 1	This result is true.
The value of 332 < 299 is 0	This result is false.

If the terms being compared are not numbers, the interpreter compares characters. For example, the two words (strings) airmail and airplane when compared character for character have the first three letters the same. Since m < p, airmail < airplane.

Equal - An equal sign (=) can have two meanings in REXX, depending on its position. For example,

```
amount = 5          /* This is an assignment */
```

gives the variable amount, the value of 5. If an equal sign is in a statement other than as an assignment, it means the statement is a comparison. For example,

```
SAY amount = 5      /* This is a comparison  */
```

compares the value of amount with 5. If they are the same, a 1 is displayed, otherwise, a 0 is displayed.

4.6 The Logical Operators, NOT, AND, OR

Logical operators can return only the values of 1 or 0. The NOT operator (¬ or \) in front of a term reverses its value either from true to false or from false to true.

```
SAY  \ 0         /* gives '1'         */
SAY  \ 1         /* gives '0'         */
SAY  \ (4 = 4)   /* gives '0'         */
SAY  \ 2         /* gives  a  syntax error      */
```

The AND operator (&) between two terms gives a value of true only if both terms are true.

```
SAY ( 3 = 3 ) & ( 5 = 5 )   /* gives '1'       */
SAY ( 3 = 4 ) & ( 5 = 5 )   /* gives '0'       */
SAY ( 3 = 3 ) & ( 4 = 5 )   /* gives '0'       */
SAY ( 3 = 4 ) & ( 4 = 5 )   /* gives '0'       */
```

The OR operator (|) between two terms gives a value of true unless both terms are false.

Note: Depending upon your keyboard and the code page you are using, you may not have the solid vertical bar to select. For this reason, REXX also recognizes the use of the split vertical bar as a logical OR symbol. Some keyboards may have both characters. If so, they are not interchangeable; only the character that is equal to the ASCII value of 124 works as the logical OR. This type of mismatch can also cause the character on your screen to be different from the character on your keyboard.

```
SAY ( 3 = 3 ) | ( 5 = 5 )   /* gives '1'       */
SAY ( 3 = 4 ) | ( 5 = 5 )   /* gives '1'       */
SAY ( 3 = 3 ) | ( 4 = 5 )   /* gives '1'       */
SAY ( 3 = 4 ) | ( 4 = 5 )   /* gives '0'       */
```

5 Automating Repetitive Tasks - Using Loops

5.1 Introduction to Loops

If you want to repeat several instructions in a procedure, you can use a loop. Loops often are used in programming because they condense many lines of instructions into a group that can be run more than once. Loops make your procedures more concise, and with a loop, you can continue asking a user for input until the correct answer is given.

With loops, you can keep adding or subtracting numbers until you want to stop. You can define how many times you want a procedure to handle an operation. You will see how to use simple loops to repeat instructions in a procedure.

The two types of loops you may find useful are repetitive loops and conditional loops. Loops begin with a DO instruction and end with the END instruction. The following is a list of topics in this section:

DO num loop	Repeats the loop a fixed number of times.
DO i=1 to 10 loop	Numbers each pass through the loop. Sets a starting and ending value for the variable.
DO WHILE	Tests for true or false at the top of the loop. Repeats the loop if true. If false, continues processing after END.
Do UNTIL	Tests for true or false at the bottom of the loop. Repeats the loop if false. If true, continues processing after END.
LEAVE	Causes the interpreter to exit a loop.
DO FOREVER	Repeats instructions until the user says to quit.
Getting out of loops	Requires that you press the Ctrl+Break keys.
Parsing words	Assigns a different variable to each word in a group.

5.2 Repetitive Loops

Simple repetitive loops can be run a number of times. You can specify the number of repetitions for the loop, or you can use a variable that has a changing value.

The following shows how to repeat a loop a fixed number of times.

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

The num is a whole number, which is the number of times the loop is to be run.

Here is LOOP.CMD, an example of a simple repetitive loop.

```
/* A simple loop */
DO 5
   SAY 'Thank-you'
END
EXIT
```

When you run the LOOP.CMD, you see this on your screen:

```
Thank-you
Thank-you
Thank-you
Thank-you
Thank-you
```

Another type of DO instruction is:

```
DO XYZ = 1 to 10
```

This type of DO instruction numbers each pass through the loop so you can use it as a variable. The value of XYZ changes (by 1) each time you pass through the loop. The 1 (or some number) gives the value you want the variable to have the first time through the loop. The 10 (or some number) gives the value you want the variable to have the last time through the loop.

NEWLOOP.CMD is an example of another loop:

```
/* Another loop */
sum = 0
DO XYZ = 1 to 7
   SAY 'Enter value' XYZ
   PULL value
   sum = sum + value
END
SAY 'The total is' sum
EXIT
```

Here are the results of the NEWLOOP.CMD procedure:

```
Enter value 1
2
Enter value 2
4
Enter value 3
6
Enter value 4
8
Enter value 5
10
Enter value 6
12
Enter value 7
14
The total is 56
```

When a loop ends, the procedure continues with the instruction following the end of the loop, which is identified by END.

5.3 Conditional Loops

Conditional loops are run when a true or false condition is met. We will now look at some instructions used for conditional loops:

DO WHILE and DO UNTIL

The DO WHILE and DO UNTIL instructions are run while or until some condition is met.

A DO WHILE loop is:

```
DO WHILE expression
   instruction1
   instruction2
   instruction3
END
```

The DO WHILE instruction tests for a true or false condition at the top of the loop; that is, before processing the instructions that follow. If the expression is true, the instructions are performed. If the expression is false, the loop ends and moves to the instruction following END.

A DO UNTIL instruction differs from the DO WHILE because it processes the body of instructions first, then evaluates the expression. If the expression is false, the instructions are repeated (a loop). If the expression is true, the procedure ends or moves to the next step outside the loop.

The DO UNTIL instruction tests at the bottom of the loop; therefore, the instructions within the DO loop are run at least once.

An example of a DO UNTIL loop follows:

```
DO UNTIL expression
   instruction1
   instruction2
   instruction3
END
```

LEAVE

You may want to end a loop before the ending conditions are met. You can accomplish this with the LEAVE instruction. This instruction ends the loop and continues processing with the instruction following END. The following procedure, LEAVE.CMD, causes the interpreter to end the loop.

```
/* Using the LEAVE instruction in a loop */
SAY 'enter the amount of money available'
PULL salary
spent = 0        /* Sets spent to a value of 0 */
DO UNTIL spent > salary
   SAY 'Type in cost of item or END to quit'
   PULL cost
      IF cost = 'END'
      THEN
      LEAVE
   spent = spent + cost
END
SAY 'Empty pockets.'
EXIT
```

DO FOREVER

There may be situations when you do not know how many times to repeat a loop. For example, you may want the user to type specific numeric data (numbers to add together), and have the loop perform the calculation until the user says to stop. For such a procedure, you can use the DO FOREVER instruction with the LEAVE instruction.

The following shows the simple use of a DO FOREVER ending when the user stops.

```
/* Using a DO FOREVER loop to add numbers */
sum = 0
DO FOREVER
   SAY 'Enter number or END to quit'
   PULL value
   IF value = 'END'
      THEN
      LEAVE  /* procedure quits when the user enters "end" */
   sum = sum + value
END
SAY 'The sum is ' sum
EXIT
```

5.4 Parsing Words

The PULL instruction collects a response and puts it into system memory as a variable. PULL also can be used to put each word from a group of words into a different variable. In REXX, this is called parsing. The variable names used in the next example are: first, second, third, and rest.

```
SAY 'Please enter three or more words'
PULL first second third rest
```

Suppose you enter this as your response:

```
garbage in garbage out
```

When you press the Enter key, the procedure continues.

However, the variables are assigned as follows:

The variable first is given the value GARBAGE.
The variable second is given the value IN.
The variable third is given the value GARBAGE.
The variable rest is given the value OUT.

In general, each variable receives a word, without blanks, and the last variable receives the rest of the input, if any, with blanks. If there are more variables than words, the extra variables are assigned the null, or empty, value.

6 Advanced REXX Functions

6.1 Introduction to Advanced REXX Functions

As you become more skilled at programming, you may want to create procedures that do more and run more efficiently. Sometimes this means adding a special function to a procedure or calling a subroutine. In this section, we shall see how these functions can help to build a better foundation in REXX.

Functions	Perform a computation and return a result.
DATATYPE()	An internal function that verifies that the data is a specific type.
SUBSTR()	An internal function that selects part of a string.
CALL	Causes the procedure to look for a subroutine label and begin running the instructions following the label.
REXX.CMD File Commands	Treat commands as expressions.
Error Messages	Tell you if the command runs correctly. If the procedure runs correctly, no message is displayed.

6.2 Functions

In REXX, a function call can be written anywhere in an expression. The function performs the requested computation and returns a result. REXX then uses the result in the expression in place of the function call.

Think of a function as: You are trying to find someone's telephone number. You call the telephone operator and ask for the number. After receiving the number, you call the person. The steps you have completed in locating and calling the person could be labeled a function.

Generally, if the interpreter finds this in an expression,

```
name(expression)
```

it assumes that name is the name of a function being called. There is no space between the end of the name and the left parenthesis. If you leave out the right parenthesis, it is an error.

The expressions inside the parentheses are the arguments. An argument can itself be an expression; the interpreter computes the value of this argument before passing it to the function. If a function requires more than one argument, use commas to separate each argument.

Built-in Functions

REXX has more than 50 built-in functions. A dictionary of built-in functions is in this documentation.

MAX is a built-in function that you can use to obtain the greatest number of a set of numbers:

```
MAX(number, number, ...)
```

For example:

```
MAX(2,4,8,6) = 8
MAX(2,4+5,6) = 9
```

Note that in the second example, the 4+5 is an expression. A function call, like any other expression, usually is contained in a clause as part of an assignment or instruction.

6.3 DATATYPE()

When attempting to perform arithmetic on data entered from the keyboard, you can use the DATATYPE() function to check that the data is valid.

This function has several forms. The simplest form returns the word, NUM, if the expression inside the parentheses () is accepted by the interpreter as a number that can be used in the arithmetic operation. Otherwise, it returns the word, CHAR. For example:

```
The value of DATATYPE(56) is NUM
The value of DATATYPE(6.2) is NUM
The value of DATATYPE('$5.50') is CHAR
```

In the following procedure, DATATYPE.CMD, the internal REXX function, DATATYPE(), is used and the user is asked to keep typing a valid number until a correct one is typed.

```
/* Using the DATATYPE() Function */
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
EXIT
```

If you want the user to type only whole numbers, you could use another form of the DATATYPE() function:

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

The arguments for this form are:

- number - refers to the data to be tested.
- whole - refers to the type of data to be tested. In this example, the data must be a whole number.

This form returns a 1 if number is a whole number, or a 0 otherwise.

6.4 SUBSTR()

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

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

For example:

```
S = 'reveal'
SAY substr(S,2,3) /* Says 'eve'. Beginning with the second  */
                  /*  character, takes three characters.    */
SAY substr(S,3,4) /* Says 'veal'. Beginning with the third  */
                  /*  character, takes four characters.     */
```

6.5 CALL

The CALL instruction causes the interpreter to search your procedure until a label is found that marks the start of the subroutine. Remember, a label (word) is a symbol followed by a colon (:). Processing continues from there until the interpreter finds a RETURN or an EXIT instruction.

A subroutine can be called from more than one place in a procedure. When the subroutine is finished, the interpreter always returns to the instruction following the CALL instruction from which it came.

Often each CALL instruction supplies data (called arguments or expressions) that the subroutine is to use. In the subroutine, you can find out what data has been supplied by using the ARG instruction.

The CALL instruction is written in the following form:

CALL name Argument1, Argument2 ...

For the name, the interpreter looks for the corresponding label (name) in your procedure. If no label is found, the interpreter looks for a built-in function or a .CMD file with that name.

The arguments are expressions. You can have up to 20 arguments in a CALL instruction. An example of a procedure that calls a subroutine follows. Note that the EXIT instruction causes a return to the operating system. The EXIT instruction stops the main procedure from continuing into the subroutine.

In the following example, REXX.CMD, the procedure calls a subroutine from a main procedure.

```
/* Calling a subroutine from a procedure */
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          /* This ends the main procedure. */
/**/
/* Subroutine starts here to repeat REXX three times. */
/* The first argument is displayed on screen three     */
/* times, with punctuation.  */
/**/
TRIPLE:
SAY ARG(1)"    "ARG(1)"   "ARG(1)"!"
RETURN/* This ends the subroutine.         */
```

When REXX.CMD is run on your system, the following is

displayed:

```
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!
```

6.6 REXX.CMD File Commands

In a REXX procedure, anything not recognized as an instruction, assignment, or label is considered a command. The statement recognized as a command is treated as an expression. The expression is evaluated first, then the result is passed to the operating system.

The following example, COPYLIST.CMD, shows how a command is treated as an expression. Note how the special character (*) is put in quotation marks. COPYLIST.CMD copies files from drive A to drive B.

```
/* Issuing a command from a procedure. This example copies        */
/* all files that have an extension of.LST from                   */
/* drive A to drive B.                                            */
SAY
COPY "a:*.lst b:" /* This statement is treated as                 */
                  /* an expression.                               */
                  /* The result is passed to the operating system */
EXIT
```

Note: In the preceding example, the whole command except for COPY is in quotation marks for the following reasons:

- If the colon (:) were not in quotation marks, the REXX interpreter would treat a: and b: as labels.
- If the asterisk (*) were not in quotation marks, the REXX interpreter would attempt to multiply the value of a: by .LST.
- It is also acceptable to include the entire command in quotation marks so that "COPY a:*.LST b:" is displayed.

6.7 Error Messages

There are two basic reasons errors occur when REXX is processing a procedure.

One reason is because of the way the procedure is written; for example, unmatched quotation marks or commas in the wrong place. Maybe an IF instruction was entered without a matching THEN. When such an error occurs, a REXX error message is issued.

A second reason for an error to occur is because of a command that the REXX procedure has issued. For example, a COPY command can fail because the user's disk is full or a file cannot be found. In this case, a regular operating system error message is issued. When you write commands in your procedures, consider what might happen if the command fails to run correctly.

When a command is issued from a REXX procedure, the command interpreter gets a return code and stores it in the REXX special variable, RC (return code). When you write a procedure, you can test for these variables to see what happens when the command is issued.

Here is how you discover a failure. When commands finish running, they always provide a return code. A return code of 0 nearly always means that all is well. Any other number usually means that something is wrong.

In the following example, ADD.CMD, there is an error in line 6; the plus sign (+) has been typed incorrectly as an ampersand (&).

```
/* This procedure adds two numbers */
SAY "Enter the first number."
PULL num1
SAY "Enter the second number."
PULL num2
SAY "The sum of the two numbers is" num1 & num2
EXIT
```

When the above procedure, ADD.CMD, is run, the following error message is displayed.

```
  6+++ SAY "The sum of the two numbers is" num1 & num2
REX0034: Error 34 running C:\REXX\ADD.CMD,line 6:logical value not 0 or 1.
```

Any command that is valid at the command prompt is valid in a REXX procedure. The command interpreter treats the command statement the same way as any other expression, substituting the values of variables, and so on. (The rules are the same as for commands entered at the command prompt.)

Return Codes: When the command interpreter has issued a command and the operating system has finished running it, he command interpreter gets the return code and stores it in the REXX special variable RC (return code). In your procedure, you should test this variable to see what happens when the command is run.

The following example shows a few lines from a procedure where the return code is tested:

```
* Testing the Return Code in a Procedure. */
COPY a:*.lst b:'
IF rc = 0   /* RC contains the return code from the COPY
command */
THEN
 SAY 'All *lst files copied'
ELSE
 SAY 'Error occurred copying files'
```

7 Keyword Instructions

7.1 Introduction to Keyword Instructions

A keyword instruction is one or more clauses, the first of which starts with a keyword that identifies the instruction. If the instruction has more than one clause, the clauses are separated by a delimiter, in this case a semicolon (;).

Some keyword instructions, such as those starting with the keyword DO, can include nested instructions. In the syntax diagrams, symbols (words) in upper case letters denote keywords; other words (such as expression) denote a collection of symbols. Keywords are not case dependent: the symbols if, If, and iF would all invoke the instruction IF. Also, you usually omit most of the clause delimiters (;) shown because they are implied by the end of a line.

7.2 ADDRESS

ADDRESS [environment [expression]]

or

ADDRESS [VALUE] expression1

Address is used to send a single command to a specified environment, code an environment, a literal string, or a single symbol, which is taken to be a constant, followed by an expression. The expression is evaluated, and the resulting command string is routed to environment. After the command is executed, environment is set back to whatever it was before, thus temporarily changing the destination for a single command.

Example:

```
ADDRESS CMD "DIR C:\STARTUP.CMD"   /*   OS/2   */
```

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

Example:

Suppose that the environment for a text editor is registered by the name EDIT:

```
address CMD
'DIR C:\STARTUP.CMD'
if rc=0 then 'COPY STARTUP.CMD *.TMP'
address EDIT
```

Subsequent commands are passed to the editor until the next ADDRESS instruction.

Similarly, you can use the VALUE form to make a lasting change to the environment. Here expression1 (which may be just a variable name) is evaluated, and the result forms the name of the environment. The subkeyword VALUE can be omitted as long as expression1 starts with a special character (so that it cannot be mistaken for a symbol or string).

Example:

```
ADDRESS('ENVIR'||number)
```

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. Repeated execution of ADDRESS alone, therefore, switches the command

destination between two environments alternately. A null string for the environment name (" ") is the same as the default environment.

7.3 ARG

ARG [template]

ARG is used to retrieve the argument strings provided to a program or internal routine and assign them to variables.

It is a short form of the following instruction:

PARSE UPPER ARG [template]

Template is a list of symbols separated by blanks or patterns.

Unless a subroutine or internal function is being run, the interpreter reads the arguments given on the program invocation, translates them to uppercase (for example, a lowercase a-z to an uppercase A-Z), and then parses them into variables. Use the PARSE ARG instruction if you do not want uppercase translation.

If a subroutine or internal function is being run, the data used will be the argument strings passed to the routine.

The ARG (and PARSE ARG) instructions can be run as often as desired (typically with different templates) and always parse the same current input strings. There are no restrictions on the length or content of the data parsed except those imposed by the caller.

Example:

```
/* String passed is "Easy Rider"  */
Arg adjective noun
/* Now:  "ADJECTIVE"  contains 'EASY'          */
/*       "NOUN"       contains 'RIDER'         */
```

If more than one string is expected to be available to the program or routine, each can be selected in turn by using a comma in the parsing template.

Example:

```
/* function is invoked by  FRED('data X',1,5)   */
Fred:  Arg string, num1, num2
/* Now:  "STRING" contains 'DATA X'            */
/*       "NUM1"   contains '1'                 */
/*       "NUM2"   contains '5'                 */
```

Notes: The argument strings to a REXX program or internal routine can also be retrieved or checked by using the ARG built-in function.

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

7.4 CALL

CALL name [[expression1] [,] [expression2]…[,] [expression20]]

or

CALL [OFF [option]]

or

CALL [ON [option] [NAME trapname]]

CALL is used to invoke a routine (if you specify name) or to control the trapping of certain conditions (if ON or OFF is specified).

To control trapping, specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap.

To invoke a routine, specify name, which is a symbol or literal string that is taken as a constant. The name must be a valid symbol. The routine invoked can be any of the following:

- An internal routine
- An external routine
- A built-in function

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

The routine can optionally return a result and is functionally identical to the clause:

result=name([[expression1] [,] [expression2]…[,] [expression20]])

The exception is that the variable result becomes uninitialized if no result is returned by the routine invoked.

The expressions are evaluated in order from left to right, and form the argument strings during execution of the routine.

Any ARG or PARSE ARG instructions or ARG built-in function in the called routine will access these strings rather than those previously active in the calling program. You can omit expressions, if appropriate, by including extra commas.

The CALL then causes a branch to the routine called name using the same mechanism as function calls. The order in which these are searched for is described in the section on functions. A brief summary follows:

Internal routines

Built-in routines

External routines

7.5 DO

DO [repetitor] [conditional];[instruction1] [instruction2]…[instruction20] END [name]

repetitor: **name=expri [TO exprt] [BY exprt] [FOR exprf]**
or: **FOREVER**
or: **exprr**

conditional: **WHILE-exprw**
or: **UNTIL-expru**

DO is used to group instructions together and optionally to execute them repetitively. During repetitive execution, a control variable (name) can be stepped through some range of values.

Syntax Notes:

- The exprr, expri, exprb, exprt, and exprf options (if any are present) are any expressions that evaluate to a number. The exprr and exprf options are further restricted to result in a nonnegative whole number. If necessary, the numbers will be rounded according to the setting of NUMERIC DIGITS.
- The exprw or expru options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used.
- The instructions can include assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
- The subkeywords TO, BY, FOR, WHILE, and UNTIL are reserved within a DO instruction, in that they cannot name variables in the expressions but they can be used as the name of the control variable. FOREVER is similarly reserved, but only if it immediately follows the keyword DO.
- The exprb option defaults to 1, if relevant.

Simple DO Group

If neither repetitor nor conditional is given, the construct merely groups a number of instructions together. These are executed once. Otherwise, the group of instructions is a repetitive DO loop, and they are executed according to the repetitor phrase, optionally modified by the conditional phrase.

In the following example, the instructions are executed once.

Example:

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

Simple Repetitive Loops

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

In the simple form of a repetitive loop, exprr is evaluated immediately (and must result in a nonnegative whole number), and the loop is then executed that many times.

Example:

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

Note that, similar to the distinction between a command and an assignment, if the first character of exprr is a symbol and the second is an "=" character, the controlled form of repetitor is expected.

Controlled Repetitive Loops

The controlled form specifies a control variable, name, which is assigned an initial value (the result of expri, formatted as though 0 had been added). The variable is then stepped

(that is, the result of exprb is added at the bottom of the loop) each time the group of instructions is run. The group is run repeatedly while the end condition (determined by the result of exprt) is not met. If exprb is positive or zero, the loop will be terminated when name is greater than exprt. If negative, the loop is terminated when name is less than exprt.

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

Example:

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

The numbers do not have to be whole numbers.

Example:

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

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

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

The processing of a controlled loop can be bounded further by a FOR phrase. In this case, exprf must be given and must evaluate to a nonnegative whole number. This acts just like the repetition count in a simple repetitive loop, and sets a limit to the number of iterations around the loop if no other condition terminates it. Similar to the TO and BY expressions, it is evaluated once only-when the DO instruction is first executed and before the control variable is given its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

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

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

Example:

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

Note: The values taken by the control variable may be affected by the NUMERIC settings, since normal REXX arithmetic rules apply to the computation of stepping the control variable.

Conditional Phrases (WHILE and UNTIL)

Any of the forms of repetitor (none, FOREVER, simple, or controlled) can be followed by a conditional phrase, which may cause termination of the loop. If you specify WHILE or UNTIL, exprw or expru, respectively, is evaluated each time around the loop using the latest values of all variables (and must evaluate to either 0 or 1). The group of instructions is repeatedly processed either while the result is 1 or until the result is 1.

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

Example:

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

Note: The processing of repetitive loops can also be modified by using the LEAVE or ITERATE instructions.

7.6 DROP

DROP name1 name 2…name20

Each name is a valid variable symbol, optionally enclosed in parentheses (to denote a subsidiary list), and separated from any other name by one or more blanks or comments.

DROP is used to unassign variables; that is, to restore them to their original uninitialized state.

Each variable specified is dropped from the list of known variables. If a single name is enclosed in parentheses, then its value is used as a subsidiary list of variables to drop. This stored list must follow the same rules as for the main list (that is, valid variable names, separated by blanks) but with no parentheses and no leading or trailing blanks. The variables are dropped in sequence from left to right. It is not an error to specify a name more than once, or to DROP a variable that is not known. If an exposed variable is named (see the PROCEDURE instruction), the variable itself in the older generation is dropped.

Example:

```
j=4
Drop  a x.3 x.j
/* would reset the variables: A, X.3, and X.4 */
/* so that reference to them returns their name.   */
```

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

Example:

```
x=4;y=5;z=6;
a='x y z'
DROP (a)    /* will drop x,y, and z */
```

If a stem is specified (that is, a symbol that contains only one period, as the last character), all variables starting with that stem are dropped.

Example:

```
Drop  x.
/* would reset all variables with names starting with X. */
```

7.7 EXIT

EXIT [expression]

EXIT is used to leave a program unconditionally. Optionally, EXIT returns a data string to the caller. The program is terminated immediately, even if an internal routine is currently being run. If no internal routine is active, RETURN and EXIT are identical in their effect on the program that is being run.

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

Example:

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

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

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

Note: The language processor does not distinguish between invocation as a command on the one hand, and invocation as a subroutine or function on the other. If the program was invoked through a command interface, an attempt is made to convert the returned value to a return code acceptable by the REXX caller. The returned string must be a whole number whose value will fit in a 16-bit signed integer (within the range -2**15 to 2**15-1).

7.8 IF

IF-expression [;] THEN [;] instruction [ELSE [;] instruction]

IF is used to conditionally process an instruction or group of instructions depending on the evaluation of the expression. The expression must evaluate to 0 or 1.

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

Example:

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

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

Example:

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

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

Example:

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

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

The symbol THEN cannot be used within expression, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be terminated by the THEN, without a semicolon being required. If this were not so, users of other computer languages would experience considerable difficulties.

7.9 INTERPRET

INTERPRET expression

INTERPRET is used to process instructions that have been built dynamically by evaluating expression.

The expression is evaluated, and is then processed (interpreted) as though the resulting string was a line inserted into the input file (and bracketed by a DO; and an END;).

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

A semicolon is implied at the end of the expression during processing, as a service to the user.

Example:

```
data='FRED'
interpret data '= 4'
* Will a) build the string  "FRED = 4"         */
*      b) execute  FRED = 4;     */
/* Thus the variable "FRED" will be set to "4"  */
```

Example:

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

Notes: Labels within the interpreted string are not permanent and are therefore ignored. Therefore, executing a SIGNAL instruction from within an interpreted string causes immediate exit from that string before the label search begins.

If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you might find that executing the instruction with TRACE R or TRACE I set is helpful.

Example:

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

when run, the following trace is displayed:

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

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

```
Hello Kitty!
```

Note: For many purposes, the VALUE function can be used instead of the INTERPRET instruction. Line 5 in the last example could therefore have been replaced by:

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

INTERPRET is usually only required in special cases, such as when more than one statement is to be interpreted at once.

7.10 ITERATE

ITERATE [name]

ITERATE is used to alter the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO loop).

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

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

Example:

```
do i=1 to 4
  if i=2 then iterate
  say i
  end
/* Would display the numbers:   1, 3, 4  */
```

Notes: If specified, name must match the name on the DO instruction in all respects except case. No substitution for compound variables is carried out when the comparison is made.

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

If more than one active loop uses the same control variable, the innermost loop is the one selected by ITERATE.

7.11 LEAVE

LEAVE [name]

LEAVE is used to cause an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO loop).

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

If name is not specified, LEAVE terminates the innermost active repetitive loop. If name is specified, it must be the name of the control variable of a currently active loop which may be the innermost loop; that loop (and any active loops inside it) is then terminated. Control then passes to the clause following the END clause that matches the DO clause of the selected loop.

Example:

```
do i=1 to 5
  say i
  if i=3 then leave
  end
/* Would display the numbers:   1, 2, 3  */
```

Notes: If specified, name must match the one on the DO instruction in all respects except case. No substitution for compound variables is carried out when the comparison is made.

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

If more than one active loop uses the same control variable, the innermost one is the one selected by LEAVE.

7.12 NOP

NOP

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

Example:

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

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

7.13 NUMERIC

NUMERIC DIGITS [expression]

or

NUMERIC FORM [option [[value] expression]]

or

NUMERIC FUZZ [expression]

The NUMERIC instruction is used to change the way in which arithmetic operations are carried out.

NUMERIC DIGITS controls the precision to which arithmetic operations and arithmetic built-in functions are evaluated. If expression is omitted, then the default value of 9 is used. Otherwise the result of the expression is rounded, if necessary, according to the current setting of NUMERIC DIGITS. The value used must be a positive whole number that is larger than the current NUMERIC FUZZ setting.

There is no limit to the value for DIGITS (except the amount of storage available), but note that high precisions are likely to require a good deal of processor time. It is recommended that you use the default value wherever possible.

You can retrieve the current setting of NUMERIC DIGITS with the DIGITS built-in function.

NUMERIC FORM

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

You can retrieve the current setting of NUMERIC FORM with the FORM built-in function

NUMERIC FUZZ

NUMERIC FUZZ controls how many digits, at full precision, are ignored during a numeric comparison operation. If expression is omitted, the default is 0 digits. Otherwise expression must evaluate to 0 or a positive whole number rounded, if necessary, according to the current setting of NUMERIC DIGITS before it is used. The value used must be a positive whole number that is smaller than the current NUMERIC DIGITS setting.

FUZZ

FUZZ temporarily reduces the value of DIGITS by the FUZZ value before every numeric comparison operation. The numbers being compared are subtracted from each other under a precision of DIGITS minus FUZZ digits and this result is then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function

> **Note:** The three numeric settings are automatically saved across subroutine and internal function calls. See the CALL instruction for more details.

7.14 OPTIONS

OPTIONS [expression]

OPTIONS is used to pass special requests or parameters to the language processor. For example, they can be language processor options or they can define a special character set.

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

ETMODE	Specifies that literal strings containing double-byte character set (DBCS) characters can be used in the program.
NOETMODE	Specifies that literal strings containing DBCS characters cannot be used in the program. NOETMODE is the default.
EXMODE	Specifies that DBCS data in mixed strings is handled on a logical character basis by instructions, operators and functions. DBCS data integrity is maintained.

NOEXMODE Specifies that any data in strings is handled on a byte basis. The integrity of any DBCS characters might be lost. NOEXMODE is the default.

Notes: Because of the scanning procedures of the language processor, you are advised to place an OPTIONS ETMODE instruction as the first instruction of a program containing DBCS literal strings.

To ensure proper scanning of a program containing DBCS literals, type the words ETMODE, NOETMODE, EXMODE, and NOEXMODE as literal strings (that is, enclosed in quotation marks) in the OPTIONS instruction.

The OPTIONS ETMODE and OPTIONS EXMODE settings are saved and restored across subroutine and function calls.

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

7.15 PARSE

PARSE [UPPER] ARG [template list]

PARSE is used to assign data from various sources to one or more variables.

If specified, a template is a list of symbols separated by blanks or patterns.

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

If the UPPER option is specified, the data to be parsed is first translated to uppercase (for example, a lowercase a-z to an uppercase A-Z). Otherwise, no uppercase translation takes place during the parsing.

The data used for each variant of the PARSE instruction is as follows:

PARSE ARG

The strings passed to the program, subroutine, or function as the input argument list, are parsed. See the ARG instruction for details and examples.

Note: The argument strings to a REXX program or internal routine can also be retrieved or checked by using the ARG built-in function.

PARSE LINEIN

The next line from the default character input stream is parsed. PARSE LINEIN is a shorter form of the following instruction:

If no line is available, program execution will normally pause until a line is complete. Note that PARSE LINEIN should only be used when direct access to the character input stream is necessary. Normal line-by-line dialogue with the user should be carried out with the PULL or PARSE PULL instructions, to maintain generality and programmability.

To check if any lines are available in the default character input stream, use the built-in function LINES.

PARSE PULL

The next string from the queue is parsed. If the queue is empty, lines will be read from the default input, typically the user's keyboard. You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions respectively. You can find the number of lines currently in the queue by using the QUEUED built-in function. The queue remains active as long as the language processor is active. The queue can be altered by other programs in the system and can be used as a means of communication between these programs and programs written in REXX.

> **Note:** PULL and PARSE PULL read first from the current data queue; if the queue is empty, they read from the default input stream, STDIN (typically, the keyboard).

PARSE SOURCE

The data parsed describes the source of the program being run.

The source string contains the operating system name followed by either COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was invoked as a host command or from a function call in an expression or using the CALL instruction. These two tokens are followed by the complete path specification of the program file.

The string parsed might, therefore, be displayed as:

```
OS/2 COMMAND C:\OS2\REXTRY.CMD
```

PARSE VALUE

The expression is evaluated, and the result is the data that is parsed. Note that WITH is a subkeyword in this context and so cannot be used as a symbol within expression. For example:

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

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

PARSE VAR name

The value of the variable specified by name is parsed. The name must be a symbol that is valid as a variable name; that is, it can not start with a period or a digit. Note that the variable name is not changed unless it appears in the template. For example:

```
PARSE VAR string word1 string
```

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

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

also translates the data from string to uppercase before it is parsed.

PARSE VERSION

Information describing the language level and the date of the language processor is parsed. This consists of five words (delimited by blanks): first the string "REXXSAA", then the language level description ("4.00"), and finally the release date ("13 June 1989").

> **Note:** PARSE VERSION information should be parsed on a word basis rather than on an absolute column position basis.

7.16 PROCEDURE

PROCEDURE [EXPOSE [name1] [name2]…[name20]]

The PROCEDURE instruction can be used within an internal routine (subroutine or function) to protect all the existing variables by making them unknown to the instructions that follow. On executing a RETURN instruction, the original variables environment is restored and any variables used in the routine (which were not exposed) are dropped.

The EXPOSE option modifies this. Any variable specified by name is exposed, so that any reference to it (including setting and dropping) is made to the environment of the variables that the caller owns. With the EXPOSE option, you must specify at least one name, a symbol separated from any other name with one or more blanks. Optionally, you can enclose a single name in parentheses to denote a subsidiary variable list. Any variables not specified by name on a PROCEDURE EXPOSE instruction are still protected. Hence, some limited set of the caller's variables can be made accessible, and these variables can be changed (or new variables in this set can be created). All these changes will be visible to the caller upon RETURN from the routine.

The variables are exposed in sequence from left to right. It is not an error to specify a name more than once, or to specify a name that has not been used as a variable by the caller.

Example:

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

toft: procedure expose j k x.j
   say j k x.j  /* would display "1 K a"   */
   k=7; m=3     /* note "M" is not exposed */
   return
```

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

If name is enclosed in parentheses (blanks are not necessary either inside or outside the parentheses but can be added if desired) then, after that variable is exposed, the value of the variable is immediately used as a subsidiary list of variables. This list of variables must follow the same rules as the main list except that no parentheses or leading or trailing blanks are allowed. The variables named in a subsidiary list are also exposed from left to right.

Example:

```
j=1;k=6;m=9
a ='j k m'
test:procedure expose (a)   /* will expose j, k, and x */
```

If a stem is specified in names, all possible compound variables whose names begin with that stem are exposed. (A stem is a symbol containing only one period, which is the last character.

Example:

```
lucky7:Procedure Expose i j a. b.
/* This exposes "I", "J", and all variables whose */
/* names start with "A." or "B."    */
A.1='7'  /* This will set "A.1" in the caller's   */
         /* environment, even if it did not       */
         /* previously exist.                     */
```

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

Only one PROCEDURE instruction in each level of routine call is allowed; all others (and those met outside of internal routines) are in error.

Notes: An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those the caller owns.

The PROCEDURE instruction must be the first instruction executed after the CALL or function invocation, that is, it must be the first instruction following the label.

See the CALL instruction for details and examples of how routines are invoked.

7.17 PULL

PULL [template]

PULL is used to read a string from the head of the currently active REXX data queue. It is a short form of the instruction:

PARSE UPPER PULL [template]

The current head-of-queue is read as one string. If no template is specified, no further action is taken, and the string is effectively discarded. If specified, a template is a list of symbols separated by blanks or patterns. The string is translated to uppercase (for example, a lowercase a-z to an uppercase A-Z), and then parsed into variables according to the rules described in the section on parsing. Use the PARSE PULL instruction if you do not want uppercase translation.

Note: If the current data queue is empty, PULL reads instead from STDIN (typically, the keyboard). The length of data read by the PULL instruction is restricted to the length of strings contained by variables.

Example:

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

Here the dummy placeholder "." is used on the template to isolate the first word the user enters.

The number of lines currently in the queue can be found with the QUEUED built-in function.

7.18 PUSH

PUSH [expression]

PUSH is used to stack the string resulting from the evaluation of expression in LIFO (last in, first out) format onto the currently active REXX data queue. If you do not specify expression, a null string is stacked.

Example:

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

```
push a 2   /* Puts "Fred 2"    onto the queue */
```

The number of lines currently in the queue can be found with the QUEUED built-in function.

7.19 QUEUE

QUEUE [expression]

QUEUE is used to append the string resulting from expression to the tail of the currently active REXX data queue. That is, it is added in FIFO (first in, first out) format.

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

Example:

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

The number of lines currently in the queue can be found with the QUEUED built-in function.

7.20 RETURN

RETURN [expression]

RETURN is used to return control (and possibly a result) from a REXX program or internal routine to the point of its invocation.

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

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

If a function is being processed, the action taken is identical, except that expression must be specified on the RETURN instruction. The result of expression is then used in the original expression at the point where the function was invoked.

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

7.21 SAY

SAY [expression]

SAY is used to write to the output stream the result of evaluating expression. The result is usually displayed to the user, but the output destination can depend on the implementation. The result of expression can be of any length.

Notes: Data from the SAY instruction is sent to the default output stream (STDOUT:) However, the standard rules for redirecting output apply to SAY output.

The SAY instruction does not format data; line wrapping is handled

by the operating system and the hardware. However formatting is accomplished, the output data remains a single logical line.

Example:

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

7.22 SELECT

SELECT; WHEN expression [;] THEN [;] instruction [OTHERWISE] [;] [intruction]] END

SELECT is used to conditionally process one of several alternative instructions.

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

If none of the WHEN expressions evaluate to 1, control passes to the instructions, if any, after OTHERWISE. In this situation, the absence of OTHERWISE causes an error.

Example:

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

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

A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.

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

7.23 SIGNAL

SIGNAL lablename

or

SIGNAL [VALUE] expression

or

SIGNAL OFF type

or

SIGNAL ON type [NAME trapname]

Where type can be ERROR, FAILURE, HALT, NOVALUE, SYNTAX, or NOTREADY.

SIGNAL is used to cause an abnormal change in the flow of control, or, if ON or OFF is specified, controls the trapping of certain conditions.

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap.

To change the flow of control, a label name is derived from labelname or taken from the result of evaluating the expression after VALUE. The labelname you specify must be a symbol, treated literally, or a literal string that is taken as a constant. The subkeyword VALUE can be omitted if expression does not begin with a symbol or literal string (for example, if it starts with a special character, such as an operator or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then terminated; that is, they cannot be reactivated. Control then passes to the first label in the program that matches the required string, as though the search had started from the top of the program.

Example:

```
Signal fred;  /* Jump to label "FRED" below */
  ....
  ....
Fred: say 'Hi!'
```

Because the search effectively starts at the top of the program, if duplicates are present, control always passes to the first occurrence of the label in the program.

When control reaches the specified label, the line number of the SIGNAL instruction is assigned to the special variable SIGL. This can aid debugging because SIGNAL can determine the source of a jump to a label.

Using SIGNAL with the INTERPRET Instruction

If, as the result of an INTERPRET instruction, a SIGNAL instruction is issued or a trapped event occurs, the remainder of the strings being interpreted are not searched for the given label. In effect, labels within interpreted strings are ignored.

7.24 TRACE

TRACE [number]

or

TRACE [? ?2 type]

where type can be All, Commands, Error, Failure, Intermediates, Labels, Normal, Off, or Results

or

TRACE [string] | [symbol] | [[VALUE] expression]

TRACE is used for debugging. It controls the tracing action taken (that is, how much is displayed to the user) during execution of a REXX program. The syntax of TRACE is more concise than other REXX instructions. The economy of keystrokes for this instruction is especially convenient since TRACE is usually entered manually during interactive debugging.

The number is a whole number.

The string or expression evaluates to:

- A number option
- One of the valid prefix, alphabetic character (word) options, or both, shown in this panel
- Null.

The symbol is taken as a constant and is:

- A number option
- One of the valid prefix, alphabetic character (word) options, or both, shown in this panel.

The tracing action is determined from the option specified following TRACE or from the result of evaluating expression. If expression is used, you can omit the subkeyword VALUE as long as expression starts with a special character or operator (so it is not mistaken for a symbol or string).

Alphabetic Character (Word) Options

Although it is acceptable to enter the word in full, only the uppercase letter is significant; all other letters are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All	All clauses are traced (that is, displayed) before execution.
Commands	All host commands are traced before execution, and any error return code is displayed.
Error	Any host command resulting in an error return code is traced after execution.
Failure	Any host command resulting in a failure is traced after execution. This is the same as the Normal option.
Intermediates	All clauses are traced before execution. Intermediate results during evaluation of expressions and substituted names are also traced.
Labels	Labels passed during execution are traced. This is especially useful with debug mode, when the language processor pauses after each label. It is also convenient for the user to make note of all subroutine calls and signals.
Normal	Any failing host command is traced after execution. This is the default

setting.

Off	Nothing is traced, and the special prefix actions (see below) are reset to OFF.
Results	All clauses are traced before execution. Final results (contrast with Intermediates, above) of evaluating an expression are traced. Values assigned during PULL, ARG, and PARSE instructions are also displayed. This setting is recommended for general debugging.

Prefix Option

The prefix ? is valid either alone or with one of the alphabetic character options. You can specify the prefix more than once, if desired. Each occurrence of a prefix on an instruction reverses the action of the previous prefix. The prefix must immediately precede the option (no intervening blanks).

The prefix ? modifies tracing and execution. ? is used to control interactive debug. During normal execution, a TRACE instruction prefixed with ? causes interactive debug to be switched on.

When interactive debug is in effect, you can switch it off by issuing a TRACE instruction with a prefix ?. Repeated use of the ? prefix, therefore, switches you alternately in and out of interactive debug. Or, interactive debug can be turned off at any time by issuing TRACE O or TRACE with no options.

Numeric Options

If interactive debug is active and if the option specified is a positive whole number (or an expression that evaluates to a positive whole number), that number indicates the number of debug pauses to be skipped over. However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses.

If interactive debug is not active, numeric options are ignored.

Format of TRACE Output

Every clause traced will be displayed with automatic formatting (indentation) according to its logical depth of nesting and so on, and the results (if requested) are indented an extra two spaces and are enclosed in double quotation marks so that leading and trailing blanks are apparent.

All lines displayed during tracing have a three-character prefix to identify the type of data being traced. The prefixes and their definitions are the following:

-	Identifies the source of a single clause, that is, the data actually in the program.
+++	Identifies a trace message. This can be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program.
>>>	Identifies the result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.
>.>	Identifies the value assigned to a placeholder during parsing.

The following prefixes are only used if Intermediates (TRACE I) are being traced:

>C>	The data traced is the name of a compound variable, traced after substitution and

before use, provided that the name had the value of a variable substituted into it.

>F> The data traced is the result of a function call.

>L> The data traced is a literal (string, uninitialized variable, or constant symbol).

>O> The data traced is the result of an operation on two terms.

>P> The data traced is the result of a prefix operation.

>V> The data traced is the contents of a variable.

8 Functions

8.1 Syntax

You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:

function-name()

or

function-name(parm1,…parm20 [,])

Parameter 1 to 20 above can be expressions.

function-name is a literal string or a single symbol, that is taken to be a constant.

There can be up to a maximum of 20 expressions, separated by commas, between the parentheses. These expressions are called the arguments to the function. Each argument expression may include further function calls.

The (must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A blank operator is assumed at this point instead.)

The arguments are evaluated in turn from left to right and they are all then passed to the function. This then executes some operation (usually dependent on the argument strings passed, though arguments are not mandatory) and eventually returns a single character string. This string is then included in the original expression as though the entire function reference had been replaced by the name of a variable that contained that data.

For example, the function SUBSTR is built-in to the language processor and could be used as:

```
N1='abcdefghijk'
Z1='Part of N1 is: 'Substr(N1,2,7)
/* would set Z1 to 'Part of N1 is: bcdefgh' */
```

A function call without any arguments must always include the parentheses to be recognized as a function call.

```
date()  /* returns the date in the default format dd mon yyyy
*/
```

Calls to Functions and Subroutines

The mechanism for calling functions and subroutines is the same. The only difference is that functions must return data, whereas subroutines need not. The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it will later be possible to return to the point of invocation to resume processing. Control is then passed to the first label in the program that matches the name. As with a routine invoked by the CALL instruction, various other pieces of status information (TRACE and NUMERIC settings and so on) are also saved. See the CALL instruction for details of this. If an internal routine is to be called as a function, you must specify an expression in any RETURN instruction to return from the function. This is not necessary if the function is called only as a subroutine.

Example:

```
/* Recursive internal function execution... */
```

```
arg x
say x'! =' factorial(x)
exit

factorial: procedure   /* calculate factorial by.. */
  arg n                /*  .. recursive invocation. */
  if n=0 then return 1
  return  factorial(n-1) * n
```

FACTORIAL is unusual in that it invokes itself (this is known as recursive invocation). The PROCEDURE instruction ensures that a new variable n is created for each invocation).

Built-in These functions are always available and are defined later in this section.

External You can write or make use of functions that are external to a program and to the language processor. An external function can be written in any language, including REXX, that supports the system-dependent interfaces used by the language processor to invoke it. Again, when called as a function, it must return data to the caller.

Notes: Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller variables are always hidden and the status of internal values (NUMERIC settings and so on) start with their defaults (rather than inheriting those of the caller).

Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the invoked REXX program; in either case, you must specify an expression.

Search Order

The search order for functions is the same as in the preceding list. That is, internal labels take first precedence, then built-in functions, and finally external functions.

Internal labels are not used if the function name is given as a string (that is, is specified in quotation marks); in this case the function must be built-in or external. This lets you usurp the name of, for example, a built-in function to extend its capabilities, but still be able to invoke the built-in function when needed.

Example:

```
/* Modified DATE to return sorted date by default */
date: procedure
      arg in
      if in='' then in='Sorted'
      return 'DATE'(in)
```

Built-in functions have names in uppercase letters. The name in the literal string must be in uppercase for the search to succeed, as in the example. The same is usually true of external functions.

External functions and subroutines have a system-defined search order.

REXX searches for external functions in the following order:

1. Functions that have been loaded into the macrospace for pre-order execution
2. Functions that are part of a function package.
3. REXX functions in the current directory, with the current extension

4. REXX functions along environment PATH, with the current extension
5. REXX functions in the current directory, with the default extension
6. REXX functions along environment PATH, with the default extension
7. Functions that have been loaded into the macrospace for post-order execution.

Errors during Execution

If an external or built-in function detects an error of any kind, the language processor is informed, and a syntax error results. Processing of the clause that included the function call is therefore terminated. Similarly, if an external function fails to return data correctly, this is detected by the language processor and reported as an error.

If a syntax error occurs during the processing of an internal function, it can be trapped (using SIGNAL ON SYNTAX) and recovery may then be possible. If the error is not trapped, the program is terminated.

Return Values

A function normally returns a value that is substituted for the function call when the expression is evaluated.

How the value returned by a function (or any REXX routine) is handled depends on whether it is called by a function call or called as a subroutine with the CALL instruction.

- A routine called as a subroutine: If the routine returns a value, that value is stored in the special variable named RESULT. Otherwise, the RESULT variable is dropped, and its value is the string "RESULT".
- A routine called as a function: If the function returns a value, that value is substituted into the expression at the position where the function was called. Otherwise, REXX stops with an error message.

Examples:

```
/* Different ways to call a REXX procedure */
call Beep 500, 100         /* Example 1: a subroutine call */
bc = Beep(500, 100)        /* Example 2: a function call   */
Beep(500, 100)             /* Example 3: result passed as  */
                           /*            a command         */
```

- In Example 1, the built-in function BEEP is called as a REXX subroutine. The return value from BEEP is placed in the REXX special variable RESULT.
- Example 2 shows BEEP called as a REXX function. The return value from the function is substituted for the function call. The clause itself is an assignment instruction; the return value from the BEEP function is placed in the variable bc.
- In Example 3, the BEEP function is processed and its return value is substituted in the expression for the function call, just as in Example 2. In this case, however, the clause as a whole evaluates to a single expression; therefore the evaluated expression is passed to the current default environment as a command.

  Note: Many other languages (such as C) throw away the return value of a function if it is not assigned to a variable. In REXX, however, a value returned as in Example 3 is passed on to the current environment or subcommand handler. If that environment is the default, then this action will result in the operating system performing a disk search for what seems to be a command.

Built-in Functions

REXX provides a rich set of built-in functions. These include character manipulation, conversion, and information functions.

Here are some general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.
- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated.
- You can supply a null string where a string is referenced.
- If an argument specifies a length, it must be a nonnegative whole number. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate that you have omitted it; for example, DATATYPE(1,), like DATATYPE(1), would return NUM.
- If you specify a pad character, it must be exactly one character long.
- If a function has a suboption you can select by specifying the first character of a string, that character can be in uppercase or lowercase letters.
- Conversion between characters and hexadecimal involves the machine representation of character strings, and hence returns appropriately different results for ASCII and EBCDIC machines.

8.2 ABBREV

ABBREV(information, info [,length])

ABBREV returns 1 if info is equal to the leading characters of information and the length of info is not less than length. ABBREV returns 0 if neither of these conditions is met.

If specified, length must be a nonnegative whole number. The default for length is the number of characters in info.

Here are some examples:

```
ABBREV('Print','Pri')       ->    1
ABBREV('PRINT','Pri')       ->    0
ABBREV('PRINT','PRI',4)     ->    0
ABBREV('PRINT','PRY')       ->    0
ABBREV('PRINT','')          ->    1
ABBREV('PRINT','',1)        ->    0
```

Note: A null string will always match if a length of 0 or the default) is used. This allows a default keyword to be selected automatically if desired. For example:

```
say 'Enter option:';   pull option .
select  /* keyword1 is to be the default */
  when abbrev('keyword1',option) then ...
  when abbrev('keyword2',option) then ...
  ...
  otherwise nop;
end;
```

8.3 ABS (Absolute Value)

ABS(number)

ABS returns the absolute value of number. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

```
ABS('12.3')        ->    12.3
ABS(' -0.307')     ->    0.307
```

8.4 ADRESS

ADRESS()

ADDRESS returns the name of the environment to which host commands are currently being submitted. Trailing blanks are removed from the result.

Here are some examples:

```
ADDRESS()     ->    'CMD'         /* OS/2 environment */
ADDRESS()     ->    'EDIT'        /* possible editor  */
```

8.5 API Functions

The following built-in REXX functions can be used in a REXX program to register, drop, or query external function packages and to create and manipulate external data queues.

RXFUNCADD

RXFUNCADD(name,module,procedure)

RXFUNCADD registers the function name, making it available to REXX procedures. A zero return value signifies successful registration.

RXFUNCDROP

RXFUNCDROP(name)

RXFUNCDROP removes (deregisters) the function name from the list of available functions. A zero return value signifies successful removal.

RXFUNCQUERY

RXFUNCQUERY(name)

RXFUNCQUERY queries the list of available functions for a registration of the name function. The function returns a value of 0 if the function is registered, and a value of 1 if it is not.

RXQUEUE

RXQUEUE("Get" | "Set" newqueuename | "Delete" queuename | "Create" [queuename])

RXQUEUE is used in a REXX program to create and delete external data queues and to set and query their names.

8.6 ARG

ARG([n [,option]])

ARG returns an argument string, or information about the argument strings to a program or internal routine.

If you do not specify a parameter, the number of arguments passed to the program or internal routine is returned.

If only n is specified, the nth argument string is returned. If the argument string does not exist, the null string is returned. n must be a positive whole number.

If you specify option, ARG tests for the existence of the nth argument string. Valid options (of which only the capitalized letter is significant and all others are ignored) are:

Exists Returns 1 if the nth argument exists, that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted Returns 1 if the nth argument was omitted, that is, if it was not explicitly specified when the routine was called. Returns 0 otherwise.

Here are some examples:

```
/*  following "Call name;" (no arguments) */
ARG()         ->    0
ARG(1)         ->    ''
ARG(2)         ->    ''
ARG(1,'e')     ->    0
ARG(1,'O')     ->    1

/*  following "Call name 'a',,'b';" */
ARG()         ->    3
ARG(1)         ->    'a'
ARG(2)         ->    ''
ARG(3)         ->    'b'
ARG(n)         ->    ''    /* for n>=4 */
ARG(1,'e')     ->    1
ARG(2,'E')     ->    0
ARG(2,'O')     ->    1
ARG(3,'o')     ->    0
ARG(4,'o')     ->    1
```

Notes: You can retrieve and directly parse the argument strings to a program or internal routine using the ARG or PARSE ARG instructions.

Programs called as commands can have only 0 or 1 argument strings. The program has no argument strings if it is called with the name only and has 1 argument string if anything else (including blanks) is included with the command.

Programs called by the REXXSTART entry point can have multiple argument strings.

8.7 BEEP

BEEP frequency,duration

BEEP sounds the speaker at frequency (Hertz) for duration milliseconds. The frequency can be any number in the range 37 to 32767 Hertz. The duration can be any number in the range 1 to 60000 milliseconds.

This routine is most useful when called as a subroutine. A null string is returned if the routine is successful.

Here is an example:

```
/* C scale */
note.1 = 262    /* middle C */
note.2 = 294    /*    D     */
note.3 = 330    /*    E     */
note.4 = 349    /*    F     */
note.5 = 392    /*    G     */
note.6 = 440    /*    A     */
note.7 = 494    /*    B     */
note.8 = 524    /*    C     */
do i=1 to 8
  call beep note.i,250    /* hold each note for */
 /* one-quarter second */
end
```

8.9 BITAND

BITAND(string1 [, [string2] [,pad]])

BITAND returns a string composed of the two input strings logically compared, bit by bit, using the AND operator. The length of the result is the length of the longer of the two strings. If no pad character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```
BITAND('73'x,'27'x) ->    '23'x
BITAND('13'x,'5555'x)          ->    '1155'x
BITAND('13'x,'5555'x,'74'x)    ->    '1154'x
BITAND('pQrS',,'DF'x)          ->    'PQRS' /* ASCII only  */
```

8.10 BITOR

BITOR(string1 [, [string2] [,pad]])

BITOR returns a string composed of the two input strings logically compared, bit by bit, using the OR operator. The length of the result is the length of the longer of the two strings. If no pad character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```
BITOR('15'x,'24'x) ->    '35'x
BITOR('15'x,'2456'x)          ->    '3556'x
BITOR('15'x,'2456'x,'F0'x)    ->    '35F6'x
BITOR('1111'x,,'4D'x)         ->    '5D5D'x
BITOR('pQrS',,'20'x)          ->    'pqrs' /* ASCII only  */
```

8.11 BITXOR

BITXOR(string1[, [string2] [,pad]])

BITXOR returns a string composed of the two input strings logically compared bit by bit using the exclusive OR operator. The length of the result is the length of the longer of the two strings. If no pad character is provided, the XOR operation terminates when the shorter

of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If pad is provided, it is used to extend the shorter of the two strings on the right, before carrying out the logical operation. The default for string2 is the zero length (null) string.

Here are some examples:

```
BITXOR('12'x,'22'x)               ->  '30'x
BITXOR('1211'x,'22'x)             ->  '3011'x
BITXOR('C711'x,'222222'x,' ')     ->  'E53302'x  /* ASCII  */
BITXOR('1111'x,'444444'x)         ->  '555544'x
BITXOR('1111'x,'444444'x,'40'x)   ->  '555504'x
BITXOR('1111'x,,'4D'x)            ->  '5C5C'x
```

8.12 B2X (Binary to Hexadecimal)

B2X(binary_string)

Converts binary_string, a string of binary (0 or 1) digits, to an equivalent string of hexadecimal characters. You can optionally include blanks in binary_string (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string uses uppercase letters for the values A-F, and does not include blanks.

binary_string can be of any length; if it is the null string, then a null string is returned. If the number of binary digits in the string is not a multiple of four, then up to three 0 digits will be added on the left before the conversion to make a total that is a multiple of four.

Here are some examples:

```
B2X('11000011')    ==   'C3'
B2X('10111')       ==   '17'
B2X('101')         ==   '5'
B2X('1 1111 0000') ==   '1F0'
```

B2X() can be combined with the functions X2D() and X2C() to convert a binary number into other forms. For example:

```
X2D(B2X('10111'))  ==   '23'   /* decimal 23 */
```

8.13 CENTER/CENTRE

CENTER or **CENTRE(string,length [,pad])**

CENTER or CENTRE returns a string of length length with string centered in it, with pad characters added as necessary to make up length. The default pad character is blank. If the string is longer than length, it will be truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand end loses or gains one more character than the left-hand end.

Here are some examples:

```
CENTER(abc,7)               ->    '  ABC  '
CENTER(abc,8,'-')           ->    '--ABC---'
CENTRE('The blue sky',8)    ->    'e blue s'
CENTRE('The blue sky',7)    ->    'e blue '
```

Note: This function can be called either CENTRE or CENTER, which avoids errors due to the difference between the British and American spellings.

8.14 CHARIN

CHARIN([name] [, [start] [,length]])

CHARIN returns a string of up to length characters read from the character input stream name. The form of the name is implementation dependent. If name is omitted, characters are read from the default input stream, STDIN:. The default length is 1.

For persistent streams, a read position is maintained for each stream. This is the same as the write position. Any read from the stream will by default start at the current read position. When the read is completed, the read position is increased by the number of characters read. A start value can be given to specify an explicit read position. This read position must be positive and within the bounds of the stream, and must not be specified for a transient stream (a port or other serial device). A value of 1 for start refers to the first character in the stream.

If you specify a length of 0, then the read position will be set to the value of start, but no characters are read and the null string is returned.

In a transient stream, if there are fewer then length characters available, then execution of the program will normally stop until sufficient characters do become available. If, however, it is impossible for those characters to become available due to an error or other problem, the NOTREADY condition is raised and CHARIN will return with fewer than the requested number of characters.

Here are some examples:

```
CHARIN(myfile,1,3)    ->   'MFC'     /* the first 3     */
                                     /* characters      */
CHARIN(myfile,1,0)    ->   ''        /* now at start    */
CHARIN(myfile)        ->   'M'       /* after last call */
CHARIN(myfile,,2)     ->   'FC'      /* after last call */
/* Reading from the default input (here, the keyboard) */
                               /* User types 'abcd efg' */
CHARIN()              ->   'a'           /* default is  */
                                         /* 1 character */
CHARIN(,,5)           ->   'bcd e'
```

Note: CHARIN returns all characters that appear in the stream including control characters such as line feed, carriage return, and end of file.

When CHARIN is used to read from the keyboard, program execution stops until you press the Enter key.

8.15 CHAROUT

CHAROUT([name] [, [string] [,start]])

CHAROUT returns the count of characters remaining after attempting to write string to the character output stream name. The form of the name is implementation dependent. If name is omitted, characters in string will be written to the default output stream, STDOUT: (normally the display) in the operating system. string can be the null string, in which case no characters are written to the stream and 0 is always returned.

For persistent streams, a write position is maintained for each stream. This is the same as the read position. Any write to the stream starts at the current write position by default. When the write is completed the write position is increased by the number of characters written. The initial write position is the end of the stream, so that calls to CHAROUT normally append to the end of the stream.

A start value can be given to specify an explicit write position for a persistent stream. This write position must be a positive whole number within the bounds of the stream (though it

can specify the character position immediately after the end of the stream). A value of 1 for start refers to the first character in the stream.

Note: In some environments, overwriting a stream with CHAROUT or LINEOUT can erase (destroy) all existing data in the stream.

The string can be omitted for persistent streams. In this case, the write position is set to the value of start that was given, no characters are written to the stream, and 0 is returned. If neither start nor string are given, the write position is set to the end of the stream. This use of CHAROUT can also have the side effect of closing or fixing the file in environments which support this concept. Again, 0 is returned. If you do not specify start or string, the stream is closed. Again, 0 is returned.

Processing of the program normally stops until the output operation is effectively complete. If, however, it is impossible for all the characters to be written, the NOTREADY condition is raised and CHAROUT returns with the number of characters that could not be written (the residual count).

Here are some examples:

```
CHAROUT(myfile,'Hi')      ->   0   /* normally */
CHAROUT(myfile,'Hi',5)    ->   0   /* normally */
CHAROUT(myfile,,6)        ->   0   /* now at char 6 */
CHAROUT(myfile)           ->   0   /* at end of stream */
CHAROUT(,'Hi')->               0   /* normally */
CHAROUT(,'Hello')         ->   2   /* maybe    */
```

Note: This routine is often best called as a subroutine. The residual count is then available in the variable RESULT.

For example:

```
Call CHAROUT myfile,'Hello'
Call CHAROUT myfile,'Hi',6
Call CHAROUT myfile
```

8.16 CHARS

CHARS([name])

CHARS returns the total number of characters remaining in the character input stream name. The count includes any line separator characters, if these are defined for the stream, and in the case of persistent streams, is the count of characters from the current read position to the end of the stream. If name is omitted, the default input stream will be used.

The total number of characters remaining cannot be determined for some streams (for example, STDIN:). For these streams, the CHARS function returns 1 to indicate that data is present, or 0 if no data is present.

Here are some examples:

```
CHARS(myfile)      ->   42   /* perhaps */
CHARS(nonfile)     ->   0    /* perhaps */
CHARS()            ->   1    /* perhaps */
```

8.17 COMPARE

COMPARE(string1,string2 [,pad])

COMPARE returns 0 if the strings, string1 and string2, are identical. Otherwise, COMPARE returns the position of the first character that does not match. The shorter string is padded on the right with pad if necessary. The default pad character is a blank.

Here are some examples:

```
COMPARE('abc','abc')          ->    0
COMPARE('abc','ak')           ->    2
COMPARE('ab ','ab')           ->    0
COMPARE('ab ','ab',' ')       ->    0
COMPARE('ab ','ab','x')       ->    3
COMPARE('ab-- ','ab','-')     ->    5
```

8.18 CONDITION

CONDITION([option])

CONDITION returns the condition information associated with the current trapped condition. You can request four pieces of information:

- The name of the current trapped condition
- Any descriptive string associated with that condition
- The instruction executed as a result of the condition trap (CALL or SIGNAL)
- The status of the trapped condition.

The following options (of which only the capitalized letter is needed, all others are ignored) can be used to obtain the following information:

Condition name	Returns the name of the current trapped condition.
Description	Returns any descriptive string associated with the current trapped condition. If no description is available, a null string is returned.
Instruction,	Returns the keyword for the instruction executed when the current condition was trapped. The keywords are CALL or SIGNAL. This is the default if you omit option.
Status	Returns the status of the current trapped condition. This can change during execution, and is either ON - the condition is enabled OFF - the condition is disabled DELAY - any new occurrence of the condition is delayed.

If no condition has been trapped (that is, there is no current trapped condition) then the CONDITION function returns a null string in all four cases.

Here are some examples:

```
CONDITION()             ->    'CALL'        /* perhaps */
CONDITION('C')          ->    'FAILURE'
CONDITION('I')          ->    'CALL'
CONDITION('D')          ->    'FailureTest'
CONDITION('S')          ->    'OFF'         /* perhaps */
```

Note: The condition information returned by the CONDITION function is saved and restored across subroutine calls (including those caused by a CALL ON condition trap). Therefore, once a subroutine invoked due to a CALL ON trap has returned, the current trapped condition reverts to the current condition before the CALL took place. CONDITION returns the values it returned before the condition was trapped.

8.19 COPIES

COPIES(string,n)

COPIES returns n concatenated copies of string. n must be a nonnegative whole number.

Here are some examples:

```
COPIES('abc',3)     ->     'abcabcabc'
COPIES('abc',0)     ->     ''
```

8.20 C2D (Character to Decimal)

C2D(string [,n])

C2D returns the decimal value of the binary representation of string. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS.

If string is the null string, then 0 is returned.

If n is not specified, string is processed as an unsigned binary number.

Here are some examples:

```
C2D('09'X)       ->         9
C2D('81'X)       ->       129
C2D('FF81'X)     ->     65409
C2D('a')         ->        97     /* ASCII */
```

If n is specified, the given string is padded on the left with 00x characters (note, not sign-extended), or truncated on the left to n characters. The resulting string of n hexadecimal digits is taken to be a signed binary number: positive if the leftmost bit is OFF, and negative, in two's complement notation, if the leftmost bit is ON. If n is 0, then 0 is always returned.

Here are some examples:

```
C2D('81'X,1)       ->     -127
C2D('81'X,2)       ->      129
C2D('FF81'X,2)     ->     -127
C2D('FF81'X,1)     ->     -127
C2D('FF7F'X,1)     ->      127
C2D('F081'X,2)     ->    -3967
C2D('F081'X,1)     ->     -127
C2D('0031'X,0)     ->        0
```

Implementation maximum: The input string cannot have more than 250 characters that will be significant in forming the final result. Leading sign characters (00x and ffx) do not count toward this total.

8.21 C2X (Character to Hexadecimal)

C2X(string)

C2X converts a character string to its hexadecimal representation. The data to be converted can be of any length. The returned string contains twice as many bytes as the input string because it is in literal string notation.

Here are some examples:

```
C2X('0123'X)     ->     '0123'
C2X('ZD8')       ->     '5A4438'     /*   ASCII    */
```

8.22 DATATYPE

DATATYPE(string [,type])

DATATYPE determines whether 'data' is numeric or alphabetic and returns a result of NUM or CHAR. If only string is specified, the returned result is NUM if string is a valid REXX number (any format); otherwise CHAR will be the returned result.

If type is specified, the returned result is 1 if string matches the type; otherwise, a 0 is returned. If string is null, 0 is returned (except when type is X, which returns 1). The following is a list of valid types. Only the capitalized letter is significant (all others are ignored).

Alphanumeric	Returns 1 if string contains only characters from the ranges a-z, A-Z, and 0-9.
Bits	Returns 1 if string contains only the characters 0 and/or 1.
C	Returns 1 if string is a mixed SBCS/DBCS string.
Dbcs	Returns 1 if string is a pure DBCS string.
Lowercase	Returns 1 if string contains only characters from the range a-z.
Mixed case	Returns 1 if string contains only characters from the ranges a-z and A-Z.
Number	Returns 1 if string is a valid REXX number.
Symbol	Returns 1 if string contains only characters that are valid in REXX symbols. Note that both uppercase and lowercase letters are permitted.
Uppercase	Returns 1 if string contains only characters from the range A-Z.
Whole number	Returns 1 if string is a REXX whole number under the current setting of NUMERIC DIGITS.
HeXadecimal	Returns 1 if string contains only characters from the ranges a-f, A-F, 0-9, and blank (so long as blanks only appear between pairs of hexadecimal characters). Also returns 1 if string is a null string.

```
DATATYPE(' 12 ')          ->   'NUM'
DATATYPE('')              ->   'CHAR'
DATATYPE('123*')          ->   'CHAR'
DATATYPE('12.3','N')      ->    1
DATATYPE('12.3','W')      ->    0
DATATYPE('Fred','M')      ->    1
DATATYPE('','M')          ->    0
DATATYPE('Fred','L')      ->    0
DATATYPE('?20K','S')      ->    1
DATATYPE('BCd3','X')      ->    1
DATATYPE('BC d3','X')     ->    1
```

8.23 DATE

DATE([option])

DATE returns, by default, the local date in the format: dd mon yyyy (for example, 27 Aug 1988), with no leading zero or blank on the day. For mon, the first three characters of the English name of the month will be used.

The following options (of which only the capitalized letter is needed, all others are ignored) can be used to obtain alternative formats:

Basedate	Returns the number of complete days (that is, not including the current day)

since and including the base date, January 1, 0001, in the format: dddddd (no leading zeros). The expression DATE(B)//7 returns a number in the range 0-6, where 0 is Monday and 6 is Sunday.
Note: The origin of January 1, 0001 is based on the Gregorian calendar. Though this calendar did not exist prior to 582, Basedate is calculated as if it did: 365 days per year, an extra day every four years except century years, and leap centuries if the century is divisible by 400. It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Days Returns the number of days, including the current day, so far in this year in the format: ddd (no leading zeros)

European Returns date in the format: dd/mm/yy.

Language Returns date in an implementation and language dependent or local date format. If no local format is available, the default format is returned.
Note: This format is intended to be used as a whole; REXX programs should not make any assumptions about the form or content of the returned string.

Month Returns full English name of the current month, for example, August

Normal Returns date in the default format: dd mon yyyy

Ordered Returns date in the format: yy/mm/dd (suitable for sorting, and so on.)

Sorted Returns date in the format: yyyymmdd (suitable for sorting, and so on.)

Usa Returns date in the format: mm/dd/yy

Weekday Returns the English name for the day of the week, in mixed case. For example, Tuesday.

Here are some examples:

```
DATE()          ->    '27 Aug 1988'  /*  perhaps  */
DATE('B')       ->    725975
DATE('D')       ->    240
DATE('E')       ->    '27/08/88'
DATE('L')       ->    '27 August 1988'
DATE('M')       ->    'August'
DATE('N')       ->    '27 Aug 1988'
DATE('O')       ->    '88/08/27'
DATE('S')       ->    '19880827'
DATE('U')       ->    '08/27/88'
DATE('W')       ->    'Saturday'
```

Note: The first call to DATE or TIME in one expression causes a time stamp to be made which is then used for all calls to these functions in that expression. Therefore, if multiple calls to any of the DATE and/or TIME functions are made in a single expression, they are guaranteed to be consistent with each other.

8.24 DELSTR (Delete String)

DELSTR(string,n [,length])

DELSTR deletes the substring of string that begins at the nth character, and is of length length. If length is not specified, the rest of string is deleted. If n is greater than the length of string, the string is returned unchanged. n must be a positive whole number.

Here are some examples:

```
DELSTR('abcd',3)        ->    'ab'
DELSTR('abcde',3,2)     ->    'abe'
DELSTR('abcde',6)       ->    'abcde'
```

8.25 DELWORD

DELWORD(string,n [,length])

DELWORD deletes the substring of string that starts at the nth word. The length option refers to the number of blank-delimited words. If length is omitted, it defaults to be the remaining words in string. n must be a positive whole number. If n is greater than the number of words in string, string is returned unchanged. The string deleted includes any blanks following the final word involved.

Here are some examples:

```
DELWORD('Now is the  time',2,2)  ->  'Now time'
DELWORD('Now is the time ',3)    ->  'Now is '
DELWORD('Now is the  time',5)    ->  'Now is the  time'
```

8.26 DIGITS

DIGITS()

DIGITS returns the current setting of NUMERIC DIGITS.

Here is an example:

```
DIGITS()    ->    9      /* by default */
```

8.27 D2C (Decimal to Character)

D2C(wholenumber [,n])

D2C returns a character string that is the ASCII representation of the decimal number. If you specify n, it is the length of the final result in characters and leading blanks are added to the output character.

If n is not specified, wholenumber must be a nonnegative number and the resulting length is as needed; therefore, the returned result has no leading 00x characters.

Here are some examples:

```
D2C(65)      ->   'A'      /* '41'x is an ASCII 'A'    */
D2C(65,1)    ->   'A'
D2C(65,2)    ->   ' A'
D2C(65,5)    ->   '    A'
D2C(109)     ->   'm'      /* '6D'x  is an ASCII 'm'   */
D2C(-109,1)  ->   'ô'      /* '147'x is an ASCII 'ô'   */
D2C(76,2)    ->   ' L'     /* '76'x  is an ASCII ' L'  */
D2C(-180,2)  ->   ' L'
```

8.28 D2X (Decimal to Hexadecimal)

D2X(wholenumber [,n])

D2X returns a string of hexadecimal characters that is the hexadecimal representation of the decimal number.

If n is not specified, wholenumber must be a nonnegative number and the returned result has no leading 0 characters.

If n is specified, it is the length of the final result in characters; that is, after conversion the input string is sign-extended to the required length. If the number is too big to fit into n characters, it is shortened on the left.

Here are some examples:

```
D2X(9)          ->    '9'
D2X(129)        ->    '81'
D2X(129,1)      ->    '1'
D2X(129,2)      ->    '81'
D2X(129,4)      ->    '0081'
D2X(257,2)      ->    '01'
D2X(-127,2)     ->    '81'
D2X(-127,4)     ->    'FF81'
D2X(12,0)       ->    ''
```

Implementation maximum: The output string cannot have more than 250 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

8.29 DIRECTORY

DIRECTORY([newdictionary])

DIRECTORY returns the current directory, first changing it to newdirectory if an argument is supplied and the named directory exists.

The return string includes a drive letter prefix as the first two characters of the directory name. Specifying a drive letter prefix as part of newdirectory causes the specified drive to become the current drive. If a drive letter is not specified, then the current drive remains unchanged.

For example, the following program fragment saves the current directory and switches to a new directory; it performs an operation there, and then returns to the former directory.

```
/* get current directory */
curdir = directory()
/* go play a game */
newdir = directory("d:/usr/games")
if newdir = "d:/usr/games" then
   do
   fortune  /* tell a fortune */
/* return to former directory */
   call directory curdir
   end
else
   say 'Can't find /usr/games'
```

8.30 ERRORTEXT

ERRORTEXT(n)

ERRORTEXT returns the error message associated with error number n. n must be in the range 0-99, and any other value is an error. If n is in the allowed range, but is not a defined REXX error number, the null string is returned.

Here are some examples:

```
ERRORTEXT(16)     ->    'Label not found'
ERRORTEXT(60)     ->    ''
```

8.31 ENDLOCAL

ENDLOCAL()

ENDLOCAL restores the drive directory and environment variables in effect before the last SETLOCAL function was executed. If ENDLOCAL is not included in a procedure, then the initial environment saved by SETLOCAL will be restored upon exiting the procedure.

ENDLOCAL returns a value of 1 if the initial environment is successfully restored, and a value of 0 if no SETLOCAL has been issued or if the actions is otherwise unsuccessful.

Note: The REXX SETLOCAL and ENDLOCAL functions can be nested.

Here is an example:

```
n = SETLOCAL()          /* saves the current environment    */
      /* The program can now change environment   */
      /* variables (with the VALUE function) and  */
      /* then work in that changed environment. */
n = ENDLOCAL()          /* restores the initial environment */
```

For additional examples, view the SETLOCAL function.

8.32 FILESPEC

FILESPEC(option,filespec)

FILESPEC returns a selected element of filespec, a given file specification, identified by one of the following strings for option:

Drive	The drive letter of the given filespec.
Path	The directory path of the given filespec.
Name	The filename of the given filespec.

If the requested string is not found, then FILESPEC returns a null string (" ").

Note: Only the initial letter of option is needed.

Here are some examples:

```
thisfile = "C:\UTIL\EXAMPLE.EXE"
say FILESPEC("drive",thisfile)     /* says "C:"          */
say FILESPEC("path",thisfile)      /* says "\OS2\UTIL\"  */
say FILESPEC("name",thisfile)      /* says "EXAMPLE.EXE" */

part = "name"
say FILESPEC(part,thisfile)        /* says "EXAMPLE.EXE" */
```

8.33 FORM

FORM()

FORM returns the current setting of NUMERIC FORM.

Here is an example:

```
FORM()    ->    'SCIENTIFIC'  /* by default */
```

8.34 FORMAT

FORMAT(number [, [before] [, [after] [, [expp] [,expt]]]])

FORMAT returns number rounded and formatted.

The number is first rounded and formatted to standard REXX rules, just as though the operation number+0 had been carried out. If only number is given, the result is precisely that of this operation. If any other options are specified, the number is formatted as follows.

The before and after options describe how many characters are to be used for the integer part and decimal part of the result respectively. If either or both of these are omitted, the number of characters used for that part is as needed.

If before is not large enough to contain the integer part of the number, an error results. If before is too large, the number is padded on the left with blanks. If after is not the same size as the decimal part of the number, the number will be rounded (or extended with zeros) to fit. Specifying 0 will cause the number to be rounded to an integer.

Here are some examples:

```
FORMAT('3',4)            ->    '   3'
FORMAT('1.73',4,0)       ->    '   2'
FORMAT('1.73',4,3)       ->    '   1.730'
FORMAT('-.76',4,1)       ->    '  -0.8'
FORMAT('3.03',4)         ->    '   3.03'
FORMAT(' - 12.73',,4)    ->    '-12.7300'
FORMAT(' - 12.73')       ->    '-12.73'
FORMAT('0.000')          ->    '0'
```

The first three arguments are as previously described. In addition, expp and expt control the exponent part of the result: expp sets the number of places to be used for the exponent part; the default is to use as many as needed. The expt sets the trigger point for use of exponential notation. If the number of places needed for the integer part exceeds expt, exponential notation is used. Likewise, exponential notation is used if the number of places needed for the decimal part exceeds twice expt. The default is the current setting of NUMERIC DIGITS. If 0 is specified for expt, exponential notation is always used unless the exponent would be 0. The expp must be less than 10, but there is no limit on the other arguments. If 0 is specified for the expp field, no exponent is supplied, and the number is expressed in simple form with added zeros as necessary (this overrides a 0 value of expt). Otherwise, if expp is not large enough to contain the exponent, an error results. If the exponent is 0, in this case (a non-zero expp), then expp+2 blanks are supplied for the exponent part of the result.

Here are some examples:

```
FORMAT('12345.73',,,2,2)    ->    '1.234573E+04'
FORMAT('12345.73',,3,,0)    ->    '1.235E+4'
FORMAT('1.234573',,3,,0)    ->    '1.235'
FORMAT('12345.73',,,3,6)    ->    '12345.73'
FORMAT('1234567e5',,3,0)    ->    '123456700000.000'
```

8.35 FUZZ

FUZZ()

FUZZ returns the current setting of NUMERIC FUZZ.

Here is an example:

```
FUZZ()    ->    0     /* by default */
```

8.36 INSERT

INSERT(new,target [, [n] [, [length] [,pad]]])

INSERT inserts the string new, padded to length length, into the string target after the nth character. If specified, n must be a nonnegative whole number. If n is greater than the length of the target string, padding is added there also. The default pad character is a blank. The default value for n is 0, which means insert before the beginning of the string.

Here are some examples:

```
INSERT(' ','abcdef',3)          ->    'abc def'
INSERT('123','abc',5,6)         ->    'abc  123   '
INSERT('123','abc',5,6,'+')     ->    'abc++123+++'
INSERT('123','abc') ->                '123abc'
INSERT('123','abc',,5,'-')      ->    '123--abc'
```

8.37 LASTPOS

LASTPOS(needle,haystack [,start])

LASTPOS returns the position of the last occurrence of one string, needle, in another, haystack. If the string needle is not found, 0 is returned. By default the search starts at the last character of haystack (that is, start=LENGTH(string)) and scans backwards. You can override this by specifying start, as the point at which the backward scan starts. start must be a positive whole number, and defaults to LENGTH(string) if larger than that value.

Here are some examples:

```
LASTPOS(' ','abc def ghi')      ->    8
LASTPOS(' ','abcdefghi')        ->    0
LASTPOS(' ','abc def ghi',7)    ->    4
```

8.38 LEFT

LEFT(string,length [,pad])

LEFT returns a string of length length, containing the leftmost length characters of string. The string returned is padded with pad characters (or truncated) on the right as needed. The default pad character is a blank. length must be nonnegative. The LEFT function is exactly equivalent to SUBSTR(string,1,length[,pad]).

Here are some examples:

```
LEFT('abc d',8)         ->    'abc d   '
LEFT('abc d',8,'.')     ->    'abc d...'
LEFT('abc  def',7)      ->    'abc  de'
```

8.39 LENGTH

LENGTH(string)

LENGTH returns the length of string.

Here are some examples:

```
LENGTH('abcdefgh')    ->    8
LENGTH('abc defg')    ->    8
LENGTH('')            ->    0
```

8.40 LINEIN

LINEIN([name] [, [line] [,count]])

LINEIN returns count (0 or 1) lines read from the character input stream name. The form of the name is implementation dependent. If name is omitted, the line is read from the default input stream. The default count is 1.

For persistent streams, a read position is maintained for each stream. This is the same as the write position. Any read from the stream starts at the current read position by default. A call to LINEIN will return a partial line if the current read position is not at the start of a line. When the read is completed, the read position is moved to the beginning of the next line. The read position may be set to the beginning of the stream by giving line a value of 1-.

If a count of 0 is given, then no characters are read and the null string is returned.

For transient streams, if a complete line is not available in the stream, then execution of the program will normally stop until the line is complete. If, however, it is impossible for a line to be completed due to an error or other problem, the NOTREADY condition is raised and LINEIN returns whatever characters are available.

Here are some examples:

```
LINEIN()                          /* Reads one line from the    */
                                  /* default input stream;       */
                                  /* normally this is an entry  */
                                  /* typed at the keyboard      */

myfile = 'ANYFILE.TXT'
LINEIN(myfile)     -> 'Current line'    /* Reads one line from  */
                                  /* ANYFILE.TXT, beginning     */
                                  /* at the current read        */
                                  /* position. (If first call,  */
                                  /* file is opened and the     */
                                  /* first line is read.)       */

LINEIN(myfile,1,1) ->'first line' /*  Opens and reads the first */
                                  /* line of ANYFILE.TXT (if    */
                                  /* the file is already open,  */
                                  /* reads first line); sets    */
                                  /* read position on the       */
                                  /* second line.               */

LINEIN(myfile,1,0) ->  ''         /* No read; opens ANYFILE.TXT */
                                  /* (if file is already open,  */
                                  /* sets the read position to  */
                                  /* the first line).           */

LINEIN(myfile,,0)  ->  ''         /* No read; opens ANYFILE.TXT */
                                  /* (no action if the file is  */
                                  /* already open).             */

LINEIN("QUEUE:") -> 'Queue line' /* Read a line from the queue; */
                                 /* If the queue is empty, the  */
                                 /* program waits until a line  */
                                 /*  is put on the queue.       */
```

Note: If the intention is to read complete lines from the default character stream, as in a simple dialogue with a user, use the PULL or PARSE PULL instructions instead for simplicity and for improved programmability. The PARSE LINEIN instruction is also useful in certain cases.

8.41 LINEOUT

LINEOUT([name] [, [string] [,line]])

LINEOUT returns the count of lines remaining after attempting to write string to the character output stream name. The count is either 0 (meaning the line was successfully written) or 1 (meaning that an error occurred while writing the line). string can be the null string, in which case only the action associated with completing a line is taken. LINEOUT adds a line-feed and a carriage-return character to the end of string.

The form of the name is implementation dependent. If name is omitted, the line is written to the default output stream, STDOUT.

For persistent streams, a write position is maintained for each stream. Any write to the stream starts at the current write position by default. Characters written by a call to LINEOUT can be added to a partial line. LINEOUT conceptually terminates a line at the end of each call. When the write is completed, the write position is set to the beginning of the line following the one just written. The initial write position is the end of the stream, so that calls to LINEOUT normally append lines to the end of the stream.

You can set the write position to the first character of a persistent stream by giving a value of 1 (the only valid value) for line.

> **Note:** In some environments, overwriting a stream using CHAROUT or LINEOUT can erase (destroy) all existing data in the stream.

You can omit the string for persistent streams. If you specify line, the write position is set to the beginning of the stream, but nothing is written to the stream, and 0 is returned. If you specify neither line nor string, the write position is set to the end of the stream. This use of LINEOUT has the effect of closing the stream in environments that support this concept.

Execution of the program normally stops until the output operation is effectively completed. If, however, it is impossible for a line to be written, the NOTREADY condition is raised and LINEOUT returns with a result of 1 (that is, the residual count of lines written).

Here are some examples:

```
LINEOUT(,'Display this')        /* Writes string to the default   */
                                /* output stream (normally, the   */
                                /* display); returns 0 if         */
                                /* successful                     */

myfile = 'ANYFILE.TXT'
LINEOUT(myfile,'A new line')    /* Opens the file ANYFILE.TXT and */
                                /* appends the string to the end. */
                                /* If the file is already open,   */
                                /* the string is written at the   */
                                /*  current write position.       */
                                /*  Returns 0 if successful.      */

LINEOUT(myfile,'A new start',1)/* Opens the file (if not already */
                                /* open); overwrites first line   */
                                /* with a new line.               */
                                /* Returns 0 if successful.       */

LINEOUT(myfile,,1)              /* Opens the file (if not already */
                                /* open). No write; sets write    */
                                /* position at first character.   */

LINEOUT(myfile)                 /* Closes ANYFILE.TXT             */
```

LINEOUT is often most useful when called as a subroutine. The return value is then available in the variable RESULT. For example:

```
Call LINEOUT 'A:rexx.bat','Shell',1
Call LINEOUT ,'Hello'
```

Note: If the lines are to be written to the default output stream without the possibility of error, use the SAY instruction instead.

8.42 LINES

LINES([name])

LINES returns 1 if any data remains between the current read position and the end of the character input stream name, and returns 0 if no data remains. In effect, LINES reports whether a read action performed by CHARIN or LINEIN will succeed.

The form of the name is implementation dependent. If you omit name, then the presence or absence of data in the default input stream (STDIN:) is returned.

Here are some examples:

```
LINES(myfile)     ->    0    /* at end of the file   */

LINES()           ->    1    /* data remains in the  */
                             /* default input stream */
                             /* STDIN:               */

LINES("COM1:")    ->    1    /* An OS/2 device name  */
                             /* always returns '1'   */
```

Note: The CHARS function returns the number of characters in a persistent stream or the presence of data in a transient stream.

8.43 MAX

MAX(number[,number2][,number3]…[,number20])

MAX returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. You can specify up to 20 numbers and can nest calls to MAX if more arguments are needed.

Here are some examples:

```
MAX(12,6,7,9)                                  ->    12
MAX(17.3,19,17.03)                             ->    19
MAX(-7,-3,-4.3)                                ->    -3
MAX(1,2,3,4,5,6,7,8,9,MAX(10,11,12,13))    ->    13
```

8.44 MIN

MIN(number[,number2][,number3]…[,number20])

MIN returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS. Up to 20 numbers can be specified, although calls to MIN can be nested if more arguments are needed.

Here are some examples:

```
MIN(12,6,7,9)          ->     6
MIN(17.3,19,17.03)     ->    17.03
MIN(-7,-3,-4.3)        ->    -7
```

8.45 OVERLAY

OVERLAY(new,target [, [n] [, [length] [,pad]]])

OVERLAY returns the string target, which, starting at the nth character, is overlaid with the string new, padded or truncated to length length. If length is specified, it must be positive or zero. If n is greater than the length of the target string, padding is added before the new string. The default pad character is a blank, and the default value for n is 1. If you specify n, it must be a positive whole number.

Here are some examples:

```
OVERLAY(' ','abcdef',3)          ->    'ab def'
OVERLAY('.','abcdef',3,2)        ->    'ab. ef'
OVERLAY('qq','abcd')             ->    'qqcd'
OVERLAY('qq','abcd',4)           ->    'abcqq'
OVERLAY('123','abc',5,6,'+')     ->    'abc+123+++'
```

8.46 POS

POS(needle,haystack [,start])

POS returns the position of one string, needle, in another, haystack. (See also the LASTPOS function.) If the string needle is not found, 0 is returned. By default, the search starts at the first character of haystack (that is, the value of start is 1). You can override this by specifying start (which must be a positive whole number) as the point at which the search starts.

Here are some examples:

```
POS('day','Saturday')        ->    6
POS('x','abc def ghi')       ->    0
POS(' ','abc def ghi')       ->    4
POS(' ','abc def ghi',5)     ->    8
```

8.47 QUEUED

QUEUED()

QUEUED returns the number of lines remaining in the currently active REXX data queue at the time the function is invoked.

Here is an example:

```
QUEUED()     ->     5     /* Perhaps */
```

8.48 RANDOM

RANDOM([max])

or

RANDOM([min, [,][max][,seed]])

RANDOM returns a quasi-random, nonnegative whole number in the range min to max inclusive. If only one argument is specified, the range will be from 0 to that number. Otherwise, the default values for min and max are 0 and 999, respectively. A specific seed (which must be a whole number) for the random number can be specified as the third argument if repeatable results are desired.

The magnitude of the range (that is, max minus min) must not exceed 100000.

Here are some examples:

```
RANDOM()          ->    305
RANDOM(5,8)       ->      7
RANDOM(,,1983)    ->    123  /* reproducible */
RANDOM(2)         ->      0
```

Note 1: To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a seed only the first time. For example, to simulate 40 throws of a six-sided, unbiased die, use:

```
sequence = RANDOM(1,6,12345)  /* any number would */
  /* do for a seed    */
do 39
   sequence = sequence RANDOM(1,6)
   end
say sequence
```

The numbers are generated mathematically, using the initial seed, so that as far as possible they appear to be random. Running the program again will produce the same sequence; using a different initial seed almost certainly produces a different sequence.

Note 2: The random number generator is global for an entire program; the current seed is not saved across internal routine calls.

Note 3: The actual random number generator used may differ from implementation to implementation.

8.49 REVERSE

REVERSE(string)

REVERSE returns string, swapped end for end.

Here are some examples:

```
REVERSE('ABc.')    ->    '.cBA'
REVERSE('XYZ ')    ->    ' ZYX'
```

8.50 RIGHT

RIGHT(string,length [,pad])

RIGHT returns a string of length length containing the rightmost length characters of string. The string returned is padded with pad characters (or truncated) on the left as needed. The default pad character is a blank. length must be nonnegative.

Here are some examples:

```
RIGHT('abc  d',8)     ->    '  abc  d'
RIGHT('abc def',5)    ->    'c def'
RIGHT('12',5,'0')     ->    '00012'
```

8.51 SETLOCAL

SETLOCAL()

SETLOCAL saves the current working drive and directory, and the current values of the environment variables that are local to the current process.

For example, SETLOCAL can be used to save the current environment before changing selected settings with the VALUE function. To restore the drive, directory, and environment, use the ENDLOCAL function.

SETLOCAL returns a value of 1 if the initial drive, directory, and environment are successfully saved, and a value of 0 if unsuccessful. If SETLOCAL is not followed by an ENDLOCAL function in a procedure, then the initial environment saved by SETLOCAL will be restored upon exiting the procedure.

Here is an example:

```
/* current path is 'C:\PROJ\FILES' */
n = SETLOCAL()        /* saves all environment settings    */

/* Now use the VALUE function to change the PATH variable. */
p = VALUE('Path','C:\PROC\PROGRAMS'.'OS2ENVIRONMENT')

/* Programs in directory C:\PROC\PROGRAMS may now be run   */

n = ENDLOCAL()  /* restores initial environment (including */
                /* the changed PATH variable, which is     */
                /* once again 'C:\PROJ\FILES'              */
```

Note: Unlike their counterparts in the OS/2 Batch language (the Setlocal and Endlocal statements), the REXX SETLOCAL and ENDLOCAL functions can be nested.

8.52 SIGN

SIGN(number)

SIGN returns a number that indicates the sign of number. number is first rounded according to standard REXX rules, just as though the operation number+0 had been carried out. If number is less than 0, then -1 is returned; if it is 0 then 0 is returned; and if it is greater than 0, 1 is returned.

Here are some examples:

```
SIGN('12.3')        ->     1
SIGN(' -0.307')     ->    -1
SIGN(0.0)           ->     0
```

8.53 SOURCELINE

SOURCELINE([n])

SOURCELINE returns the line number of the final line in the source file if you omit n, or returns the nth line in the source file if you specify n.

If specified, n must be a positive whole number, and must not exceed the number of the final line in the source file.

Here are some examples:

```
SOURCELINE()    ->   10
SOURCELINE(1)   ->   '/* This is a 10-line program */'
```

8.54 SPACE

SPACE(string [, [n] [,pad]])

SPACE formats the blank-delimited words in string with n pad characters between each word. The n must be nonnegative. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for n is 1, and the default pad character is a blank.

Here are some examples:

```
SPACE('abc  def  ')          ->    'abc def'
SPACE('  abc def',3)         ->    'abc   def'
SPACE('abc  def  ',1)        ->    'abc def'
SPACE('abc  def  ',0)        ->    'abcdef'
SPACE('abc  def  ',2,'+')    ->    'abc++def'
```

8.55 STREAM

STREAM(name[,[C,streamcommand] | [D] | [S]])

STREAM returns a string describing the state of, or the result of an operation upon, the character stream name. This function is used to request information on the state of an input or output stream, or to carry out some specific operation on the stream.

The first argument, name, specifies the stream to be accessed. The second argument can be one of the following strings (of which only the first letter is needed) which describes the action to be carried out:

Command	An operation (specified by the streamcommand given as the third argument) to be applied to the selected input or output stream. The string that is returned depends on the command performed, and can be the null string.
Description	Also returns the current state of the specified stream. It is identical to the State operation, except that the returned string is followed by a colon and, if available, additional information about ERROR or NOTREADY states.
State	Returns a string that indicates the current state of the specified stream. This is the default operation.

When used with the State option, STREAM returns one of the following strings:

'ERROR'	The stream has been subject to an erroneous operation (possibly during input, output, or through the STREAM function. Additional information about the error may be available by invoking the STREAM function with a request for the implementation-dependent description.
'NOTREADY'	The stream is known to be in a state such that normal input or output operations attempted upon it would raise the NOTREADY condition. For example, a simple input stream may have a defined length; an attempt to read that stream (with the CHARIN or LINEIN built-in functions, perhaps) beyond that limit may make the stream unavailable until the stream has been closed, for example, with LINEIN(name), and then reopened.
'READY'	The stream is known to be in a state such that normal input or output operations can be attempted. This is the usual state for a stream, though it does not guarantee that any particular operation will succeed.
'UNKNOWN'	The state of the stream is unknown. This response can be used to indicate that the state of the stream cannot be determined.

Note: The state (and operation) of an input or output stream is global to a REXX program, in that it is not saved and restored across function and subroutine calls (including those caused by a CALL ON condition trap).

Stream Commands

The following stream commands are used to:

- Open a stream for reading or writing
- Close a stream at the end of an operation
- Position the read or write position within a persistent stream (for example, a file)
- Get information about a stream (its existence, size, and last edit date).

The streamcommand argument must be used when you select the operation C (command). The syntax is:

STREAM(name,'C',streamcommand)

In this form, the STREAM function itself returns a string corresponding to the given streamcommand if the command is successful. If the command is unsuccessful, STREAM returns an error message string in the same form as that supplied by the D (Description) operation.

Command strings - The argument streamcommand can be any expression that REXX evaluates as one of the following command strings:

'OPEN' Opens the named stream. The default for 'OPEN' is to open the stream for both reading and writing data. To specify whether name is only to be read or only to be written to, add the word READ or WRITE to the command string.

The STREAM function itself returns 'READY' if the named stream is successfully opened or an appropriate error message if unsuccessful.

```
stream(strout,'c','open')
stream(strout,'c','open write')
stream(strinp,'c','open read')
```

'CLOSE' Closes the named stream. The STREAM function itself returns 'READY' if the named stream is successfully closed or an appropriate error message otherwise.

If an attempt is made to close an unopened file, then STREAM() returns a null string ("").

```
stream('STRM.TXT','c','close')
```

'SEEK' offset Sets the read or write position a given number (offset) within a persistent stream.

Note: In environments in which the read and write positions are the same. To use this command, the named stream must first be opened (with the 'OPEN' stream command, described previously).

The offset number can be preceded by one of the following characters:

= Explicitly specifies the offset from the beginning of the stream. This is the default if no prefix is supplied.
< Specifies offset from the end of the stream.
+ Specifies offset forward from the current read or write position.
- Specifies offset backward from the current read or write position.

The STREAM function itself returns the new position in the stream if the read or write position is successfully located; an appropriate error message is displayed otherwise

```
stream(name,'c','seek =2')
stream(name,'c','seek +15')
stream(name,'c','seek -7')
fromend  = 125
stream(name,'c','seek <'fromend)
```

Used with these stream commands, the STREAM function returns specific information about a stream.

'QUERY EXISTS' Returns the full path specification of the named stream, if it exists, and a null string otherwise.

```
stream('..\file.txt','c','query
exists')
```

'QUERY SIZE' Returns the size in bytes of a persistent stream.

```
stream('..\file.txt','c','query
size')
```

'QUERY DATETIME' Returns the date and time stamps of a stream.

```
stream('..\file.txt','c','query
datetime')
```

8.56 STRIP

STRIP(string [, [option] [,char]])

STRIP removes leading and trailing characters from string based on the option you specify. Valid options (of which only the capitalized letter is significant, all others are ignored) are:

Both	Removes both leading and trailing characters from string. This is the default.
Leading	Removes leading characters from string.
Trailing	Removes trailing characters from string.

The third argument, char, specifies the character to remove; the default is a blank. If you specify char, it must be exactly one character long.

Here are some examples:

```
STRIP('  ab c  ')        ->    'ab c'
STRIP('  ab c  ','L')    ->    'ab c  '
STRIP('  ab c  ','t')    ->    '  ab c'
STRIP('12.7000',,0)      ->    '12.7'
STRIP('0012.700',,0)     ->    '12.7'
```

8.57 SUBSTR

SUBSTR(string,n [, [length] [,pad]])

SUBSTR returns the substring of string that begins at the nth character, and is of length length and padded with pad if necessary. n must be a positive whole number.

If length is omitted, the rest of the string will be returned. The default pad character is a blank.

Here are some examples:

```
SUBSTR('abc',2)          ->    'bc'
SUBSTR('abc',2,4)        ->    'bc  '
SUBSTR('abc',2,6,'.')    ->    'bc....'
```

Note: In some situations the positional (numeric) patterns of parsing templates are more convenient for selecting substrings, especially if more than one substring is to be extracted from a string.

8.58 SUBWORD

SUBWORD(string,n [,length])

SUBWORD returns the substring of string that starts at the nth word, and is of length length, blank-delimited words. n must be a positive whole number. If you omit length, it defaults to the number of remaining words in string. The returned string never has leading or trailing blanks, but includes all blanks between the selected words.

Here are some examples:

```
SUBWORD('Now is the  time',2,2)    ->    'is the'
SUBWORD('Now is the  time',3)      ->    'the  time'
SUBWORD('Now is the  time',5)      ->    ''
```

8.59 SYMBOL

SYMBOL(name)

SYMBOL returns the state of the symbol named by name. If name is not a valid REXX symbol, BAD is returned. SYMBOL returns VAR if it is the name of a variable (that is, a symbol that has been assigned a value). Otherwise, SYMBOL returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

As with symbols in REXX expressions, lowercase characters in name are translated to uppercase and substitution in a compound name occurs if possible.

Note: You should specify name as a literal string (or derived from an expression) to prevent substitution before it is passed to the function.

Here are some examples:

```
/* following: Drop A.3;  J=3 */
SYMBOL('J')      ->   'VAR'
SYMBOL(J)        ->   'LIT' /* has tested "3"     */
SYMBOL('a.j')    ->   'LIT' /* has tested "A.3"   */
SYMBOL(2)        ->   'LIT' /* a constant symbol  */
SYMBOL('*')      ->   'BAD' /* not a valid symbol */
```

8.60 TIME

TIME([option])

TIME returns the local time in the 24-hour clock format hh:mm:ss (hours, minutes, and seconds) by default; for example:

04:41:37

You can use the following options (for which only the capitalized letter is needed) to obtain alternative formats, or to gain access to the elapsed-time clock:

Civil Returns hh:mmxx, the time in Civil format, in which the hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters "am" or "pm" to distinguish times in the morning (midnight 12:00am through 11:59am) from noon and afternoon (noon 12:00pm through 11:59pm). The hour will not have a leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

Elapsed Returns ssssssss.uu0000, the number of seconds.hundredths since the elapsed time clock was started or reset (see below). The returned number has no leading zeros, but always has four trailing zeros in the decimal portion. It is not affected by the setting of NUMERIC DIGITS.

Hours Returns number of hours since midnight in the format hh (no leading zeros).

Long Returns time in the format hh:mm:ss.uu0000 (where uu is the fraction of seconds in hundredths of a second).

Minutes Returns number of minutes since midnight in the format: mmmm (no leading zeros).

Normal Returns the time in the default format hh:mm:ss, as described above.

Reset Returns sssssssss.uu0000, the number of seconds.hundredths since the elapsed time clock was started or reset (see below) and also resets the elapsed-time clock to zero. The returned number has no leading zeros, but always has four trailing zeros in the decimal portion.

Seconds Returns number of seconds since midnight in the format sssss (no leading zeros).

Here are some examples:

```
TIME('L')    ->   '16:54:22.120000'   /* Perhaps */
TIME()       ->   '16:54:22'
TIME('H')    ->   '16'
TIME('M')    ->   '1014'/* 54 + 60*16 */
TIME('S')    ->   '60862'  /* 22 + 60*(54+60*16) */
TIME('N')    ->   '16:54:22'
TIME('C')    ->   '4:54pm'
```

The Elapsed-Time Clock

The elapsed-time clock may be used for measuring real time intervals. On the first call to the elapsed-time clock, the clock is started, and both TIME('E') and TIME('R') will return 0.

The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected even if an internal routine resets the clock.

Here is an example of the elapsed-time clock:

```
time('E')    ->    0          /* The first call */
/* pause of one second here */
time('E')    ->    1.020000   /* or thereabouts */
/* pause of one second here */
time('R')    ->    2.030000   /* or thereabouts */
/* pause of one second here */
time('R')    ->    1.050000   /* or thereabouts */
```

Note: See the DATE function about consistency of times within a single expression. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single expression always return the same result. For the

same reason, the interval between two normal TIME and DATE results may be calculated exactly using the elapsed-time clock.

Implementation maximum: If the number of seconds in the elapsed time exceed nine digits (equivalent to over 31.6 years), an error will result.

8.61 TRACE

TRACE([option])

TRACE returns trace actions currently in effect.

If option is supplied, it must be the valid prefix (?), one of the alphabetic character options (that is, starting with A, C, E, F, I, L, N, O, or R) associated with the TRACE instruction, or both. The function uses option to alter the effective trace action (such as tracing labels). Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active.

Unlike the TRACE instruction, option cannot be a number.

Here are some examples:

```
TRACE()       ->   '?R' /* maybe */
TRACE('O')    ->   '?R' /* also sets tracing off    */
TRACE('?I')   ->   'O'  /* now in interactive debug */
```

8.62 TRANSLATE

TRANSLATE(string [, [tableo] [, [tablei] [,pad]]])

TRANSLATE translates characters in string to other characters, or reorders characters in a string. If neither translate table is given, string is simply translated to uppercase (for example, a lowercase a-z to an uppercase A-Z). The output table is tableo and the input translate table is tablei (the default is XRANGE('00'x,'FF'x)). The output table defaults to the null string and is padded with pad or truncated as necessary. The default pad is a blank. The tables can be of any length; the first occurrence of a character in the input table is the one that is used if there are duplicates.

Here are some examples:

```
TRANSLATE('abcdef')                      ->    'ABCDEF'
TRANSLATE('abbc','&','b')                ->    'a&&c'
TRANSLATE('abcdef','12','ec')            ->    'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')      ->    '12..ef'
TRANSLATE('4123','abcd','1234')          ->    'dabc'
```

Note: The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

8.63 TRUNC

TRUNC(number [,n])

TRUNC returns the integer part of number, and n decimal places. The default n is zero, and it returns an integer with no decimal point. If you specify n, it must be a nonnegative whole number. The number is first rounded according to standard REXX rules, just as though the operation number+0 had been carried out. The number is then truncated to n decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

Here are some examples:

```
TRUNC(12.3)->    12
TRUNC(127.09782,3)     ->    127.097
TRUNC(127.1,3)         ->    127.100
TRUNC(127,2)           ->    127.00
```

Note: The number is rounded according to the current setting of NUMERIC DIGITS if necessary before being processed by the function.

8.64 VALUE

VALUE(name [, [newvalue] [,selector]])

VALUE returns the value of the symbol named by name, and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment, but other, external collections of variables may be selected. If you use the function to refer to REXX variables, then name must be a valid REXX symbol. (You can confirm this by using the SYMBOL function.) Lowercase characters in name are translated to uppercase. If name is a compound symbol, then REXX substitutes symbol values to produce the derived name of the symbol.

If you specify newvalue, then the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of name as it was before the new assignment.

Here are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)      ->  'A3'
VALUE('a'k||k)   ->  '7'
VALUE('fred')    ->  'K'  /* looks up FRED   */
VALUE(fred)      ->  '3'  /* looks up K      */
VALUE(fred,5)    ->  '3'  /* and sets K=5    */
VALUE(fred)      ->  '5'
VALUE('LIST.'k) ->  'Hi' /* looks up LIST.5 */
```

Notes: If the VALUE function refers to an uninitialized REXX variable, then the default value of the variable is always returned; the NOVALUE condition is not raised. NOVALUE is never raised by a reference to an external collection of variables.

The VALUE function is used when a variable contains the name of another variable, or when a name is constructed dynamically. If the name is specified as a single literal string, the symbol is a constant and so the whole function call can usually be replaced directly by the string between the quotation marks. (For example, fred=VALUE('k'); is identical to the assignment fred=k;, unless the NOVALUE condition is being trapped.

To effect temporary changes to environment variables, use the SETLOCAL and ENDLOCAL functions.

8.65 VERIFY

VERIFY(string,reference [, [option] [,start]])

VERIFY returns a number indicating whether string is composed only of characters from reference. VERIFY returns the position of the first character in string that is not also in reference. If all the characters were found in reference, 0 is returned.

The third argument, option, can be any expression that results in a string starting with N or M that represents either Nomatch (the default) or Match.. Only the first character of option is significant and it can be in uppercase or lowercase, as usual. If Nomatch is specified, the position of the first character in string that is not also in reference is returned. 0 is returned if all characters in string were found in reference. If Match is specified, the position of the first character in string that is in reference is returned, or 0 is returned if none of the characters were found.

The default for start is 1, thus, the search starts at the first character of string. You can override this by specifying a different start point, which must be a positive whole number.

VERIFY always returns 0 if string is null or if start is greater than LENGTH(string). If reference is null, VERIFY returns 0 if you specify Match; otherwise, 1 is returned.

Here are some examples:

```
VERIFY('123','1234567890')              ->    0
VERIFY('1Z3','1234567890')              ->    2
VERIFY('AB4T','1234567890')             ->    1
VERIFY('AB4T','1234567890','M')         ->    3
VERIFY('AB4T','1234567890','N')         ->    1
VERIFY('1P3Q4','1234567890',,3)         ->    4
VERIFY('AB3CD5','1234567890','M',4)     ->    6
```

8.66 WORD

WORD(string,n)

WORD returns the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in string, the null string is returned. This function is equivalent to SUBWORD(string,n,1).

Here are some examples:

```
WORD('Now is the time',3)    ->    'the'
WORD('Now is the time',5)    ->    ''
```

8.67 WORDINDEX

WORDINDEX(string,n)

WORDINDEX returns the position of the first character in the nth blank-delimited word in string. n must be a positive whole number. If there are fewer than n words in the string, 0 is returned.

Here are some examples:

```
WORDINDEX('Now is the time',3)    ->    8
WORDINDEX('Now is the time',6)    ->    0
```

8.68 WORDLENGTH

WORDLENGTH(string,n)

WORDLENGTH returns the length of the nth blank-delimited word in string. n must be a positive whole number. If here are fewer than n words in the string, 0 is returned.

Here are some examples:

```
WORDLENGTH('Now is the time',2)       ->    2
WORDLENGTH('Now comes the time',2)    ->    5
WORDLENGTH('Now is the time',6)       ->    0
```

8.69 WORDPOS

WORDPOS(phrase,string [,start])

WORDPOS searches string for the first occurrence of the sequence of blank-delimited words phrase, and returns the word number of the first word of phrase in string. Multiple blanks between words in either phrase or string are treated as a single blank for the comparison, but otherwise the words must match exactly.

By default, the search starts at the first word in string. You can override this by specifying start (which must be positive), the word at which to start the search.

Here are some examples:

```
WORDPOS('the','now is the time')              ->  3
WORDPOS('The','now is the time')              ->  0
WORDPOS('is the','now is the time')           ->  2
WORDPOS('is   the','now is the time')         ->  2
WORDPOS('is   time ','now is   the time')     ->  0
WORDPOS('be','To be or not to be')            ->  2
WORDPOS('be','To be or not to be',3)          ->  6
```

8.70 WORDS

WORDS(string)

WORDS returns the number of blank-delimited words in string.

Here are some examples:

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

8.71 XRANGE

XRANGE([start] [,end])

XRANGE returns a string of all one-byte codes between and including the values start and end. The default value for start is `00'x, and the default value for end is `FF'x. If start is greater than end, the values wrap from `FF'x to`00'x. If specified, start and end must be single characters.

Here are some examples:

```
XRANGE('a','f')       ->   'abcdef'
XRANGE('03'x,'07'x)   ->   '0304050607'x
XRANGE(,'04'x)        ->   '0001020304'x
XRANGE('i','j')       ->   '898A8B8C8D8E8F9091'x  /* EBCDIC */
XRANGE('FE'x,'02'x)   ->   'FEFF000102'x
XRANGE('i','j')       ->   'ij'                   /* ASCII  */
```

8.72 X2B (Hexadecimal to Binary)

X2B(hexstring)

X2B converts hexstring (a string of hexadecimal characters) to an equivalent string of binary digits. hexstring can be of any length; each hexidecimal character is converted to a string of four binary digits. The returned string has a length that is a multiple of four, and does not include any blanks.

Blanks can optionally be added (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If hexstring is null, X2B returns 0.

Here are some examples:

```
X2B('C3')         ==  '11000011'
X2B('7')          ==  '0111'
X2B('1 C1')       ==  '000111000001'
```

You can combine X2B() may be combined with the functions D2X() and C2X() to convert decimal numbers or character strings into binary form.

Here are some examples:

```
X2B(C2X('C3'x))   ==  '11000011'
X2B(D2X('129'))   ==  '10000001'
X2B(D2X('12'))    ==  '1100'
```

8.73 X2C (Hexadecimal to Character)

X2C(hexstring)

X2C converts hexstring (a string of hexadecimal characters) to character.

hexstring can be of any length. You can optionally add blanks to hexstring (at byte boundaries only, not leading or trailing positions) to aid readability; they are ignored.

If necessary, hexstring is padded with a leading 0 to make an even number of hexadecimal digits.

Here are some examples:

```
X2C('4865 6c6c 6f') ->  'Hello'     /*  ASCII   */
X2C('3732 73')      ->  '72s'       /*  ASCII   */
X2C('F')            ->  '0F'x
```

8.74 X2D (Hexadecimal to Decimal)

X2D(hexstring [,n])

X2D converts hexstring (a string of hexadecimal characters) to decimal. If the result cannot be expressed as a whole number, an error results. That is, the result must have no more digits than the current setting of NUMERIC DIGITS.

You can optionally add blanks to hexstring (at byte boundaries only, not leading or trailing positions) to aid readability; they are ignored.

If hexstring is null, X2D returns 0.

If n is not specified, hexstring is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')         ->    14
X2D('81')         ->    129
X2D('F81')        ->    3969
X2D('FF81')       ->    65409
X2D('c6 f0'X)     ->    240
```

If n is specified, the given sequence of hexadecimal digits is padded on the left with zeros (note, not sign-extended), or truncated on the left to n characters. The resulting string of n hexadecimal digits is taken to be a signed binary number-positive if the leftmost bit is OFF, and negative, in two's complement notation, if the leftmost bit is ON. If n is 0, X2D returns 0.

Here are some examples:

```
X2D('81',2)      ->    -127
X2D('81',4)      ->    129
X2D('F081',4)    ->    -3967
X2D('F081',3)    ->    129
X2D('F081',2)    ->    -127
X2D('F081',1)    ->    1
X2D('0031',0)    ->    0
```