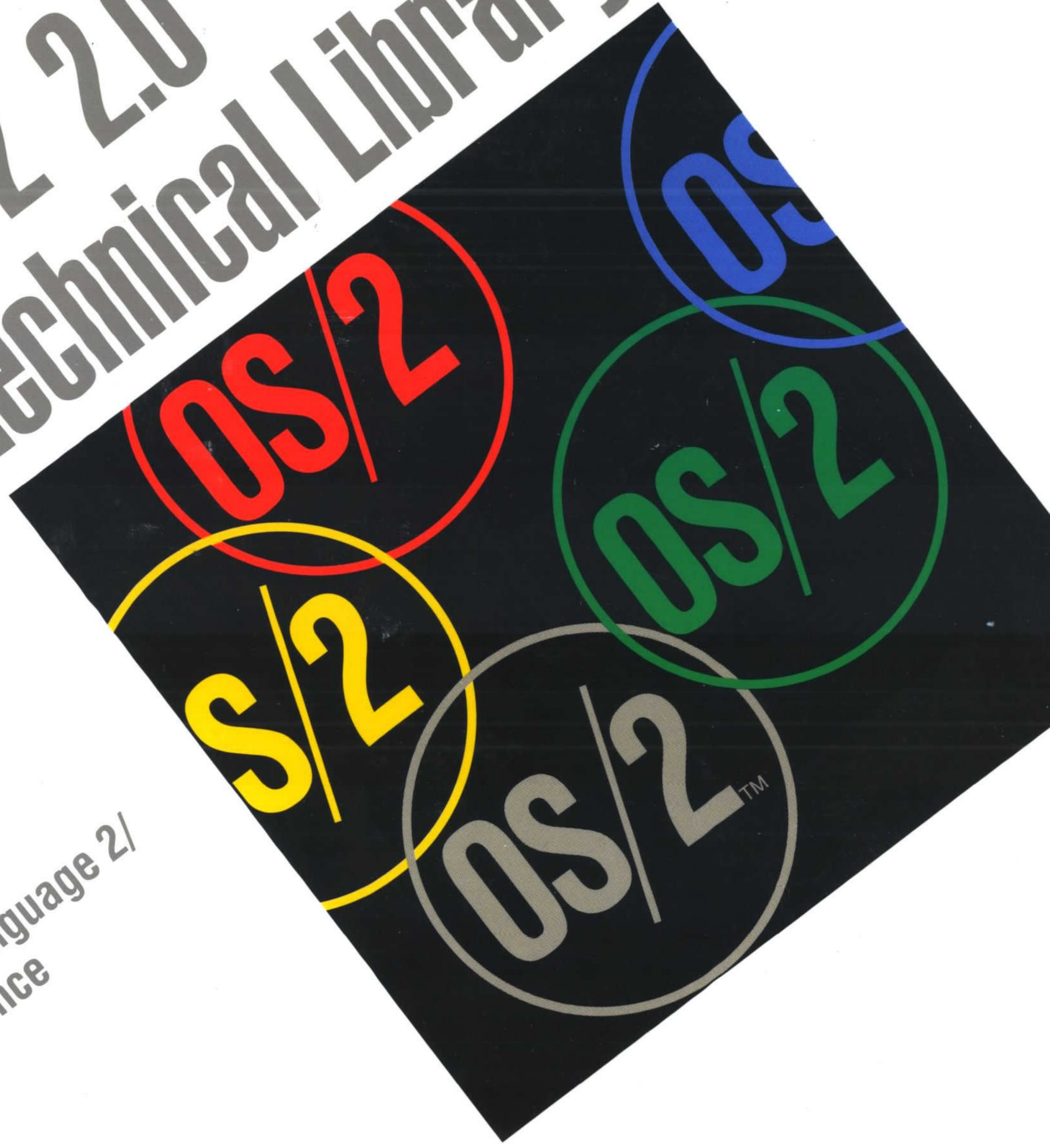
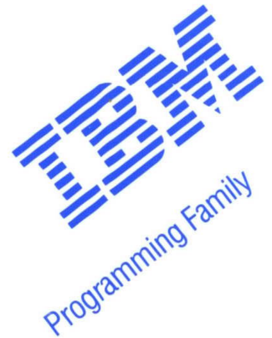
OS/2 2.0 Technical Library



Procedures Language 2/ REXX Reference

Version 2.00

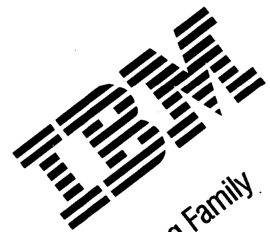


Programming Family

OS/2 2.0 Technical Library

Procedures Language 2/ REXX Reference

Version 2.00



Programming Family

Note

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xi.

First Edition (December 1991)

The following paragraph does not apply to the United Kingdom or any country where such provisions are inconsistent with local law: INTERNATIONAL BUSINESS MACHINES CORPORATION PROVIDES THIS PUBLICATION "AS IS" WITHOUT WARRANTY OF ANY KIND, EITHER EXPRESS OR IMPLIED, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE. Some states do not allow disclaimer of express or implied warranties in certain transactions, therefore, this statement may not apply to you.

This publication could include technical inaccuracies or typographical errors. Changes are periodically made to the information herein; these changes will be incorporated in new editions of the publication. IBM may make improvements and/or changes in the product(s) and/or the program(s) described in this publication at any time.

It is possible that this publication may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Requests for technical information about IBM products should be made to your IBM Authorized Dealer or your IBM Marketing Representative.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

COPYRIGHT LICENSE: This publication contains printed sample application programs in source language, which illustrate OS/2 programming techniques. You may copy and distribute these sample programs in any form without payment to IBM, for the purposes of developing, using, marketing or distributing application programs conforming to the OS/2 application programming interface.

Each copy of any portion of these sample programs or any derivative work, which is distributed to others, must include a copyright notice as follows: "© (your company name) (year) All Rights Reserved."

Contents

Chapter 1. Introduction . . . 1-1
What the SAA Solution Is . . . 1-1
Supported Environments . . . 1-1
Common Programming Interface . . . 1-2
How to Read the Syntax Diagrams . . . 1-2

Chapter 2. General Concepts . . . 2-1
Brief Description of the REstructured eXtended eXecutor Language . . . 2-1
REXX and the OS/2 Operating System . . . 2-1
Structure and General Syntax . . . 2-2
Characters . . . 2-2
Tokens . . . 2-2
Implied Semicolons . . . 2-7
Continuations . . . 2-7
Expressions and Operators . . . 2-7
Expressions . . . 2-7
Operators . . . 2-8
String Concatenation . . . 2-8
Arithmetic . . . 2-9
Comparison . . . 2-9
Logical (Boolean) . . . 2-10
Parentheses and Operator Precedence . . . 2-11
Clauses and Instructions . . . 2-12
Null Clauses . . . 2-12
Labels . . . 2-12
Instructions . . . 2-13
Assignments . . . 2-13
Keyword Instructions . . . 2-13
Commands . . . 2-13
Assignments and Symbols . . . 2-13
Constant Symbols . . . 2-14
Simple Symbols . . . 2-14
Compound Symbols . . . 2-14
Stems . . . 2-16
Commands to External Environments . . . 2-17
Environment . . . 2-17
Commands . . . 2-17
Using REXX on the OS/2 Operating System . . . 2-18
Calling Another REXX Program . . . 2-18
Generating a Return Code . . . 2-18
Running a REXX Program . . . 2-18
Editing Commands . . . 2-19

Chapter 3. Keyword Instructions . . . 3-1
ADDRESS . . . 3-2
ARG . . . 3-4
CALL . . . 3-6
DO . . . 3-9
Simple DO Group . . . 3-9
Simple Repetitive Loops . . . 3-10
Controlled Repetitive Loops . . . 3-10
Conditional Phrases (WHILE and UNTIL) . . . 3-12

DROP . . . 3-14
EXIT . . . 3-15
IF . . . 3-16
INTERPRET . . . 3-17
ITERATE . . . 3-19
LEAVE . . . 3-20
NOP . . . 3-21
NUMERIC . . . 3-22
OPTIONS . . . 3-24
PARSE . . . 3-25
PROCEDURE . . . 3-27
PULL . . . 3-29
PUSH . . . 3-30
QUEUE . . . 3-31
RETURN . . . 3-32
SAY . . . 3-33
SELECT . . . 3-34
SIGNAL . . . 3-35
TRACE . . . 3-37
Alphabetic Character (Word) Options . . . 3-37
Prefix Option . . . 3-38
Numeric Options . . . 3-39
Tracing Tips . . . 3-39
A Typical Example . . . 3-39
Format of TRACE Output . . . 3-39

Chapter 4. Functions . . . 4-1
Syntax . . . 4-1
Calls to Functions and Subroutines . . . 4-1
Search Order . . . 4-2
Errors During Execution . . . 4-4
Return Values . . . 4-5
Built-in Functions . . . 4-5
ABBREV (Abbreviation) . . . 4-6
ABS (Absolute Value) . . . 4-7
ADDRESS . . . 4-7
ARG (Argument) . . . 4-7
BEEP . . . 4-8
BITAND (Bit by Bit AND) . . . 4-9
BITOR (Bit by Bit OR) . . . 4-9
BITXOR (Bit by Bit Exclusive OR) . . . 4-10
B2X (Binary to Hexadecimal) . . . 4-10
CENTER/CENTRE . . . 4-11
CHARIN (Character Input) . . . 4-11
CHAROUT (Character Output) . . . 4-12
CHARS (Characters Remaining) . . . 4-13
COMPARE . . . 4-13
CONDITION . . . 4-14
COPIES . . . 4-15
C2D (Character to Decimal) . . . 4-15
C2X (Character to Hexadecimal) . . . 4-16
DATATYPE . . . 4-16
DATE . . . 4-17
DBCS (Double-Byte Character Set Functions) . . . 4-18
DELSTR (Delete String) . . . 4-19
DELWORD (Delete Word) . . . 4-19

DIGITS . . . 4-19
DIRECTORY . . . 4-19
D2C (Decimal to Character) . . . 4-20
D2X (Decimal to Hexadecimal) . . . 4-21
ENDLOCAL . . . 4-21
ERRORTEXT . . . 4-22
FILESPEC . . . 4-22
FORM . . . 4-23
FORMAT . . . 4-23
FUZZ . . . 4-24
INSERT . . . 4-24
LASTPOS (Last Position) . . . 4-24
LEFT . . . 4-25
LENGTH . . . 4-25
LINEIN (Line Input) . . . 4-25
LINEOUT (Line Output) . . . 4-27
LINES (Lines Remaining) . . . 4-28
MAX (Maximum) . . . 4-29
MIN (Minimum) . . . 4-29
OVERLAY . . . 4-29
POS (Position) . . . 4-30
QUEUED . . . 4-30
RANDOM . . . 4-30
REVERSE . . . 4-31
RIGHT . . . 4-31
SETLOCAL . . . 4-32
SIGN . . . 4-32
SOURCELINE . . . 4-33
SPACE . . . 4-33
STREAM . . . 4-33
Stream Commands . . . 4-34
STRIP . . . 4-36
SUBSTR (Substring) . . . 4-37
SUBWORD . . . 4-37
SYMBOL . . . 4-37
TIME . . . 4-38
TRACE . . . 4-40
TRANSLATE . . . 4-40
TRUNC (Truncate) . . . 4-41
VALUE . . . 4-41
VERIFY . . . 4-43
WORD . . . 4-43
WORDINDEX . . . 4-43
WORDLENGTH . . . 4-44
WORDPOS (Word Position) . . . 4-44
WORDS . . . 4-44
XRANGE (Hexadecimal Range) . . . 4-45
X2B (Hexadecimal to Binary) . . . 4-45
X2C (Hexadecimal to Character) . . . 4-45
X2D (Hexadecimal to Decimal) . . . 4-46
OS/2 Applications Programming Interface Functions . . . 4-47
RXFUNCADD . . . 4-47
RXFUNCDROP . . . 4-47
RXFUNCQUERY . . . 4-47
Queue Interface . . . 4-47
RXQUEUE . . . 4-47

REXX Utilities (RexxUtil) . . . 4-49
RxMessageBox . . . 4-49
SysCls . . . 4-51
SysCreateObject . . . 4-51
SysCurPos . . . 4-51
SysCurState . . . 4-52
SysDeregisterObjectClass . . . 4-52
SysDriveInfo . . . 4-53
SysDriveMap . . . 4-53
SysDropFuncs . . . 4-54
SysFileDelete . . . 4-54
SysFileTree . . . 4-55
SysFileSearch . . . 4-56
SysGetEA . . . 4-58
SysGetKey . . . 4-58
SysGetMessage . . . 4-58
SysIni . . . 4-59
SysMkDir . . . 4-61
SysOS2Ver . . . 4-62
SysPutEA . . . 4-62
SysQueryClassList . . . 4-62
SysRegisterObjectClass . . . 4-63
SysRmDir . . . 4-63
SysSearchPath . . . 4-64
SysSetIcon . . . 4-64
SysSleep . . . 4-65
SysTempFileName . . . 4-65
SysTextScreenRead . . . 4-66
SysTextScreenSize . . . 4-66
SysWaitNamedPipe . . . 4-66

Chapter 5. Parsing . . . 5-1
Simple Templates for Parsing into Words . . . 5-1
The Period as a Placeholder . . . 5-2
Templates Containing String Patterns . . . 5-3
Templates Containing Positional (Numeric) Patterns . . . 5-4
Combining Patterns and Parsing Into Words . . . 5-6
Parsing with Variable Patterns . . . 5-7
Using UPPER . . . 5-8
Parsing Instructions Summary . . . 5-8
Parsing Instructions Examples . . . 5-9
Advanced Topics in Parsing . . . 5-10
Parsing Multiple Strings . . . 5-10
Combining String and Positional Patterns: A Special Case . . . 5-11
Parsing with DBCS Characters . . . 5-11
Details of Steps in Parsing . . . 5-11

Chapter 6. Numbers and Arithmetic . . . 6-1
Introduction . . . 6-1
Definition . . . 6-2
Numbers . . . 6-2
Precision . . . 6-2
Arithmetic Operators . . . 6-3
Arithmetic Operation Rules—Basic Operators . . . 6-3
Addition and Subtraction . . . 6-4
Multiplication . . . 6-4

Division 6-4
Basic Operator Examples 6-5
Arithmetic Operation Rules—Additional Operators 6-5
Power 6-5
Integer Division 6-5
Remainder 6-6
Additional Operator Examples 6-6
Numeric Comparisons 6-6
Exponential Notation 6-7
Whole Numbers 6-9
Numbers Used Directly by REXX 6-9
Errors 6-9

Chapter 7. Conditions and Condition Traps 7-1
Action Taken When a Condition Is Not Trapped 7-2
Action Taken When a Condition Is Trapped 7-2
Condition Information 7-4
The Special Variable RC 7-5
The Special Variable SIGL 7-5

Chapter 8. Input and Output Streams 8-1
The Input and Output Model 8-2
Character Input Streams 8-2
Character Output Streams 8-3
The STREAM Function 8-3
External Data Queue 8-3
Implementation 8-4
Queue Interface 8-4
Access to Queues 8-5
Session Queues 8-5
Private Queues 8-5
Special Considerations 8-5
Detached Processes 8-6
Multi-Programming Considerations 8-6
Errors During Input and Output 8-6
Examples of Input and Output 8-8
Summary of Instructions and Functions 8-9

Chapter 9. Application Programming Interface 9-1
General Characteristics 9-2
RXSTRINGs 9-3
Invoking the REXX Interpreter 9-4
From the OS/2 operating system 9-4
From Within an Application 9-4
The RexxStart Function 9-5
RexxStart 9-5
Subcommand Interfaces 9-9
Registering Subcommand Handlers 9-9
Creating Subcommand Handlers 9-9
Subcommand Interface Functions 9-12
RexxRegisterSubcomDll 9-12
RexxRegisterSubcomExe 9-13
RexxDeregisterSubcom 9-14
RexxQuerySubcom 9-15
Return Codes 9-17
External Functions 9-18

Registering External Functions 9-18
Creating External Functions 9-18
Calling External Functions 9-19
External Function Interface Functions 9-20
RexxRegisterFunctionDll 9-20
RexxRegisterFunctionExe 9-22
RexxDeregisterFunction 9-23
RexxQueryFunction 9-24
Return Codes 9-25
System Exits 9-26
Writing System Exit Handlers 9-26
Exit Return Codes 9-26
Exit Parameters 9-27
Identifying Exit Handlers to REXX 9-27
System Exit Definitions 9-29
System Exit Details 9-30
RXFNC 9-30
RXCMD 9-32
RXMSQ 9-33
RXSIO 9-35
RXHLT 9-37
RXTRC 9-38
RXINI 9-39
RXTER 9-39
System Exit Functions 9-40
RexxRegisterExitDll 9-40
RexxRegisterExitExe 9-41
RexxDeregisterExit 9-42
RexxQueryExit 9-43
Variable Pool Interface 9-45
RexxVariablePool Interface Function 9-45
RexxVariablePool 9-45
Interface Types 9-50
Symbolic Interface 9-50
Direct Interface 9-50
RexxVariablePool Restrictions 9-50
Halt and Trace Functions 9-52
Halt and Trace Functions 9-53
RexxSetHalt 9-53
RexxSetTrace 9-54
RexxResetTrace 9-55
Macrospace Interface 9-56
Search Order 9-56
Storage of Macrospace Libraries 9-56
Macrospace Interface Functions 9-57
RexxAddMacro 9-57
RexxDropMacro 9-58
RexxClearMacroSpace 9-59
RexxSaveMacroSpace 9-60
RexxLoadMacroSpace 9-61
RexxQueryMacro 9-62
RexxReorderMacro 9-63
Return Codes 9-64

Chapter 10. Debugging Aids 10-1
Interactive Debugging of Programs 10-1

RXTRACE Variable 10-2

Chapter 11. Reserved Keywords and Special Variables 11-1
Reserved Keywords 11-1
Special Variables 11-2

Chapter 12. Useful OS/2 Commands 12-1
CALL Command 12-1
Other OS/2 Commands 12-1
Subcommand Handler Services 12-2
The RXSUBCOM Command 12-2
RXSUBCOM REGISTER 12-2
RXSUBCOM DROP 12-3
RXSUBCOM QUERY 12-3
RXSUBCOM LOAD 12-3
Queue Services (Filters) 12-4
RXQUEUE filter 12-4

Appendix A. Error Numbers and Messages A-1

Appendix B. Double-Byte Character Set (DBCS) Support B-1
General Description B-1
Enabling DBCS Data Operations B-2
Pure DBCS Strings and Mixed SBCS/DBCS Strings B-2
Mixed String Validation B-3
Instruction Examples B-3
PARSE B-3
SAY and TRACE B-5
DBCS Function Handling B-5
Built-in Function Examples B-7
ABBREV B-7
COMPARE B-7
COPIES B-8
INSERT and OVERLAY B-8
LEFT, RIGHT, and CENTER B-8
LENGTH B-9
LINEIN B-9
REVERSE B-9
SPACE B-9
STRIP B-9
SUBSTR and DELSTR B-10
SUBWORD and DELWORD B-10
TRANSLATE B-10
VERIFY B-11
WORD, WORDINDEX, and WORDLENGTH B-11
WORDS B-12
WORDPOS B-12
DBCS Processing Functions B-12
Counting Option B-12
Function Descriptions B-12
DBADJUST B-12
DBBRACKET B-13
DBCENTER B-13
DBLEFT B-14
DBRIGHT B-14
DBRLEFT B-15

DBRRIGHT B-15
DBTODBCS B-16
DBTOSBCS B-16
DBUNBRACKET B-17
DBVALIDATE B-17
DBWIDTH B-18

Index X-1

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM's product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of IBM's intellectual property rights or other legally protectible rights may be used instead of the IBM product, program, or service. Evaluation and verification of operation in conjunction with other products, programs, or services, except those expressly designated by IBM, are the user's responsibility.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

Trademarks

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

Operating System/2	OS/2
Operating System/400	OS/400
Systems Application Architecture	SAA
Enterprise Systems Architecture/370	PS/2

Double-Byte Character Set (DBCS)

Throughout this publication, you will see references to specific values for character strings. The values are for single-byte character set (SBCS). If you use the double-byte character set (DBCS), note that one DBCS character equals two SBCS characters.

About This Book

Who Should Read This Book

This book describes the OS/2*Procedures Language processor and the REstructured eXtended eXecutor (abbreviated REXX) language. This book is intended for experienced programmers, particularly those who have used a block-structured, high-level language (for example, PL/I, Algol, C, or Pascal).

This book is a reference rather than a tutorial. It assumes you are already familiar with REXX programming concepts.

Descriptions include the use and syntax of the language and explain how the language processor "interprets" the language as a program is executing.

How to Use This Book

The material in this book is arranged in chapters:

1. Introduction
2. General Concepts
3. Keyword Instructions (in alphabetic order)
4. Functions (in alphabetic order)
5. Parsing (a method of dividing character strings, such as commands)
6. Numbers and Arithmetic
7. Conditions and Condition Traps
8. Input and Output Streams
9. Applications Programming Interface
10. Debug Aids
11. Reserved Keywords and Special Variables
12. Some Useful OS/2 Commands

There are also appendixes covering:

- Error Numbers and Messages
- Double-Byte Character Set (DBCS) Support

For Further REXX Information

Here is a list of books that you may wish to include in your REXX library:

- The *Procedures Language 2/REXX User's Guide* offers a general introduction to programming for beginners and extensive practical examples of REXX applications for OS/2 programmers of all levels.

- The *SAA* CPI Procedures Language Level 2 Reference*, may be useful to more experienced REXX users who may wish to code portable programs. This book defines the SAA Procedures Language. Descriptions include the use and syntax of the language as well as explanations on how the language processor interprets the language as a program is running.

* Trademark of IBM Corporation

Chapter 1. Introduction

This introductory section:

- Gives a brief overview of the Systems Application Architecture* (SAA) solution
- Explains the Common Programming Interface
- Explains how to read a syntax diagram.

What the SAA Solution Is

The SAA solution is based on a set of software interfaces, conventions, and protocols that provide a framework for designing and developing applications. The SAA Procedures Language has been defined as a superset of the REXX language that can be used in a number of computing environments.

If you are using REXX only in an OS/2 environment, this will have no effect on your programs. If you plan on running your programs on other environments, however, some restrictions may apply. We suggest that you consult the *SAA CPI Procedures Language Level 2 Reference*, SC24-5549.

The SAA solution:

- Defines a Common Programming Interface that you can use to develop consistent, integrated enterprise software
- Defines Common Communications Support that you can use to connect applications, systems, networks, and devices
- Defines a Common User Access architecture that you can use to achieve consistency in screen layout and user interaction techniques
- Offers some applications and application development tools written by IBM.

Supported Environments

Several combinations of IBM hardware and software have been selected as SAA environments. These are environments in which IBM will manage the availability of support for applicable SAA elements, and the conformance of those elements to SAA specifications. The SAA environments are the following:

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

* Trademark of IBM Corporation

Common Programming Interface

The Common Programming Interface (CPI) provides languages and services that programmers can use to develop applications that take advantage of SAA consistency.

The components of the interface currently fall into two general categories:

- Languages

  Application Generator
  C
  COBOL
  FORTRAN
  PL/I
  Procedures Language
  RPG*

- Services

  Communications
  Database
  Dialog
  Presentation
  PrintManager
  Query
  Repository
  Resource Recovery.

The CPI is not in itself a product or a piece of code. But—as a definition—it does establish and control how IBM products are being implemented, and it establishes a common base across the applicable SAA environments.

How to Read the Syntax Diagrams

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

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

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

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

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

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

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

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

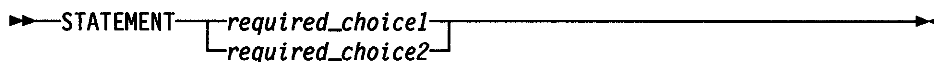
```
►►──STATEMENT──required_item──────────────────────────────────────►◄
```

- Optional items appear below the main path.

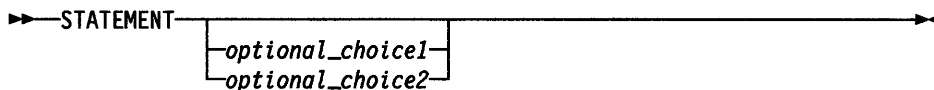
```
►►──STATEMENT──┬───────────────┬──────────────────────────────────►◄
               └─optional_item─┘
```

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

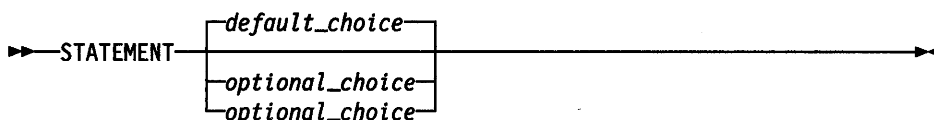
If you *must* choose one of the items, one item of the stack appears on the main path.



If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.

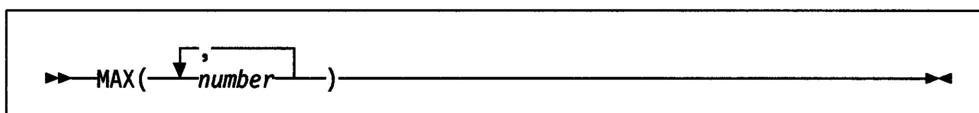


- An arrow returning to the left above the main line indicates an item that can be repeated.

A repeat arrow above a stack indicates that you can repeat the items in the stack.

- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

The following example shows how the syntax is described:



Chapter 2. General Concepts

Brief Description of the REstructured eXtended eXecutor Language

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

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

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

No restrictions are imposed by the language on program format. There can be more than one clause on a line, or a single clause can occupy more than one line. Indentation is allowed. Programs can, therefore, be coded in a format that emphasizes their structure, making them easier to read.

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

Symbols (variable names) are limited to a length of 250 characters.

You can use compound symbols, such as

```
NAME.X.Y
```

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

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

REXX and the OS/2 Operating System

REXX has been designed to be an integral part of the OS/2 operating system. When you install the operating system, you can choose if you want to install REXX, PMREXX, or REXX information. There is no installation process or explicit invocation of the language processor. REXX program files have the default extension *CMD*, and they can contain OS/2 commands as well as REXX instructions. Anywhere you can use an OS/2 command or batch-file, you can use a REXX program.

Structure and General Syntax

Programs written in the REstructured eXtended eXecutor (REXX) language must start with a comment in the first column of the first line. This distinguishes a REXX program from an OS/2 batch file.

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

- Zero or more blanks (which are ignored)
- A sequence of tokens (see "Tokens")
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:) if it follows a single symbol.

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to operator characters and special characters (see page 2-6) are also removed.

Characters

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

For information about Double-Byte Character Set characters, see Appendix B, "Double-Byte Character Set (DBCS) Support" on page B-1.

Tokens

Programs written in REXX are composed of tokens. (Tokens can be of any length, up to an implementation-restricted maximum.) They are separated by blanks or by the nature of the tokens themselves. The classes of tokens are:

Comments:

A sequence of characters (on one or more lines) delimited by /* and */. Within these delimiters any characters are allowed. Comments can contain other comments, as long as each begins and ends with the necessary delimiters. They are called **nested comments**. You can write comments anywhere in a program. The language processor ignores them (and, therefore, they can be of any length), but they do act as separators.

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

A comment can contain any characters, and the only significant characters within a comment are the /* and the */ (comment start and end delimiters). This means you must take special care when commenting out lines of code containing /* or */ as part of a literal string. Consider the following program segment:

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

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

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

This is incorrect because the language processor would interpret the /* that is part of the literal string /*123 as the start of a nested comment. It would then fail to process the rest of the program because it would be looking for a matching comment end (*/).

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

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

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

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

Literal Strings:

A literal string is a sequence including any character except linefeed (X'10') and delimited by the single quote (') or the double quote ("). Use two consecutive double quotation marks ("") to represent a " character within a string delimited by double quotation marks. Similarly, use two consecutive single quotation marks ('') to represent a ' character within a string delimited by single quotation marks. A literal string is a constant and its contents are never modified when it is processed. Literal strings must be complete on a single line (this means that unmatched quotation marks may be detected on the line where they occur).

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

These are valid strings:

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

Implementation maximum: A literal string can contain up to 250 characters. The length of the evaluated result of an expression, however, is only limited by the available virtual storage of your computer, with an additional limit of 512MB maximum per process.

Note that a string followed immediately by a (is considered to be the name of a function. If followed immediately by the symbol X or x, it is considered to be a hexadecimal string. If followed immediately by the symbol B or b, it is considered to be a binary string. These forms are now described in detail.

Hexadecimal Strings:

Any sequence of zero or more hexadecimal digits (0–9, a–f, A–F), optionally separated by blanks, delimited by single or double quotation marks, and immediately followed by the symbol X or x (neither can be part of a longer symbol). A single leading 0 is added, if necessary, at the front of the string to make an even number of hexadecimal digits, which

represent a character string constant formed by packing the hexadecimal digits given. Packing the hexadecimal digits converts the pair of hexadecimal digits into one equivalent character, for example: '41'X to A. The blanks, which may be present only at byte boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

These are valid hexadecimal strings:

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

Note: If you use explicit hexadecimal strings, your program may operate differently when ported to different machines.

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

Binary Strings:

Any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles), optionally separated by one or more blanks. The first group may have fewer than four digits; in this case, up to three 0 digits are assumed to the left of the first digit, making a total of four digits. The entire string is delimited by matching single or double quotation marks, and immediately followed by the symbol b or B (which cannot be part of a longer symbol). The blanks, which may only be present at byte or nibble boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

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

These are valid binary strings:

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

Note: If you use explicit binary strings, your program may operate differently when ported to different machines.

Implementation maximum: The packed length of a binary-literal string may not exceed 100 bytes.

Symbols:

Symbols are groups of characters, selected from the:

- English alphabetic characters (A–Z and a–z)
- Numeric characters (0–9)
- Characters . ! ? and underscore.

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

These are valid symbols:

```
Fred
Albert.Hall
WHERE?
```

A symbol can be a label (see page 2-12) or a REXX keyword (see page 11-1). If a symbol does not begin with a digit or a period, you can use it as a variable and can assign it a value. If you have not assigned it a value, its value is the characters of the symbol itself, translated to uppercase (that is, lowercase a–z to uppercase A–Z). Symbols that begin with a number or a period are constant symbols and cannot be assigned a value. A symbol may include other characters in one situation only. If the first part of a symbol starts with a digit (0–9) or a period, it may end with the sequence E or e, followed immediately by an optional sign (- or +), followed immediately by one or more digits (which cannot be followed by any other symbol characters). The symbol thus defined may be a number in exponential notation. The sign in this context is part of the symbol and is not an operator.

These are valid numbers in exponential notation:

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

Numbers:

These are character strings consisting of one or more decimal digits, optionally prefixed by a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 6-1-6-9 for a full definition of numbers.

Numbers can have leading blanks (before and after the sign, if any) and can have trailing blanks. Embedded blanks are not permitted. Note that a symbol (see preceding) or a literal string may be a number. A number cannot be the name of a variable.

These are valid numbers:

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

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

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

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

Operators:

The characters: `+ - \ / % * | ¦ & = ¬ > <` and the sequences `>= <= \> \< \= >< <> == \== // && || ** ¬> ¬< ¬= ¬== >> << >>= \<< ¬<< \>> ¬>> <<=` are operator tokens (see page 2-8), with or without embedded blanks or comments. (For the OS/2 operating system, `¦¦` can also be used as the concatenation symbol.) A few of these are also used in parsing templates, and the equal sign is also used to indicate assignment. Blanks (and comments) adjacent to operator characters have no effect on the operator; thus, operators constructed from more than one character can have embedded blanks and comments. One or more blanks, where they occur in expressions but are not adjacent to another operator, also act as an operator. Blanks adjacent to operator characters are removed. Therefore, the following are identical in meaning.

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

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

Note: On the OS/2 operating system, the REXX interpreter uses ASCII character 124 in the concatenation operator and as the logical OR operator. Depending on the code page or keyboard for your particular country, ASCII 124 may be shown as a solid vertical bar (`|`) or a split vertical bar (`¦`). The character on the screen may not match the character engraved on the key. If you are receiving error 13, `invalid character in program` on an instruction including a vertical bar character, make sure this character is ASCII 124.

The REXX interpreter uses ASCII character 170 for the logical NOT operator. Depending on your country, the `¬` might not appear on your keyboard. If the character is not available, the backslash (`\`) may be used in place of `¬`.

Special Characters:

The characters `, ; : ) (` together with the individual characters from the operators have special significance when found outside of literal strings. All these characters constitute the set of special characters. They all act as token delimiters, and blanks adjacent to any of these are removed. There is an exception: a blank adjacent to the outside of a parenthesis is deleted only if it is also adjacent to another special character (unless the character is a parenthesis and the blank is outside it, too). For example, the clause:

```
'REPEAT'   B + 3;
```

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

```
'REPEAT' B+3;
```

Implied Semicolons

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

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

- The line ends in the middle of a comment
- The last noncomment token was the continuation character (denoted by a comma).

In these situations, it is not considered the end of a clause and a semicolon is not implied.

Semicolons are also implied automatically after certain keywords when they are used in the correct context. The keywords that have this effect are: ELSE, OTHERWISE, and THEN. These special cases reduce typographical errors significantly.

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

Continuations

One way to continue a clause onto the next line is to use the comma, which is referred to as the **continuation character**. The comma is functionally replaced by a blank, and, thus, no semicolon is implied.

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

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

This displays:

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

Expressions and Operators

Expressions

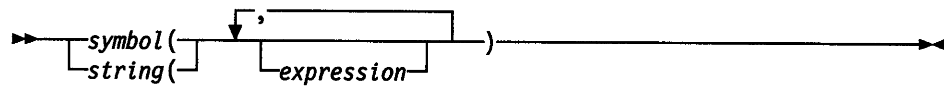
Clauses can include expressions consisting of **terms** (strings, symbols, and function calls) interspersed with operators and parentheses.

Terms include:

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

during evaluation. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.

- **Function invocations**—see page 4-1—which are of the form:



Evaluation of an expression is left to right, modified by parentheses and by operator precedence in the usual algebraic manner (see "Parentheses and Operator Precedence" on page 2-11). Expressions are always wholly evaluated, unless an error occurs during evaluation.

All data is in the form of "typeless" character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so forth). Consequently, the result of evaluating any expression is itself a character string. All terms and results (except arithmetic and logical expressions) may be the **null string** (a string of length `0`). Note that REXX imposes no restriction on the maximum length of results. Expression results are limited only by the amount of storage available to the REXX interpreter.

Operators

The following pages describe how each operator (except for the prefix operators) acts on two terms, which may be symbols, strings, function calls, intermediate results, or sub-expressions in parentheses. Each prefix operator acts on the term or sub-expression that follows it. There are four types of operators:

String Concatenation

The concatenation operators combine two strings to form one string. The combination may occur with or without an intervening blank:

(blank) Concatenate terms with one blank in between

|| Concatenate without an intervening blank

Note: For the OS/2 operating system, ¦¦ can also be used as the concatenation symbol. See page 2-6 for additional information on the OS/2 concatenation operator.

(abuttal) Concatenate without an intervening blank

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

The **abuttal** operator is assumed between terms that are not separated by another operator. This can occur when two terms are syntactically distinct, such as a literal string and a symbol, or when they are separated only by a comment. An example of syntactically distinct terms is: if `Fred` has the value `37.4`, then `Fred'%'` evaluates to `37.4%`. Any comments between the terms are irrelevant.

Examples:

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

The two adjoining strings, one hexadecimal and one literal, `'4a 4b'x'LMN'` evaluate to `JKLMN`.

In the case of:

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

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

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

results in an abuttal, and evaluates to 37.40.

Arithmetic

You can combine character strings that are valid numbers (see page 2-5) using the arithmetic operators:

+	Add
-	Subtract
*	Multiply
/	Divide
%	Divide and return the integer part of the result
//	Divide and return the remainder (not modulo, since the result may be negative)
**	Power (raise a number to a whole-number power)
Prefix -	Negate the following term. Same as the subtraction: 0-term.
Prefix +	Take the following term as if it was the addition: 0+term.

See Chapter 6, "Numbers and Arithmetic" on page 6-1 for details of accuracy, the format of valid numbers, and the combination rules for arithmetic. Note that if an arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

The comparison operators return the value 1 if the result of the comparison is true, or 0 otherwise.

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

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

Character comparison and strict comparison operations are both case-sensitive, and for both the exact collating order may depend on the character set used for the implementation. For example, in an EBCDIC environment, lowercase alphabetics precede uppercase, and the digits 0–9 are higher than all alphabetics. In an ASCII environment, the digits are lower than the alphabetics, and lowercase alphabetics are higher than uppercase alphabetics. The OS/2 operating system in an ASCII environment.

The comparison operators and operations are:

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

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

Logical (Boolean)

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

&
AND
Returns 1 if both terms are true.

|
Inclusive OR
Returns 1 if either term is true.

&&
Exclusive OR
Returns 1 if either (but not both) is true.

Prefix \,¬
Logical NOT
Negates; 1 becomes 0 and vice-versa.

Note: See page 2-6 for information on logical operators used with the OS/2 operating system.

Parentheses and Operator Precedence

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

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

  `term1 operator1 term2 operator2 term3 ...`

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

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

For example, * (multiply) has a higher priority than + (add), so 3+2*5 evaluates to 13 (rather than the 25 that would result if strict left to right evaluation occurred). Likewise, the expression -3**2 evaluates to 9 (instead of -9) because the prefix minus operator has a higher priority than the power operator.

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

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

Examples:

Suppose the symbol `A` is a variable whose value is `3` and `DAY` is a variable whose value is `Monday`. Then:

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

Note: The REXX order of precedence usually causes no difficulty because it is the same as in conventional algebra and other computer languages. There are two differences from common notations:

- The prefix minus operator always has a higher priority than the power operator.
- Power operators (like other operators) are evaluated left-to-right.

For example:

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

Clauses and Instructions

Clauses can be subdivided into the following types:

Null Clauses

A clause consisting only of blanks or comments or both is a **null clause**. It is completely ignored.

Note: A null clause is not an instruction; for example, putting an extra semicolon after the THEN or ELSE in an IF instruction is not equivalent to using a dummy instruction (as it would be in C). The NOP instruction is provided for this purpose.

Labels

A clause that consists of a single symbol followed by a colon is a **label**. The colon in this context implies a semicolon (clause separator), so no semicolon is required. Labels identify the targets of CALL instructions, SIGNAL instructions, and internal function calls. They can be traced selectively to aid debugging.

Any number of successive clauses may be labels, thus permitting multiple labels before another type of clause. Duplicate labels are permitted, but because the search effectively starts at the top of the program, the control, following a CALL or SIGNAL instruction, is always passed to the first occurrence of the label. The duplicate labels occurring later can be traced but cannot be used as a target of a CALL, SIGNAL, or function invocation.

Instructions

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

Assignments

Single clauses of the form *symbol=expression* are instructions known as **assignments**. An assignment gives a variable a (new) value. See "Assignments and Symbols."

Keyword Instructions

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

```
DO
   instruction
   instruction
   instruction
END
```

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

Commands

Single clauses consisting of just an expression are instructions known as **commands**. The expression is evaluated and passed as a command string to the currently active environment.

Assignments and Symbols

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

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

symbol=expression;

is taken to be an assignment. The result of *expression* becomes the new value of the variable named by the symbol to the left of the equal sign.

Example:

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

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

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

Example:

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

Symbols can be subdivided into four classes: constant symbols, simple symbols, compound symbols, and stems. Simple symbols can be used for variables where the name corresponds to a single value. Compound symbols and stems are for more complex collections of variables, such as arrays and lists.

Constant Symbols

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

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

These are constant symbols:

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

Simple Symbols

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

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

These are simple symbols:

```
FRED
Whatagoodidea?    /* Same as WHATAGOODIDEA? */
?12
```

Compound Symbols

A **compound symbol** contains at least one period and at least two other characters. It cannot start with a digit or a period, and, if there is only one period, the period cannot be the last character.

The name begins with a **stem** (that part of the symbol up to and including the first period). This is followed by a **tail**, parts of the name (delimited by periods) that are constant symbols, simple symbols, or null. You cannot use constant symbols with embedded signs (for example, 12.3E+5) after a stem; in this case, the whole symbol would not be a valid symbol.

These are compound symbols:

```
FRED.3
Array.I.J
AMESSY..One.2.
```

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

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

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

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

is given by

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

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

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

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

```
a=3     /* assigns '3' to the variable A    */
b=4               /*  '4'     to B          */
c='Fred'          /*  'Fred'  to C          */
a.b='Fred'        /*  'Fred'  to A.4        */
a.fred=5          /*  '5'     to A.FRED     */
a.c='Bill'        /*  'Bill'  to A.Fred     */
c.c=a.fred        /*  '5'     to C.Fred     */
x.a.b='Annie'     /*  'Annie' to X.3.4      */
say  a  b  c  a.a  a.b  a.c  c.a  a.fred x.a.4
/* displays the string:                     */
/*    "3 4 Fred A.3 Fred Bill C.3 5 Annie"  */
```

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

Stems

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

These are stems:

```
FRED.
A.
```

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

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

For example:

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

say  hole.1  hole.mouse  hole.9

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

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

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

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

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

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. `DROP FRED.` drops all variables with that stem (see page 3-14), and `PROCEDURE EXPOSE FRED.` exposes *all possible* variables with that stem (see page 3-27).

Notes:

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

2. Since an expression can include the operator =, and an instruction may consist purely of an expression (see next section), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is

(or starts with) an equal sign (=) is an **assignment**, rather than an expression (or an instruction). This is not a restriction, since you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

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

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

is an assignment, not an ADDRESS instruction.

Commands to External Environments

Environment

The base system for the language processor is assumed to include at least one active environment for processing commands. One of these is selected by default on entry to a REXX program. You can change the environment by using the ADDRESS instruction. You can find out the name of the active environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program. The environment so selected will depend on the caller; for example if a REXX program is called from the OS/2 operating system, then the default environment is CMD. If called from an editor that accepts subcommands from the language processor, the default environment may be that editor.

A REXX program can issue commands — called *subcommands* — to other OS/2 application programs. For example, a REXX program written for a text editor can inspect a file being edited, issue subcommands to make changes, test return codes to check that the subcommands have been executed as expected, and display messages to the user when appropriate.

An application that uses REXX as a macro language must register its environment with the REXX language processor. For a discussion of this mechanism see "Subcommand Interfaces" on page 9-9.

Commands

To issue a command to the active environment, use a clause of the form:

```
expression;
```

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

The environment then processes the command (which may have side-effects). It eventually returns control to the language processor, after setting a **return code**. The language processor places this return code in the REXX special variable RC. For example, where the default environment is the OS/2 operating system, the sequence:

```
fname = "CHESHIRE"
exten = "CAT"
"ERASE" fname"."exten
```

would result in the string `ERASE CHESHIRE.CAT` being passed to the OS/2 operating system. Of course, the simpler expression:

```
"ERASE CHESHIRE.CAT"
```

would have the same effect in this case.

On return, the return code placed in RC will have the value 0. if the file CHESHIRE.CAT were erased; a nonzero value if the file could not be found in the current directory.

Because of the return codes, errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is ON (see Chapter 7, "Conditions and Condition Traps" on page 7-1). They may also cause the command to be traced if `TRACE E` or `TRACE F` is set. `TRACE Normal` is the same as `TRACE F`, and is the default—see page 3-37.

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

```
delete "*".lst          /* not "multiplied by" */

var.003 = anyvalue
type "var.003"          /* not a compound symbol */

w = any
dir"/w"                 /* not "divided by ANY" */
```

Using REXX on the OS/2 Operating System

This section contains some general tips and information about using REXX on Operating System/2. Specifically:

- How a REXX program calls another REXX program
- How a return code is generated
- How a REXX program is run
- How commands are edited.

Calling Another REXX Program

REXX programs can call other REXX programs as external functions or subroutines. REXX programs can also call each other as commands by using the OS/2 CALL command. Performance is improved by using subroutine or function calls rather than using the OS/2 CALL command.

Generating a Return Code

When REXX programs call other REXX programs as commands, the return code of the command is the exit value of the called program, when that value is a whole number in the range -32768 to 32767. Otherwise the exit value is ignored and the called program is given a return code of 0.

Running a REXX Program

REXX programs on OS/2 are executed in two stages. First, the entire program is scanned, and a special version of it is constructed. This version is called the tokenized image. The tokenized image is then used during the second phase, when the instructions of the program are actually run. Running the program in two stages causes the program to run faster than if the program were run in a single stage.

To further enhance performance, the tokenized image is saved in an extended attribute associated with the source file. Subsequent execution then skips the tokenization stage and uses the tokenized image that is already present. If the program is a read-only file, the tokenized is not saved and is recreated each time the program runs. The extended attributes of a file are limitted to 64KB in size, so for a large REXX program it may be impossible to save the tokenized image. Again, this means that the program will be re-tokenized each time it runs. There is no limit on the size of a REXX program, but for the best performance, you should break very large programs up into smaller pieces. The great majority of REXX programs have a tokenized image smaller than 64KB.

Note: The REXX interpreter may be changed when installing a new release of the OS/2 operating system or when installing a corrective service diskette. This means that when the REXX interpreter is changed, the tokenized image of programs may be updated. Also, when sharing REXX programs over a network between different systems, the best performance is achieved by ensuring all the systems have the same level of OS/2. The tokenized image, therefore, is valid for all users. Also, when upgrading to another level of REXX, re-tokenizing the programs on the network drive improves performance.

Editing Commands

When running a REXX program from the command line, the normal command line editing takes place:

- ¦ means piping
- > and < mean redirection
- /q suppresses echoing of OS/2 commands

Specifying //t causes REXX to tokenize the program and save the tokenized image—but not execute the program. This is the only parameter the REXX interpreter currently recognizes. However, the characters // are reserved for future REXX options. The characters // should not be used in a parameter to a REXX program

Chapter 3. Keyword Instructions

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

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

As explained in "Keyword Instructions" on page 2-13, a keyword instruction is recognized *only* if its keyword is the first token in a clause, and if the second token does not start with an = character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. Therefore,

```
arg(fred) rest
```

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

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

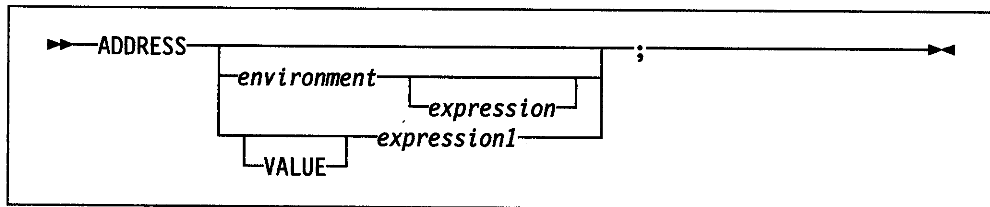
Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the expression from the subkeyword in the example following:

```
ADDRESS VALUE expression
```

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

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

ADDRESS



ADDRESS temporarily or permanently changes the destination of commands.

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 execution of the command, *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    */

/* In a mainframe (for example, CMS) environment, the  */
/* ADDRESS instruction might be used like this: */

ADDRESS CMS 'STATE PROFILE EXEC A' /*   VM      */
```

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

Examples:

Say 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. You can omit the subkeyword VALUE if *expression1* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator character or parenthesis).

Example:

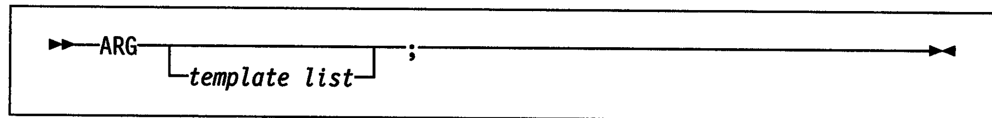
```
ADDRESS ('ENVIR'||number)
```

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.

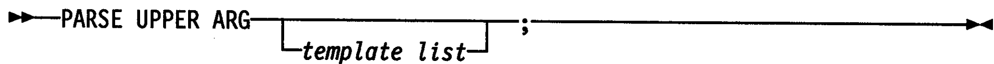
The two environment names are automatically saved across subroutine and internal function calls. See the CALL instruction (page 3-6) for more details.

You can retrieve the current ADDRESS setting using the ADDRESS built-in function ("ADDRESS" on page 4-7). The registration of alternative subcommand environments is described on page 9-9.

ARG



ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is just a short form of the instruction



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

Unless a subroutine or internal function is being executed, the strings passed as parameters to the program are parsed into variables according to the rules described in the section on parsing (page 5-1).

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

In either case, the strings passed are translated to uppercase (that is, lowercase a–z to uppercase A–Z) before they are processed. Use the PARSE ARG instruction if you do not desire uppercase translation.

The ARG (and PARSE ARG) instructions can be executed as often as desired (typically with different templates) and always parse the same current input string (or strings). The only restrictions on the length or content of the data parsed are those the caller imposes.

Example:

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

Arg adjective noun .

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

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

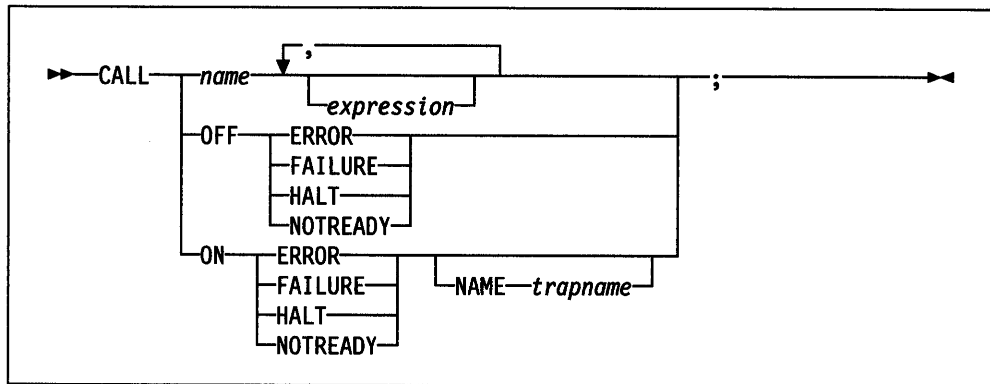
Example:

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

Fred:  Arg string, num1, num2

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

Notes:

1. The ARG built-in function can also retrieve or check the argument strings to a REXX program or internal routine. See page 4-7.
2. The source of the data being processed is also made available on entry to the program. See the PARSE instruction (SOURCE option) on page 3-26 for details.

CALL



CALL invokes a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

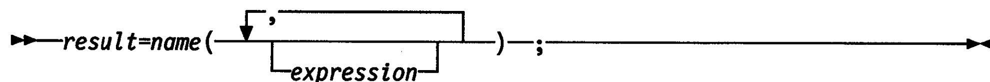
To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, "Conditions and Condition Traps" on page 7-1.

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

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

If *name* is a string (that is, you specify *name* in quotation marks), the search for internal routines is bypassed, and only a built-in function or an external routine is invoked. Note that the names of built-in functions are in uppercase, and, therefore you should uppercase the name in the literal string. On the OS/2 operating system, file names may be in upper, lower, or mixed case. The OS/2 operating system uses a case insensitive search for files; therefore, when using CALL to run a REXX subroutine contained on a disk file, the case does not matter.

The invoked routine can optionally return a result, and so the CALL instruction is functionally identical with the clause:



except that the variable RESULT becomes uninitialized if the routine invoked returns no result.

The OS/2 operating system supports specifying up to 20 expressions, separated by commas. 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 accesses 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 exactly the same mechanism as function calls. The section on functions (page 4-1) describes the order in which these are searched for; briefly, it is as follows:

Internal routines:
These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. You can use SIGNAL and CALL together to call an internal routine whose name is determined at the time of execution; this is known as a multi-way call (see page 3-36). The RETURN instruction completes the execution of an internal routine.

Built-in routines:
These are routines built into the language processor for providing various functions. They always return a string containing the result of the function. (See page 4-5.)

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

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

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

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

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

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

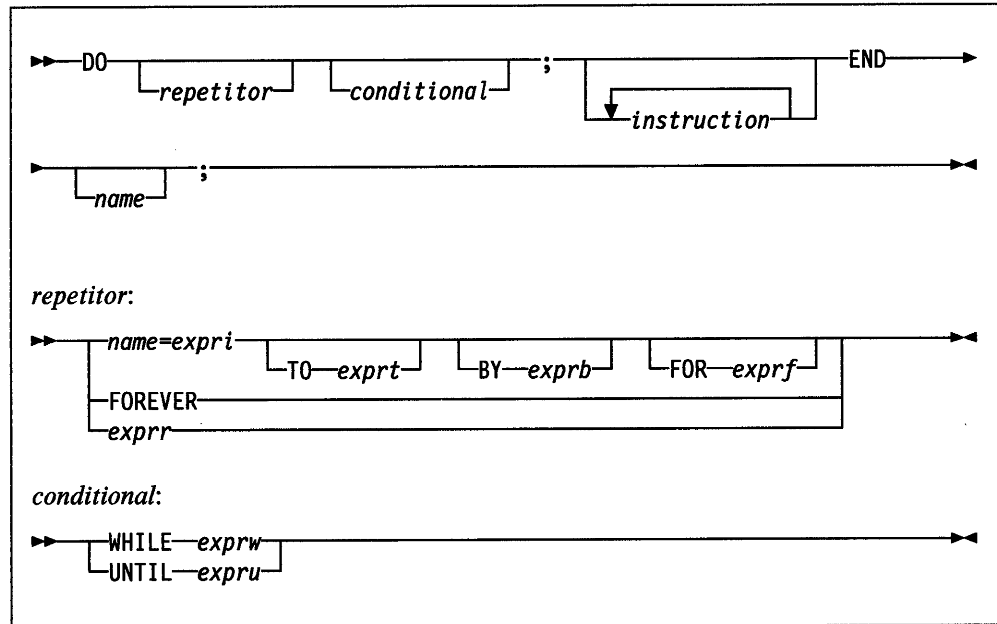
Example:

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

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

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

- **The status of DO loops and other structures**: Executing a SIGNAL while within a subroutine is safe because DO loops, and so forth, that were active when the subroutine was called are not terminated. (But those currently active within the subroutine are terminated).
- **Trace action**: Once a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this does not affect the tracing of the caller. Conversely, if you only wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) is saved across routines.
- **NUMERIC settings**: The DIGITS, FUZZ, and FORM of arithmetic operations, (in "NUMERIC" on page 3-22) are saved and are then restored on return. A subroutine can, therefore, set the precision, and so forth, that it needs to use without affecting the caller.
- **ADDRESS settings**: The current and previous destinations for commands (see "ADDRESS" on page 3-2) are saved and are then restored on return.
- **Condition traps**: CALL ON and SIGNAL ON are saved and then restored on return. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions the caller set up.
- **Condition information**: This is the information the CONDITION built-in function returns. See "CONDITION" on page 4-14.
- **Elapsed-time clocks**: A subroutine inherits the elapsed-time clock from its caller (see "TIME" on page 4-38), but because the time clock is saved across routine calls, a subroutine or internal function can independently restart and use the clock without affecting its caller. For the same reason, a clock started within an internal routine is not available to the caller.
- **OPTIONS settings**: ETMODE and EXMODE are saved and are then restored on return. For more information, see "OPTIONS" on page 3-24.

Implementation maximum: The total nesting of control structures, which includes internal and external routine calls, may not exceed a depth of 100.



DO groups instructions together and optionally executes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

Syntax Notes:

- The *exprr*, *expri*, *exprb*, *exprt*, and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a nonnegative whole number. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.
- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
- The *instruction* can be any instruction, including assignments, commands, and keyword instructions (these include 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 be used as symbols in any of the expressions. FOREVER is also reserved, but only if it immediately follows the keyword DO.
- The *exprb* option defaults to 1, if relevant.

Simple DO Group

If you specify neither *repetitor* nor *conditional*, 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 are both  */
/* executed if A has the value "3".                  */
If a=3 then Do
              a=a+2
              Say 'Smile!'
            End
```

Simple Repetitive Loops

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

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

Note: For a discussion on conditional phrases, see "Conditional Phrases (WHILE and UNTIL)" on page 3-12.

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 token of *exprr* is a symbol and the second token is (or starts with) =, 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) before the first execution of the instruction list. The variable is then stepped (by adding the result of *exprb*, at the bottom of the loop) each time the group of instructions is executed. The group is executed repeatedly while the end condition (determined by the result of *exprt*) is not met. If *exprb* is positive or 0, the loop is 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 executes indefinitely unless some other condition terminates it.

Example:

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

The numbers do not have to be whole numbers:

Example:

```
X=0.3                       /* Displays:   */
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 usually considered good programming practice, though it may be appropriate in certain circumstances.

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

The execution of a controlled loop can be bounded further by a FOR phrase. In this case, you must specify *exprf*, and it must evaluate to a 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. Like the TO and BY expressions, it is evaluated once only—when the DO instruction is first executed and before the control variable receives its initial value. Like the TO condition, the FOR condition is checked at the start of each iteration.

Example:

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

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

Example:

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

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

Conditional Phrases (WHILE and UNTIL)

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

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

Example:

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

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

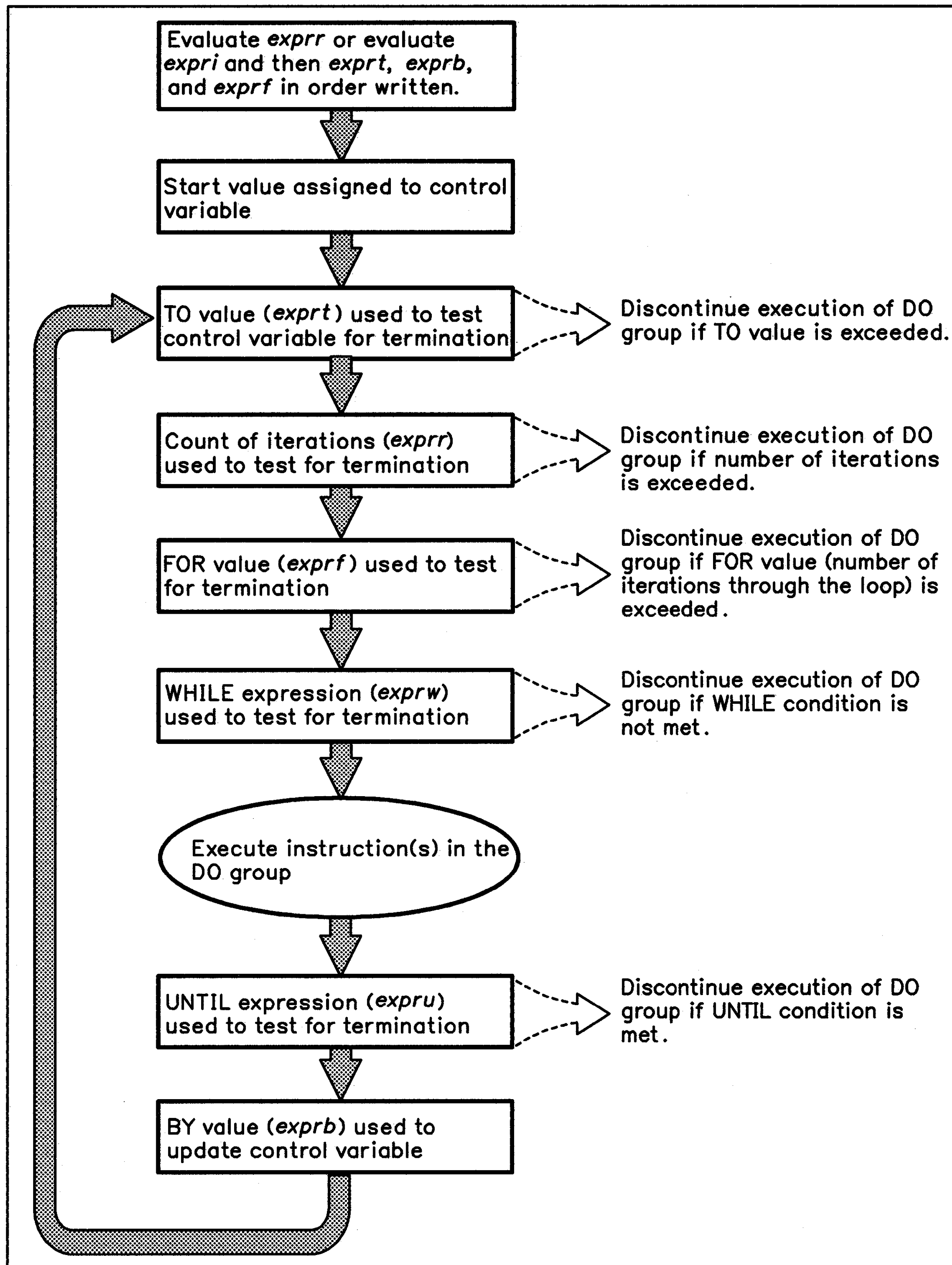
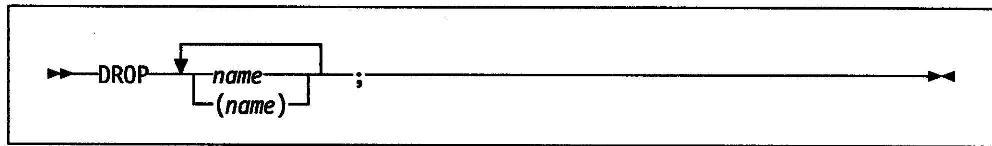


Figure 3-1. Concept of a DO Loop

DROP



DROP "unassigns" variables, that is, restores them to their original uninitialized state. Each *name* identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments. Optionally, you can enclose a *name* in parentheses to denote a subsidiary list.

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. (Blanks are not necessary either inside or outside the parentheses, but you can add them if desired.) This subsidiary list must follow the same rules as the main list (that is, be valid variable names, separated by blanks) except that no parentheses are allowed. 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
/* Resets 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:

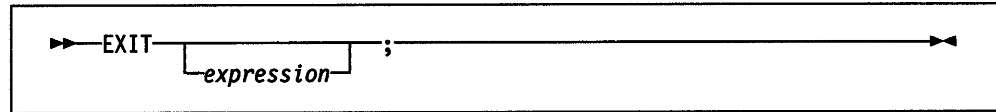
```
mylist='a b c'
drop (mylist) d
/* Resets the variables A, B, C, and D        */
/* Does not drop MYLIST                       */
```

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

Example:

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

EXIT



EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is terminated immediately, even if an internal routine is currently being executed. If no internal routine is active, RETURN (see page 3-32) and EXIT are identical in their effect on the program that is being executed.

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

Example:

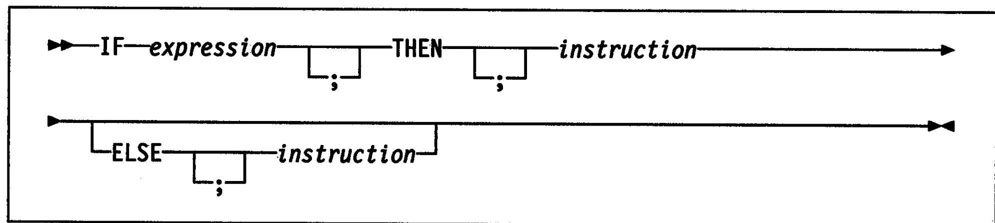
```
j=3
Exit j*4
/* Would exit with the string '12' */
```

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 instruction EXIT, 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 underlying operating system. (*Host* in this sense means the current command environment.) 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)). If the conversion fails, no error is raised, and a 0 return code is returned.

IF



IF conditionally processes 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, the instruction after the 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 before the ELSE.

Example:

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

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

Example:

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

Notes:

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

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

INTERPRET

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

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

The *expression* is evaluated and is then executed (interpreted) just as though the resulting string were 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 execution, if one was not supplied.

Example:

```
data='FRED'
interpret data '= 4'
/* 1) Builds the string  "FRED = 4"        */
/* 2) Executes:  FRED = 4;                 */
/* Thus the variable FRED is set to "4"    */
```

Example:

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

Notes:

1. Labels within the interpreted string are not permanent and are therefore an error.
2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with `TRACE R` or `TRACE I` set is helpful.

   Example:

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

   When this is run it gives the trace:

```
[C:\]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!
[C:\]
```

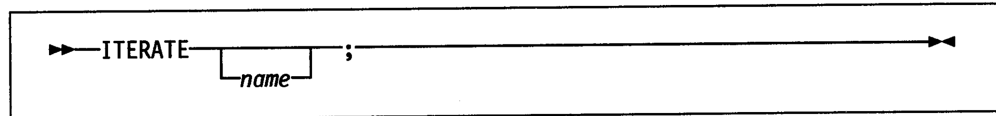
Here, lines 3 and 4 set the variables used in line 5. Execution of line 5 then proceeds in two stages. First the string to be interpreted is built up, using a literal string, a variable (INDIRECT), and another literal. The resulting pure character string is then interpreted, just as though it were actually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

3. For many purposes, the VALUE function (see page 4-41) 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 required only in special cases, such as when more than one statement is to be interpreted at once.

ITERATE



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

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

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

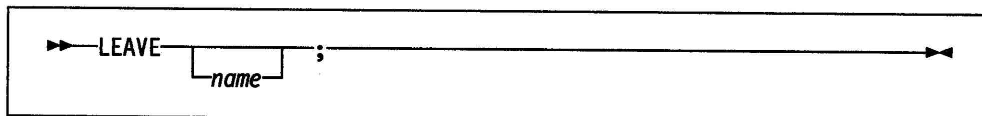
Example:

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

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. ITERATE cannot be used to step an inactive loop.
3. If more than one active loop uses the same control variable, ITERATE selects the innermost loop.

LEAVE



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

Processing of the group of instructions is terminated, and control is passed to the instruction following the END clause, just as though the END clause had been encountered and the termination condition had been met 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), and that loop (and any active loops inside it) is then terminated. Control then passes to the clause following the END that matches the DO clause of the selected loop.

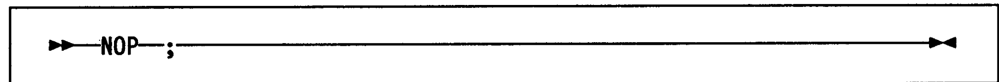
Example:

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

Notes:

1. If specified, *name* must match the symbol naming the control variable in the DO clause in all respects except case. No substitution for compound variables is carried out when the comparison is made.
2. A loop is active if it is currently being executed. If a subroutine is called (or an INTERPRET instruction is executed) during execution of a loop, the loop becomes inactive until the subroutine has returned or the INTERPRET instruction has completed. LEAVE cannot be used to terminate an inactive loop.
3. If more than one active loop uses the same control variable, LEAVE selects the innermost loop.

NOP



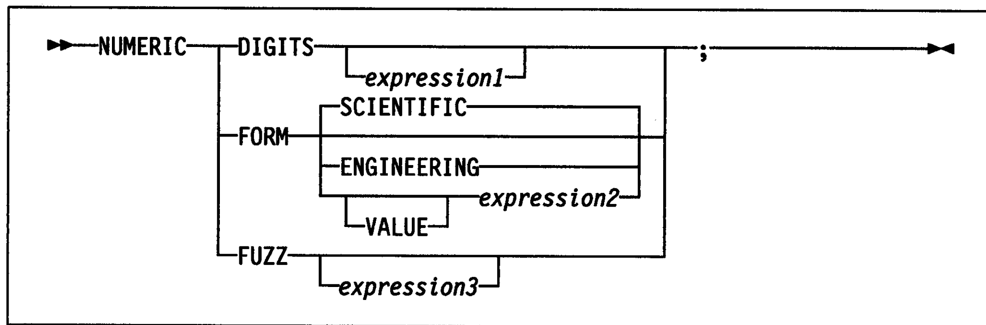
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

Example:

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

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

NUMERIC



NUMERIC changes the way in which arithmetic operations are carried out. The options of this instruction are described in detail on pages 6-1-6-9, but in summary:

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

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

You can retrieve the current NUMERIC DIGITS setting with the DIGITS built-in function. See "DIGITS" on page 4-19.

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

You can retrieve the current NUMERIC FORM setting with the FORM built-in function. See "FORM" on page 4-23.

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

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−FUZZ` digits and this result is then compared with 0.

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function. See "FUZZ" on page 4-24.

Note: The three numeric settings are automatically saved across subroutine and internal function calls. See the CALL instruction (page 3-6) for more details.

OPTIONS

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

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

The language processor recognizes the following words:

ETMODE specifies that literal strings and comments containing DBCS characters are checked for being valid DBCS strings.

NOETMODE specifies that literal strings and comments containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default.

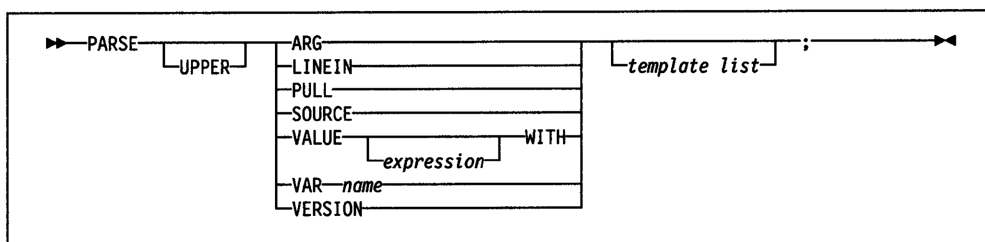
EXMODE specifies that instructions, operators, and functions handle DBCS data in mixed strings on a logical character basis. DBCS data integrity is maintained.

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

Notes:

1. Because of the language processor's scanning procedures, you are advised to place an `OPTIONS 'ETMODE'` instruction as the first instruction in a program containing DBCS literal strings and DBCS comments.
2. To ensure proper scanning of a program containing DBCS literals and DBCS comments, enter the words ETMODE, NOETMODE, EXMODE, and NOEXMODE as literal strings (that is, enclosed in quotation marks) in the OPTIONS instruction.
3. The OPTIONS ETMODE and OPTIONS EXMODE settings are saved and restored across subroutine and function calls.
4. The words ETMODE, NOETMODE, EXMODE, and NOEXMODE can appear several times within the result. The one that takes effect is determined by the last valid one specified between the pairs ETMODE-NOETMODE and EXMODE-NOEXMODE.

PARSE



PARSE assigns data (from various sources) to one or more variables according to the rules and templates described in the section on parsing (page 5-1).

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

Each template is applied to a single source string. Specifying multiple templates is never a syntax error, but only the PARSE ARG variant can supply more than one non-null source string. See page 5-10 for information on parsing multiple source strings.

If you do not specify a template, 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 queue is empty), a line is taken from the default input stream, and for PARSE VALUE, *expression* is evaluated. For PARSE VAR, the specified variable is accessed. If it does not have a value, the NOVALUE condition is raised, if it is enabled.

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

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

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

Note: You can also retrieve or check the argument strings to a REXX program or internal routine with the ARG built-in function (page 4-7).

PARSE LINEIN
The next line from the default input stream is parsed. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) PARSE LINEIN is a shorter form of the 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.

To check if any lines are available in the default input stream, use the built-in function LINES; see page 4-28. Also see page 4-25 for a description of the LINEIN function.

PARSE PULL
The next string from the external data queue is parsed. If the external data queue is empty, lines are read from the default input (typically the user's terminal). 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 with the QUEUED built-in function, described on page 4-30. The queue remains active as long as the language processor is active. Other programs in the system can alter the queue and use it as a means of communication with 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). (See the PULL instruction, on page 3-29, for further details.)

PARSE SOURCE
The data parsed describes the source of the program being executed.

The source string contains the characters OS/2, 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 via the CALL instruction. These two tokens are followed by the complete path specification of the program file.

The string parsed might therefore look like this:

```
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 cannot be used as a symbol within *expression.*

Thus, for example:

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

gets the current time and splits it 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 cannot start with a period or a digit). Note that the variable *name* is not changed unless it appears in the template, so that for example:

```
PARSE VAR string word1 string
```

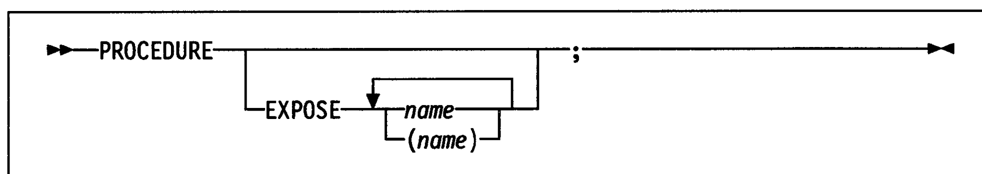
removes the first word from *string*, puts it in the variable *word1*, and assigns the remainder back to *string*. Similarly

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

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

PARSE VERSION
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 (for example, `4.00, 13 June 1989`).

PROCEDURE



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

If you use the EXPOSE option, any variable specified by *name* is exposed, so that any reference to it (including setting and dropping) is made to the variables environment 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. Any variables in the main program that are not exposed are still protected. Therefore, some limited set of the caller's variables can be made accessible, and these variables can be changed (or new variables in this set can be created). All these changes are visible to the caller upon RETURN from the routine.

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 the caller has not used as a variable.

Example:

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

toft: procedure expose j k x.j
   say j k x.j  /* Displays "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 a single *name* is enclosed in parentheses then, after that variable is exposed, the value of the variable is immediately used as a subsidiary list of variables. (Blanks are not necessary either inside or outside the parentheses, but you can add them if desired.) This list must follow the same rules as the main list (that is, valid variable names, separated by blanks) except that no parentheses are allowed. The variables named in a subsidiary list are also exposed from left to right.

Example:

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

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

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

Example:

```
/* This is the main program */
a=11; b=12; c=13
Showlist='a b'     /* but not C              */
call Playvars
say a b c d        /* Displays "11 New 13 9" */
exit

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

Specifying a **stem** as *name* exposes this stem and *all possible* compound variables whose names begin with that stem. (A **stem** is a symbol containing just one period, which is the last character. See page 2-16.)

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 sets A.1 in the caller's         */
         /* environment, even if it did not       */
         /* previously exist.                     */
```

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

Notes:

1. Only one PROCEDURE instruction in each level of routine call is allowed.
2. An internal routine need not include a PROCEDURE instruction, in which case the variables it is manipulating are those the caller "owns."

See the CALL instruction and function descriptions on pages 3-6 and 4-1 for details and examples of how routines are invoked.

PULL

```
>>--PULL--+---------------+--;-------------------------------------><
          +-template list-+
```

PULL reads a string from the head of the external data queue. It is just a short form of the instruction:

```
>>--PARSE UPPER PULL--+---------------+--;-------------------------><
                      +-template list-+
```

The current head-of-queue is read as one string. Without a *template list* specified, no further action is taken (and the string is thus effectively discarded). If specified, a *template list* is usually a single template, which is a list of symbols separated by blanks or patterns or both. The *template list* can be several templates separated by commas, but PULL parses only one source string; if you specify several comma-separated templates, variables in templates other than the first one are assigned the null string. The string is translated to uppercase (that is, lowercase a–z to uppercase A–Z) and then parsed into variables according to the rules described in the section on parsing (page 5-1). Use the PARSE PULL instruction if you do not desire uppercase translation.

Note: If the current data queue is empty, PULL reads instead from STDIN (typically, the keyboard). A question-mark is displayed to the user as a prompt. The length of data read by the PULL instruction is restricted to the length of strings contained by variables.

Example:

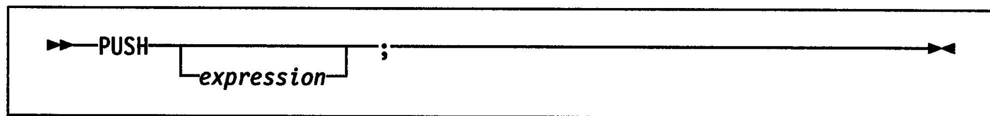
```
Say 'Do you want to erase the file?  Answer Yes or No:'
Pull answer .
if answer='NO' then say 'The file will not be erased.'
```

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

If the external data queue is empty, a line is read from the default input stream and the program pauses, if necessary, until a line is complete. (This is as though PARSE UPPER LINEIN had been executed. See page 3-25.)

The QUEUED built-in function ("QUEUED" on page 4-30) returns the number of lines currently in the external data queue.

PUSH



PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.)

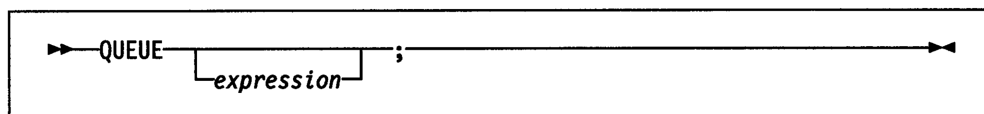
If you do not specify *expression*, a null string is stacked.

Example:

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

The QUEUED built-in function ("QUEUED" on page 4-30) returns the number of lines currently in the external data queue.

QUEUE



QUEUE appends the string resulting from *expression* to the tail of the external data queue. That is, it is added FIFO (First In, First Out). (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.)

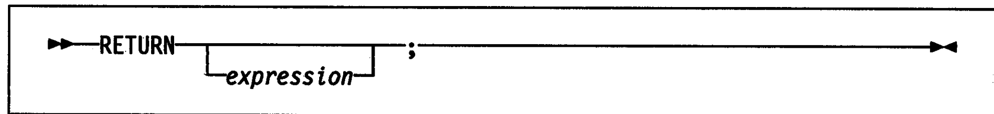
If you do not specify *expression*, a null string is queued.

Example:

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

The QUEUED built-in function ("QUEUED" on page 4-30) returns the number of lines currently in the external data queue.

RETURN



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

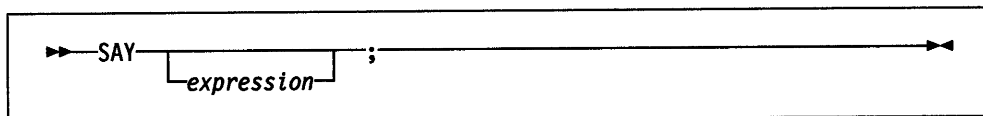
If no internal routine (subroutine or function) is active, RETURN and EXIT are identical in their effect on the program that is being executed. (See page 3-15.)

If a *subroutine* is being executed (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so forth) are also restored. (See page 3-6.)

If a *function* is being executed, 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. See the description of functions on page 4-1 for more details.

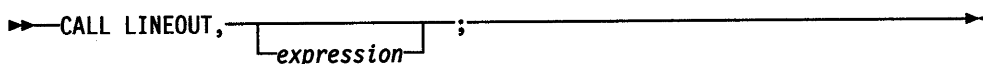
If a PROCEDURE instruction was executed within the routine (subroutine or internal function), all variables of the current generation are dropped (and those of the previous generation are exposed) after *expression* is evaluated and before the result is used or assigned to RESULT.

SAY



SAY writes to the output stream the result of evaluating *expression*. This typically displays the result to the user, but the output destination can depend on the implementation. The result of *expression* may be of any length. If you omit *expression*, the null string is written.

The SAY instruction is a shorter form of the instruction:



except that SAY does not affect the special variable RESULT and SAY does not close the stream if you omit *expression*. See page 4-27 for details of the LINEOUT function.

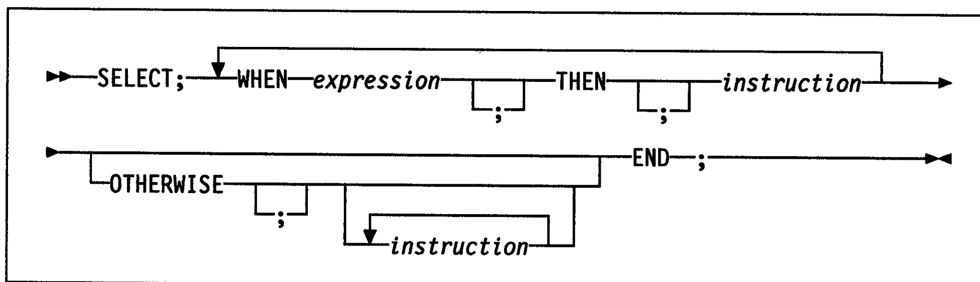
Notes:

1. Data from the SAY instruction is sent to the default output stream (STDOUT). However, the standard OS/2 rules for redirecting output apply to SAY output.
2. 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
/* Displays: "100 divided by 4 => 25"  */
```

SELECT



SELECT conditionally executes one of several alternative instructions.

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

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

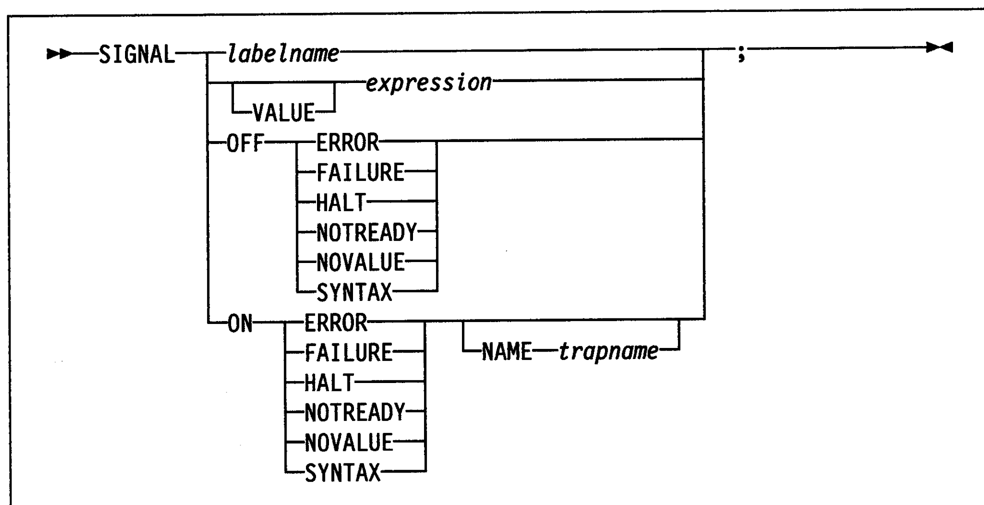
Example:

```
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 do not have any"
       say "checks left outstanding."
       end
  Otherwise
       say "You have just overdrawn your account."
       say "Your balance now shows" balance "dollars."
       say "Oops!  Hope the bank does not close your account."
end  /* Select */
```

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, IF, or the SELECT instruction itself.
2. A null clause is not an instruction, so putting an extra semicolon after a WHEN clause is not equivalent to putting a dummy instruction. The NOP instruction is provided for this purpose.
3. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the WHEN clause to be terminated by the THEN without a ; (delimiter) being required.

SIGNAL



SIGNAL causes an *abnormal* change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions (if you specify ON or OFF).

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, "Conditions and Condition Traps" on page 7-1 .

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, which is treated literally, or a literal string that is taken as a constant. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or literal string (that is, if it starts with a special character, such as an operator or parenthesis). All active pending DO, IF, SELECT, and INTERPRET instructions in the current routine are then terminated (that is, they cannot be resumed). 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 you can use SIGL to determine the source of a jump to a label.

For information about using SIGNAL with the INTERPRET instruction, see Note 1 on page 3-17.

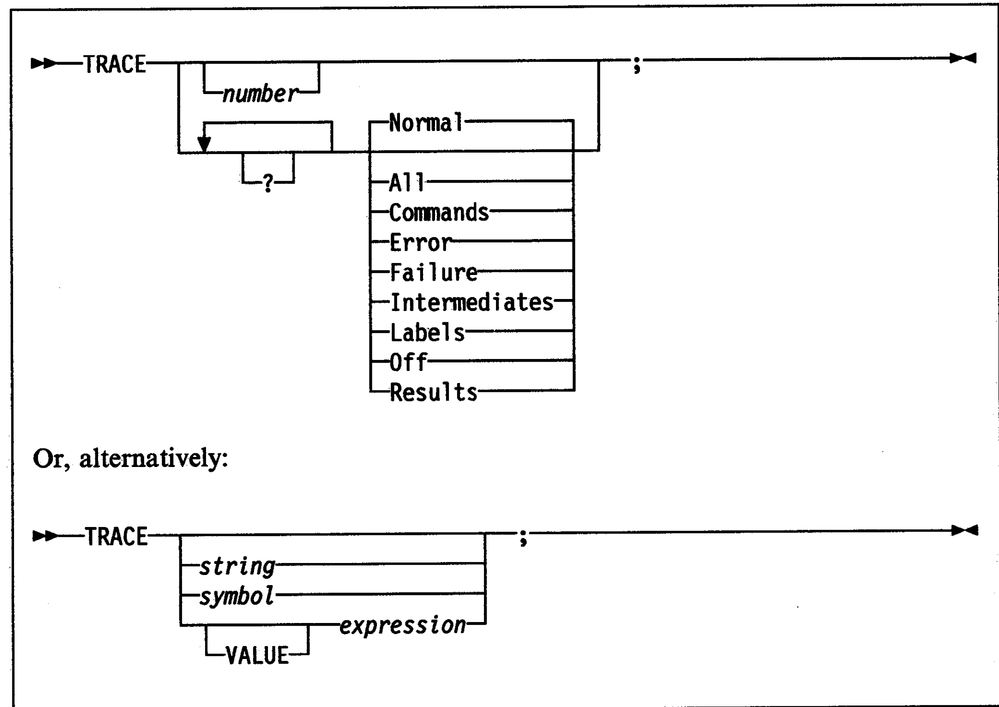
Using SIGNAL VALUE

The VALUE form of the SIGNAL instruction allows a branch to a label whose name is determined at the time of execution. This can safely effect a multi-way CALL (or function call) to internal routines because any DO loops, and so forth, in the calling routine are protected against termination by the call mechanism.

Example:

```
fred='pete'
call multiway fred, 7
   ....
   ....
Multiway: procedure
   arg label .             /* One word, upper case             */
                           /* Can add checks for valid labels here */
   signal value label      /* Jump to wherever                 */
   ....
Pete: say arg(1) '!' arg(2) /* Displays: "pete ! 7"            */
   return
```

TRACE



TRACE is primarily 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 that of other REXX instructions. The economy of key strokes for this instruction is especially convenient since TRACE is usually entered manually during interactive debugging.

If specified, the *number* must be a whole number.

The *string* or *expression* evaluates to:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later
- Null.

The *symbol* is taken as a constant, and is, therefore:

- A numeric option
- One of the valid prefix or alphabetic character (word) options described later.

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. 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 you can enter the word in full, only the capitalized and boldfaced letter is needed; all characters following it are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All All clauses are traced (that is, displayed) before execution.

Commands All commands are traced before execution, and any error return code is displayed.

Error Any command resulting in an error or failure is traced after execution, together with the return code from the command.

Failure Any 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 command is traced after execution. **This is the default setting.**

For the default OS/2 command processor, an attempt to issue an unknown command will raise a FAILURE condition. The CMD return code for an unknown command is 1041. An attempt to issue a command to an unknown command environment will also raise a FAILURE condition; in such a case, the variable RC is set to 30.

Off Nothing is traced, and the special prefix actions (described later) are reset to OFF.

Results All clauses are traced before execution. Final results (contrast with `Intermediates`, preceding) 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 as follows:

? Controls interactive debug. During normal execution, a TRACE option prefixed with ? causes interactive debug to be switched on. (See the separate section in Chapter 10, "Debugging Aids" on page 10-1 for full details of this facility). While interactive debug is on, interpretation pauses after most clauses that are traced. For example, the instruction `TRACE ?E` makes the language processor pause for input after executing any command that returns an error (that is, a nonzero return code or explicit setting of the error condition by the command handler).

Any TRACE instructions in the file being traced are ignored. (This is so that you are not taken out of interactive debug unexpectedly.)

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, you can turn off interactive debug at any time by issuing `TRACE 0` 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. (See separate section in Chapter 10, "Debugging Aids" on page 10-1, for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, `TRACE -100` means that the next 100 clauses that would normally be traced are not, in fact, displayed. After that, tracing resumes as before.

Tracing Tips

1. When a loop is being traced, the DO clause itself is traced on every iteration of the loop.
2. If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N and interactive debug (?) off.
3. You can retrieve the trace actions currently in effect by using the TRACE built-in function ("TRACE" on page 4-40).
4. Comments in the source REXX program are not included in the trace output.
5. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.
6. Trace actions are automatically saved across subroutine and function calls. See the CALL instruction (page 3-6) for more details.

A Typical Example

One of the most common traces you will use is:

```
TRACE ?R
/* Interactive debug is switched on if it was off, */
/*  and tracing Results of expressions begins.     */
```

Format of TRACE Output

Every clause traced is displayed with automatic formatting (indentation) according to its logical depth of nesting and so forth. Results (if requested) are indented an extra two spaces and are enclosed in double quotes so that leading and trailing blanks are apparent. Any control codes in the data encoding (ASCII values less than '20'x) are replaced by a question mark (?) to avoid screen interference.

All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

- Identifies the source of a single clause, that is, the data actually in the program.

+++ Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).

>>> Identifies the result of an expression (for `TRACE R`) or the value assigned to a variable during parsing, or the value returned from a subroutine call.

>.> Identifies the value "assigned" to a placeholder during parsing (see page 5-2).

The following prefixes are only used if `TRACE Intermediates` is in effect:

>C> The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

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

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

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

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

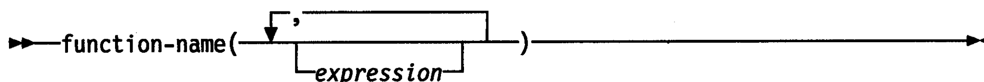
>V> The data traced is the contents of a variable.

Following a syntax error that SIGNAL ON SYNTAX does not trap, the clause in error is always traced. If an attempt to transfer control to a label that could not be found caused the error, that label is also traced.

Chapter 4. Functions

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` is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. On the OS/2 operating system, the implementation maximum is up to 20 expressions. These expressions are called the **arguments** to the function. Each argument expression may include further function calls.

Note that the (must be adjacent to the name of the function, with no blank in between, or the construct is not recognized as a function call. (A **blank operator** would be assumed at this point instead.) Only a comment (which has no effect) can appear between the name and the left parenthesis.

The arguments are evaluated in turn from left to right and 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 just 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 (see page 4-37) and could be used as:

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

A function call without any arguments must always include the parentheses; otherwise it would not be recognized as a function call.

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

Calls to Functions and Subroutines

The function calling mechanism is identical with that for subroutines. The only difference between functions and subroutines is that functions must return data, whereas subroutines need not.

The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine invoked by the CALL instruction, various other status information (TRACE and NUMERIC settings and so forth) is saved too. See the CALL instruction (page 3-6) for details about this. You can use SIGNAL and CALL together to call an internal routine whose name is determined at the time of execution; this is known as a multi-way call (see page 3-36).

If you are calling an internal routine as a function, you *must* specify an expression in any RETURN instruction to return from it. This is not necessary if it is called 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 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 in the next section of this manual. (See pages 4-5—4-46.)

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

Notes:

1. Calling an external REXX program as a function is similar to calling an internal routine. The external routine is, however, an implicit PROCEDURE in that all the caller's variables are always hidden and the status of internal values (NUMERIC settings and so forth) start with their defaults (rather than inheriting those of the caller).
2. Other REXX programs can be called as functions. You can use either EXIT or RETURN to leave the invoked REXX program, and in either case you must specify an expression.

Search Order

The search order for functions is: internal routines take precedence, then built-in functions, and finally external functions.

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

Example:

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

Built-in functions have uppercase names, and so the name in the literal string must be in uppercase for the search to succeed, as in the example. On the OS/2 operating system, file names may be in upper, lower, or mixed case. That operating system uses a case insensitive search for files. When calling a REXX subroutine, the case of the name does not matter.

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

REXX searches for external functions in this order.

1. Functions that have been loaded into the macrospace for pre-order execution; see "Macrospace Interface" on page 9-56.
2. Functions that are part of a function package; see "External Functions" on page 9-18.
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.

The full search pattern for functions and routines is shown in Figure 4-1 on page 4-4.

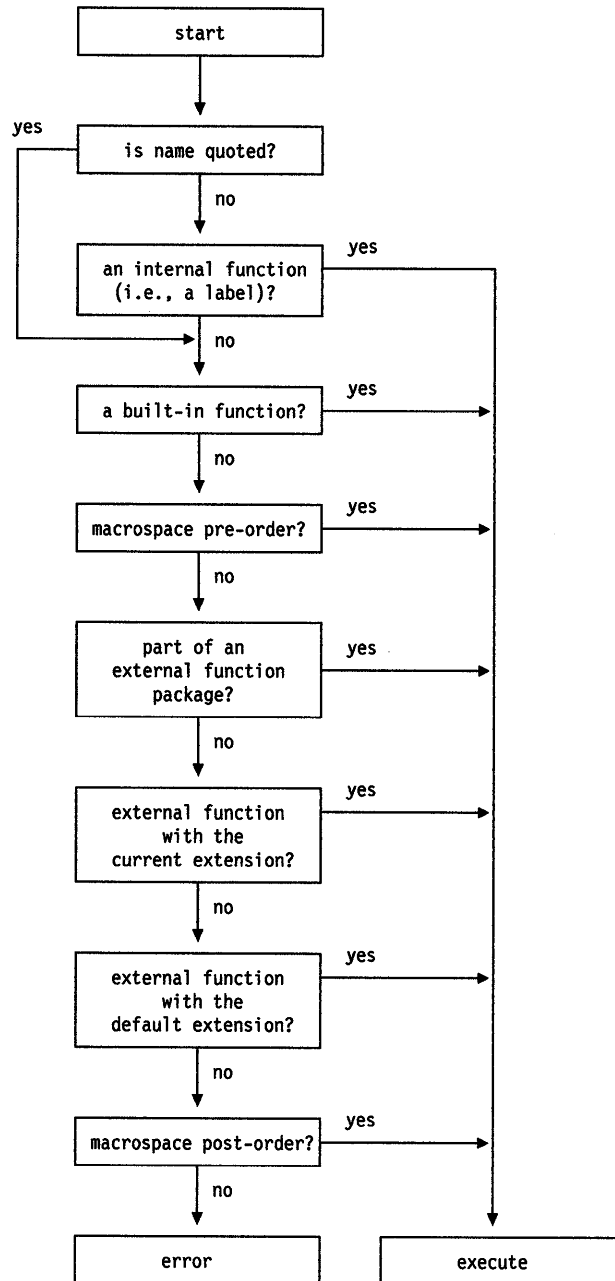


Figure 4-1. Function and Routine Resolution and Execution

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. Execution of the clause that included the function call is, therefore, terminated. Similarly, if an external function fails to return data correctly, the language processor detects this and reports it as an error.

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

Here are some examples of how to call a REXX procedure:

```
call Beep 500, 100         /* Example 1: a subroutine call */
```

The built-in function BEEP is called as a REXX subroutine. The return value from BEEP is placed in the REXX special variable RESULT.

```
bc = Beep(500, 100)        /* Example 2: a function call   */
```

BEEP is 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.

```
Beep(500, 100)             /* Example 3: result passed as  */
                           /*            a command         */
```

The BEEP function is executed and its return value is substituted in the expression for the function call, just as in the preceding example. 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 the third example is passed on to the current environment or subcommand handler. If that environment is CMD (the default), then this action will result in the OS/2 program performing a disk search for what seems to be a command.

Built-in Functions

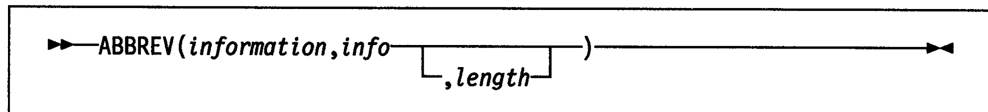
REXX provides a rich set of built-in functions. These include character manipulation, conversion, and information functions.

General notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.

- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated. Any argument named as a `number` is rounded, if necessary, according to the current setting of NUMERIC DIGITS (just as though the number had been added to 0) and checked for validity before use. This occurs in the following functions: ABS, FORMAT, MAX, MIN, SIGN, and TRUNC, and for certain options of DATATYPE.
- Any argument named as a `string` may be a null string.
- 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 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 an option you can select by specifying the first character of a string, that character can be in upper- or lowercase.
- Conversion between characters and hexadecimal involves the machine representation of character strings, and returns appropriately different results for ASCII and EBCDIC machines.
- A number of the functions described in this chapter support DBCS. A complete list and description of these functions is given in Appendix B, "Double-Byte Character Set (DBCS) Support" on page B-1.

ABBREV (Abbreviation)



returns 1 if *info* is equal to the leading characters of *information and* the length of *info* is not less than *length*. Returns 0 if either of these conditions is not met.

If you specify *length*, it must be a nonnegative whole number. The default for *length* is the number of characters in *info*.

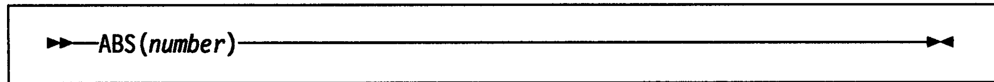
Here are some examples:

```
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 always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired; for example:

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

ABS (Absolute Value)

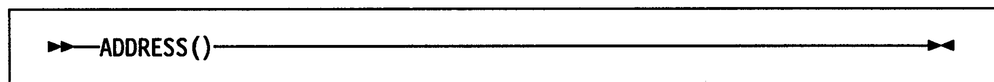


returns the absolute value of *number*. The result has no sign and is formatted according to the current NUMERIC settings.

Here are some examples:

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

ADDRESS

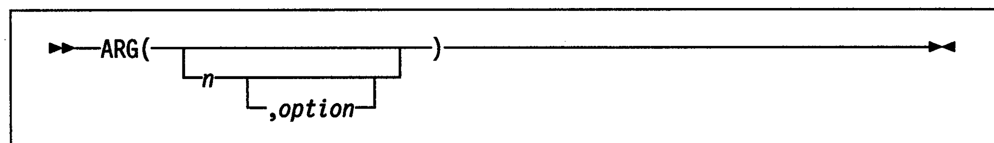


returns the name of the environment to which commands are currently being submitted. Trailing blanks are removed from the result.

Here are some examples:

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

ARG (Argument)



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

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

If you specify only *n*, the *n*th argument string is returned. If the argument string does not exist, the null string is returned. *n* must be a positive whole number.

If you specify *option*, ARG tests for the existence of the *n*th argument string. The following are valid *option*s. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Exists
: returns 1 if the *n*th argument exists; that is, if it was explicitly specified when the routine was called. Returns 0 otherwise.

Omitted returns 1 if the *n*th argument was omitted; that is, if it was *not* explicitly specified when the routine was called. Returns 0 otherwise.

Here are some examples:

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

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

Notes:

1. The number of argument strings is the largest number *n* for which `ARG(n,'e')` would return 1. That is, it is the position of the last explicitly specified argument string.
2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1 argument string if anything else (including blanks) is included with the command.
3. Programs called by the REXSAA entry point (see 9-4) may have multiple argument strings.
4. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See pages 3-4, 3-25, and 5-1.)

BEEP
(Non-SAA Function)

BEEP is a non-SAA built-in function provided by the OS/2 operating system.

```
►►──BEEP(frequency,duration)──────────────────────────────►◄
```

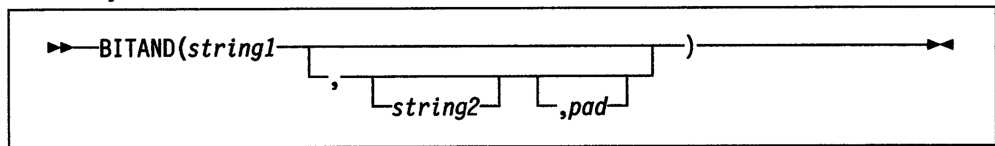
sounds the speaker at *frequency* (Hertz) for *duration* (milliseconds). The *frequency* can be any whole number in the range 37 to 32767 Hertz. The *duration* can be any number in the range 1 to 60000 milliseconds.

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

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 = 523    /*    C     */

do i=1 to 8
  call beep note.i,250      /* hold each note for */
                            /* one-quarter second */
end
```

BITAND (Bit by Bit AND)

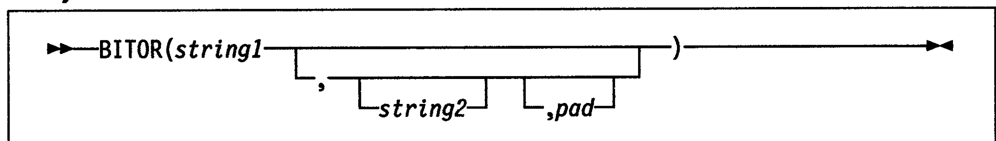


returns a string composed of the two input strings logically ANDed together, bit by bit. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion 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('12'x)                 ->    '12'x
BITAND('73'x,'27'x)           ->    '23'x
BITAND('13'x,'5555'x)         ->    '1155'x
BITAND('13'x,'5555'x,'74'x)   ->    '1154'x
BITAND('pQrS',,'DF'x)         ->    'PQRS' /* ASCII only  */
```

BITOR (Bit by Bit OR)

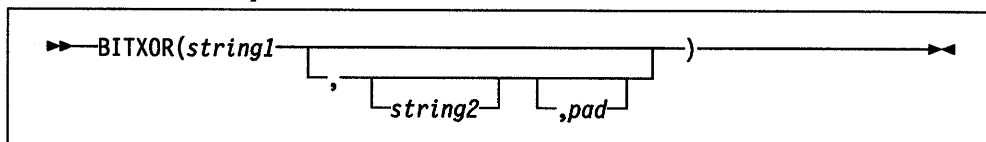


returns a string composed of the two input strings logically ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation terminates when the shorter of the two strings is exhausted, and the unprocessed portion 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('12'x)                  ->    '12'x
BITOR('15'x,'24'x)            ->    '35'x
BITOR('15'x,'2456'x)          ->    '3556'x
BITOR('15'x,'2456'x,'F0'x)    ->    '35F6'x
BITOR('1111'x,,'4D'x)         ->    '5D5D'x
BITOR('pQrS',,'20'x)          ->    'pqrs' /* ASCII only  */
```

BITXOR (Bit by Bit Exclusive OR)

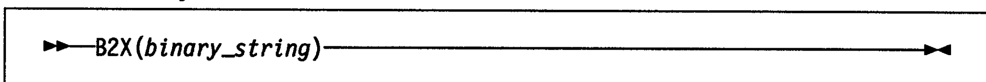


returns a string composed of the two input strings logically eXclusive ORed together, bit by bit. The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the XOR operation terminates when the shorter of the two strings is exhausted, and the unprocessed 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)                      -> '12'x
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
```

B2X (Binary to Hexadecimal)



returns a string, in character format, that represents *binary_string* converted to hexadecimal.

The *binary_string* is a string, of any length, of binary (0 or 1) digits. You can optionally include blanks in *binary_string* (at four-digit boundaries only, not leading or trailing) to aid readability; they are ignored.

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

If *binary_string* is null, B2X returns a null string. If the number of binary digits in *binary_string* is not a multiple of four, then up to three 0 digits are added on the left before the conversion to make a total that is a multiple of four.

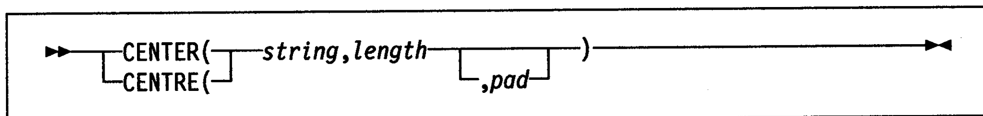
Here are some examples:

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

You can combine B2X with the functions X2D and X2C to convert a binary number into other forms. For example:

```
X2D(B2X('10111'))  ->   '23'   /* decimal 23 */
```

CENTER/CENTRE



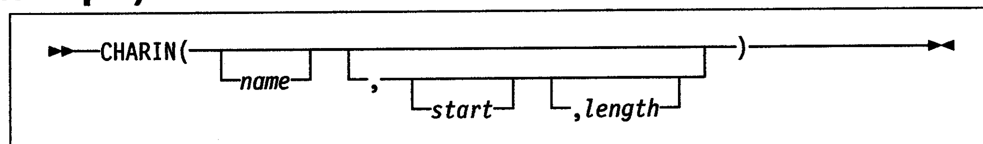
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The default *pad* character is blank. If the string is longer than *length,* it is truncated at both ends to fit. If an odd number of characters are truncated or added, the right-hand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

Here are some examples:

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

Note: To avoid errors because of the difference between British and American spellings, this function can be called either CENTRE or CENTER.

CHARIN (Character Input)



returns a string of up to *length* characters read from the character input stream *name*. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) If you omit *name*, characters are read from the device named STDIN, which is the default input stream. The default *length* is 1.

For persistent streams, a read position is maintained for each stream. On OS/2, this is the same as the write position. Any read from the stream starts at the current read position by default. When the read is completed, the read position is increased by the number of characters read. You can give a *start* value 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 value of 1 for *start* refers to the first character in the stream.

If you specify a *length* of 0, then the read position is 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 than *length* characters available, then execution of the program normally stops 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 (see "Errors During Input and Output" on page 8-6) and CHARIN returns with fewer than the requested number of characters.

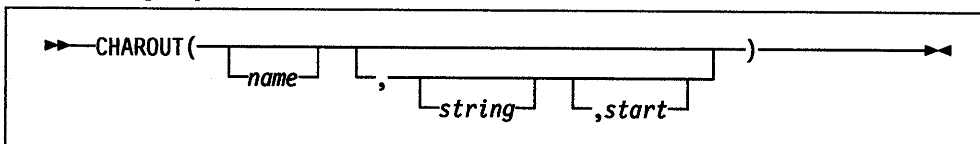
Here are some examples:

```
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'
```

Notes:

1. CHARIN returns all characters that appear in the stream, including control characters such as line feed, carriage return, and end of file.
2. When CHARIN is used to read from the keyboard, program execution stops until you press the Enter key.

CHAROUT (Character Output)



returns the count of characters remaining after attempting to write *string* to the character output stream *name*. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) If you omit *name*, characters in *string* are written to the device STDOUT (normally the display), which is the default output stream. *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. On OS/2, 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 usually append to the end of the stream.

You can give a *start* value 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.

You can omit the *string* 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 you do not specify *start* or *string*, the stream is closed. Again, 0 is returned.

Execution 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 (see "Errors During Input and Output" on page 8-6) and CHAROUT returns with the number of characters that could not be written (the residual count).

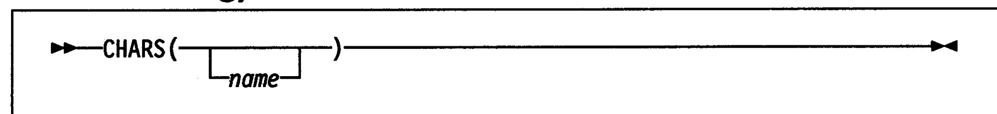
Here are some examples:

```
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
```

CHARS (Characters Remaining)



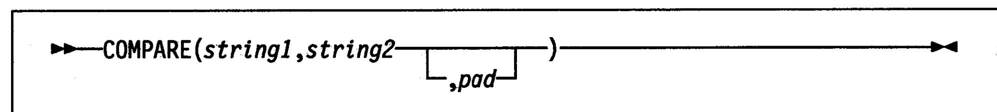
returns the total number of characters remaining in the character input stream *name*. The count includes any line separator characters, if these are defined for the stream, and in the case of persistent streams is the count of characters from the current read position. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) If you omit *name*, the number of characters available in the default input stream (STDIN) is returned.

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. For OS/2 devices, CHARS always returns 1.

Here are some examples:

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

COMPARE

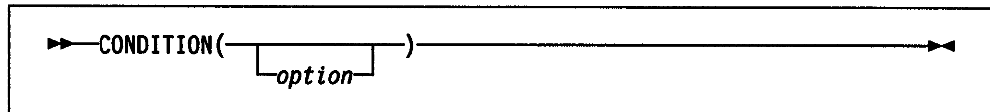


returns 0 if the strings, *string1* and *string2*, are identical. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with *pad* if necessary. The default *pad* character is a blank.

Here are some examples:

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

CONDITION



returns the condition information associated with the current trapped condition. (See Chapter 7, "Conditions and Condition Traps" on page 7-1 for a description of condition traps.) You can request four pieces of information:

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

Request this information by using the following *options*. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Condition name — returns the name of the current trapped condition.

Description — returns any descriptive string associated with the current trapped condition. See page 7-4 for the list of possible strings. If no description is available, returns a null string.

Instruction — returns either CALL or SIGNAL, the keyword for the instruction processed when the current condition was trapped. This is the default if you omit *option*.

Status — returns the status of the current trapped condition. This can change during processing, and is either:

ON - the condition is enabled

OFF - the condition is disabled

DELAY - any new occurrence of the condition is delayed or ignored.

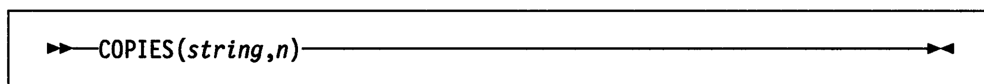
If no condition has been trapped (that is, there is no current trapped condition), then the CONDITION function returns a null string in all four cases.

Here are some examples:

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

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

COPIES

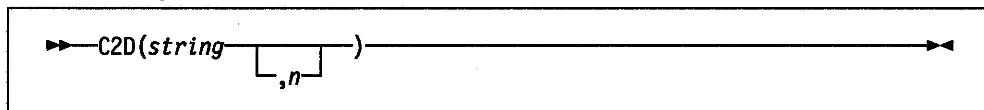


returns *n* concatenated copies of *string*. *n* must be a nonnegative whole number.

Here are some examples:

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

C2D (Character to Decimal)



returns the decimal value of the binary representation of *string*. If the result cannot be expressed as a whole number, an error results. That is, the result must not have more digits than the current setting of NUMERIC DIGITS. If you specify *n*, it is the length of the returned result. If you do not specify *n*, *string* is processed as an unsigned binary number.

If *string* is null, returns 0.

Here are some examples:

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

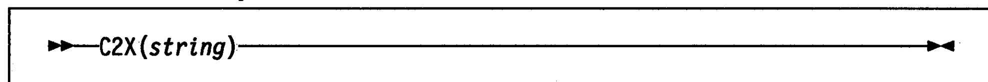
If you specify *n*, the string is taken as a signed number expressed in *n* characters. The number is positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. The *string* is padded on the left with '00'x characters (note, not "sign-extended"), or truncated on the left to *n* characters. If *n* is 0, C2D always returns 0.

Here are some examples:

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

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

C2X (Character to Hexadecimal)



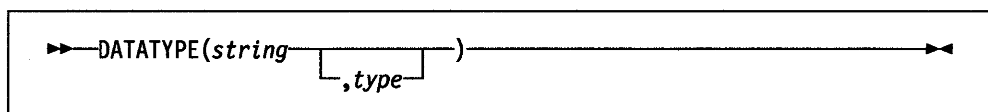
returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string. On an ASCII system, C2X(1) returns 31 because the ASCII representation of the character 1 is '31'X.

The string returned uses uppercase alphabetics for the values A – F and does not include blanks. If *string* is null, returns a null string. The *string* can be of any length.

Here are some examples:

```
C2X('0123'X)    ->    '0123'   /* '30313233'X     in ASCII */
C2X('ZD8')      ->    '5A4438' /* '354134343338'X in ASCII */
```

DATATYPE



returns NUM if you specify only *string* and if *string* is a valid REXX number (any format) that can be added to 0 without error; returns CHAR if *string* is not a valid number.

If you specify *type*, returns 1 if *string* matches the type; otherwise returns 0. If *string* is null, returns 0 (except when *type* is X, which returns 1). The following are valid *types*. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Alphanumeric — returns 1 if *string* contains only characters from the ranges a – z, A – Z, and 0 – 9.

Binary — returns 1 if *string* contains only the characters 0 or 1 or both.

C — returns 1 if *string* is a mixed SBCS/DBCS string.

Dbcs — returns 1 if *string* is a 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 DATATYPE(string) would return NUM.

Symbol returns 1 if *string* contains only characters that are valid in REXX symbols (see page 2-4). Note that both uppercase and lowercase alphabetics are permitted.

Uppercase returns 1 if *string* contains only characters from the range A – Z.

Whole number returns 1 if *string* is a REXX whole number under the current setting of NUMERIC DIGITS.

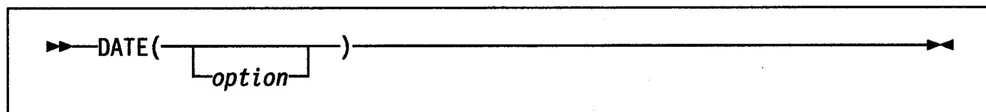
heXadecimal returns 1 if *string* contains only characters from the ranges a – f, A – F, 0 – 9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if *string* is a null string.

Here are some examples:

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

Note: The DATATYPE function tests the meaning or type of characters in a string, independent of the encoding of those characters (for example, ASCII or EBCDIC and so forth).

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 are used.

You can use the following *options* to obtain specific formats. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Base returns the number of complete days (that is, not including the current day) since and including the base date, 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 1582, this base date 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. On the OS/2 operating system, the Language format is dd Month yyyy. The name of the month will be according to the national language installed on the system. If no local date 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 format: dd mon yyyy. **This is the default.**

Ordered returns date in the format: yy/mm/dd (suitable for sorting, and so forth)

Standard returns date in the format: yyyymmdd (suitable for sorting, and so forth)

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

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

Here are some examples:

```
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'
```

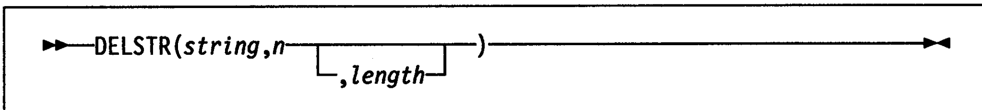
Note: The first call to DATE or TIME in one clause causes a time stamp to be made that is then used for *all* calls to these functions in that clause. Therefore, multiple calls to any of the DATE or TIME functions or both in a single expression or clause are guaranteed to be consistent with each other.

DBCS (Double-Byte Character Set Functions)

The following are all DBCS processing functions. See page B-1.

DBADJUST	DBRIGHT	DBTOSBCS
DBBRACKET	DBRLEFT	DBUNBRACKET
DBCENTER	DBRRIGHT	DBVALIDATE
DBLEFT	DBTODBCS	DBWIDTH

DELSTR (Delete String)

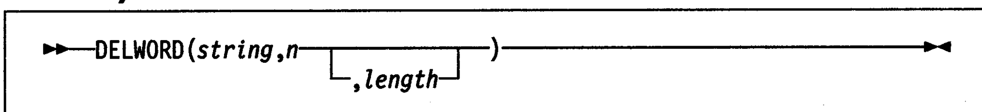


returns *string* after deleting *length* characters beginning at the *n*th character. If you omit *length*, it defaults to the remaining characters in *string*. If *n* is greater than the length of *string*, returns *string* unchanged. *n* must be a positive whole number.

Here are some examples:

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

DELWORD (Delete Word)

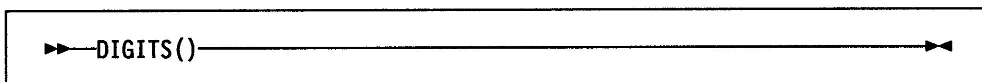


returns *string* after deleting *length* blank-delimited words, beginning at the *n*th word. If you omit *length*, it defaults to the remaining words in *string*. *n* must be a positive whole number. If *n* is greater than the number of words in *string*, returns *string* unchanged. The string deleted 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'
```

DIGITS



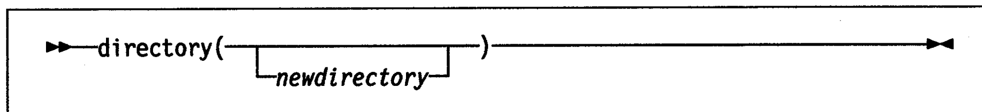
returns the current setting of NUMERIC DIGITS.

Here is an example:

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

DIRECTORY
(Non-SAA Function)

DIRECTORY is a non-SAA built-in function provided by the OS/2 operating system.



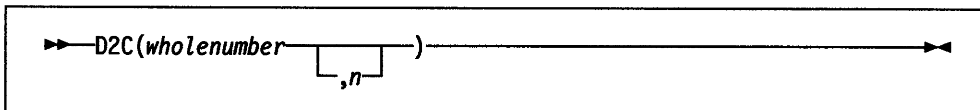
returns the current directory, first changing it to *newdirectory* if an argument is supplied and the named directory exists. If *new directory* is not specified, the name of the current directory is returned. Otherwise, an attempt is made to change to the specified *newdirectory*. If successful, the name of the *newdirectory* is returned; if an error occurred, null is returned.

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'
```

D2C (Decimal to Character)



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

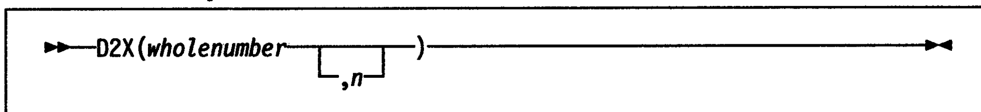
If you omit *n*, *wholenumber* must be a nonnegative number and the result length is as needed; therefore, the returned result has no leading '00'x characters.

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'       /* '93'x is an ASCII 'ô'    */
D2C(76,2)     ->   ' L'      /* '4C'x  is an ASCII ' L'  */
D2C(-180,2)   ->   ' L'
```

Implementation maximum: The output string may not have more than 250 significant characters, though a longer result is possible if it has additional leading sign characters ('00'x and 'FF'x).

D2X (Decimal to Hexadecimal)



returns a string, in character format, that represents *wholenumber*, a decimal number, converted to hexadecimal. The returned string uses uppercase alphabetics for the values A–F and does not include blanks.

If you specify *n*, it is the length of the final result in characters. If you specify *n*, after conversion the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left.

If you omit *n*, *wholenumber* must be a nonnegative number and the returned result has no leading 0 characters.

Here are some examples:

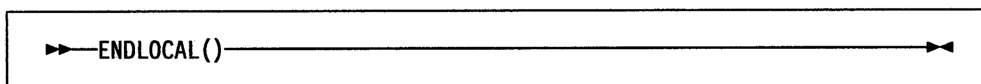
```
D2X(9)          ->    '9'
D2X(129)        ->    '81'
D2X(129,1)      ->    '1'
D2X(129,2)      ->    '81'
D2X(129,4)      ->    '0081'
D2X(257,2)      ->    '01'
D2X(-127,2)     ->    '81'
D2X(-127,4)     ->    'FF81'
D2X(12,0)       ->    ''
```

Implementation maximum: The output string may not have more than 500 significant hexadecimal characters, though a longer result is possible if it has additional leading sign characters (0 and F).

ENDLOCAL

(Non-SAA Function)

ENDLOCAL is a non-SAA built-in function provided by the OS/2 operating system.



restores the drive, directory, and environment variables in effect before the last SETLOCAL function (see page 4-32) 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 action is otherwise unsuccessful.

Note: Unlike their counterparts in the OS/2 batch language (the Setlocal and Endlocal statements), 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, see "SETLOCAL" on page 4-32.

ERRORTEXT

```
►►──ERRORTEXT(n)──────────────────────────────────────────►◄
```

returns the REXX error message associated with error number *n*. The *n* must be in the range 0−99, and any other value is an error. Returns the null string if *n* is in the allowed range but is not a defined REXX error number. See Appendix A, "Error Numbers and Messages" on page A-1 for a complete description of error numbers and messages.

Here are some examples:

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

FILESPEC

(Non-SAA Function)

FILESPEC is a non-SAA built-in function provided by the OS/2 operating system.

```
►►──FILESPEC(option,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*.

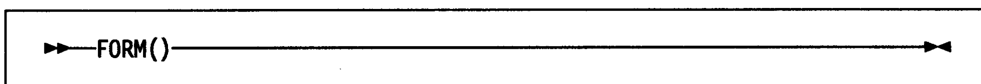
If the requested string is not found, then FILESPEC returns a null string ("").

Note: Only the the initial letter of *option* is needed.

Here are some examples:

```
thisfile = "C:\OS2\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"  */
```

FORM

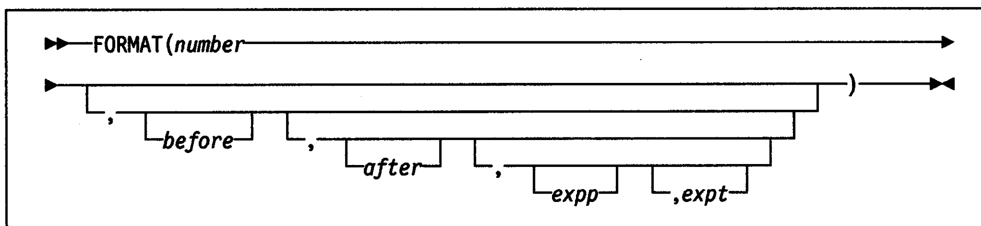


returns the current setting of NUMERIC FORM.

Here is an example:

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

FORMAT



returns *number*, rounded and formatted.

The *number* is first rounded and formatted according to standard REXX rules, just as though the operation `number+0` had been carried out. The result is precisely that of this operation if you specify only *number*. If you specify any other options, the *number* is formatted as follows.

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

If *before* is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If *before* is 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 is rounded (or extended with zeros) to fit. Specifying `0` causes the number to be rounded to an integer.

Here are some examples:

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

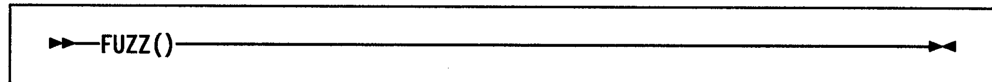
The first three arguments are as described above. In addition, *expp* and *expt* control the exponent part of the result: *expp* sets the number of places for the exponent part; the default is to use as many as needed. The *expt* sets the trigger point for use of exponential notation. If the number of 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 *expt* is `0`, exponential notation is always used unless the exponent would be `0`. If *expp* is `0`, no exponent is supplied, and the

number is expressed in *simple* form with added zeros as necessary (this overrides a 0 value of *expt* if necessary). Otherwise, if *expp* is not large enough to contain the exponent, an error results. If the exponent would be 0 in this case (a nonzero *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'
```

FUZZ

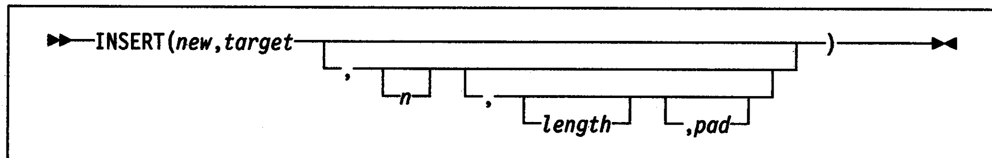


returns the current setting of NUMERIC FUZZ.

Here is an example:

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

INSERT

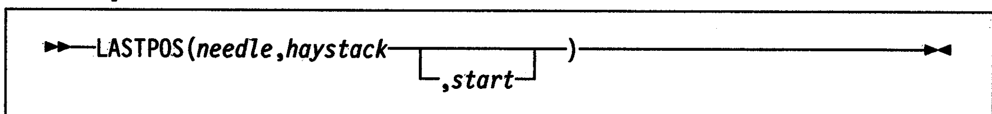


inserts the string *new*, padded to length *length*, into the string *target* after the *n*th character. If specified, *n* must be a nonnegative whole number. If *n* is greater than the length of the target string, padding is added before the string *new* also. The default *pad* character is a blank. 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'
```

LASTPOS (Last Position)



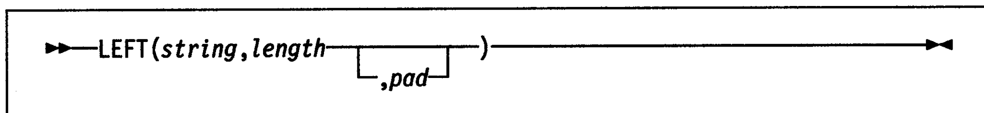
returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns 0 if *needle* is the null string or is not found. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying *start*, the point at which the backward scan

starts. *start* must be a positive whole number and defaults to `LENGTH(haystack)` if larger than that value or omitted.

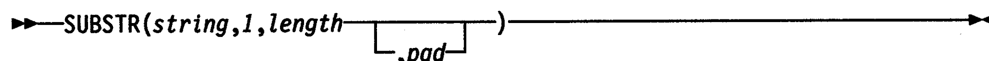
Here are some examples:

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

LEFT



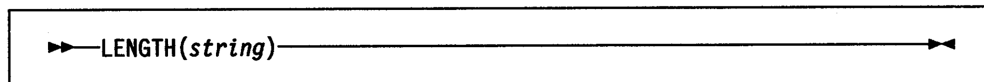
returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be nonnegative. The LEFT function is exactly equivalent to:



Here are some examples:

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

LENGTH

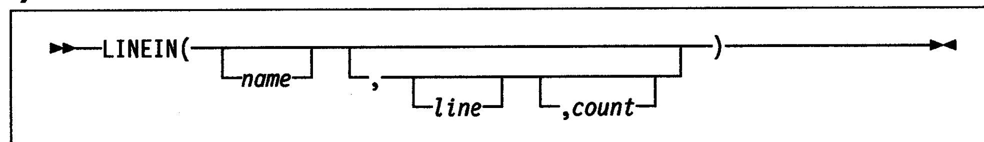


returns the length of *string*.

Here are some examples:

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

LINEIN (Line Input)



returns *count* lines read from the character input stream *name*. *count* must be 1 or `0`. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) See page 8-1 for a list of device names. If you omit *name*, the line is read from the OS/2 default input stream, STDIN. The default *count* is 1.

For persistent streams, a read position is maintained for each stream. On OS/2, this is the same as the write position. Any read from the stream starts at the current

read position by default. (Under certain circumstances, a call to LINEIN returns a partial line. This can happen if the stream has already been read with the CHARIN function, and part but not all of a line (and its termination, if any) has already been read.) 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—the only valid value for *line* on OS/2.

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

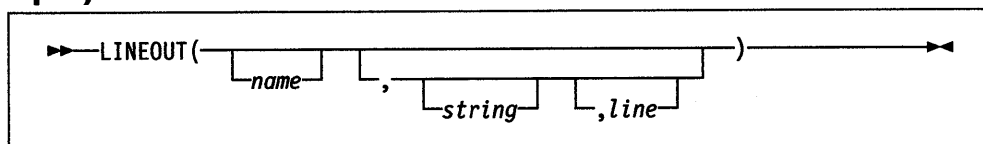
For transient streams, if a complete line is not available in the stream, then execution of the program normally stops until the line is complete. If, however, it is impossible for a line to be completed due to an error or other problem, the NOTREADY condition is raised (see "Errors During Input and Output" on page 8-6) 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 input stream, as in a simple dialogue with a user, use the PULL or PARSE PULL instructions instead for simplicity. The PARSE LINEIN instruction is also useful in certain cases.

LINEOUT (Line Output)



returns the count of lines remaining after attempting to write *string* to the character output stream *name*. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) 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*.

If you omit *name*, the line is written to the OS/2 default output stream, STDOUT (normally the display).

For persistent streams, a write position is maintained for each stream. On the OS/2 operating system, this is the same as the read position. Any write to the stream starts at the current write position by default. (Under certain circumstances the characters written by a call to LINEOUT may be added to a partial line previously written to the stream with the CHAROUT routine. 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 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*.

You can omit the *string* for persistent streams. If you specify *line*, the write position is set to the start of the line that was given, 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 (such as the OS/2 operating system) that support this concept. Again, 0 is returned.

Execution of the program normally stops until the output operation is effectively complete. If, however, it is impossible for a line to be written, the NOTREADY condition is raised (see "Errors During Input and Output" on page 8-6), and LINEOUT returns a result of 1 (that is, the residual count of lines written).

Here are some examples:

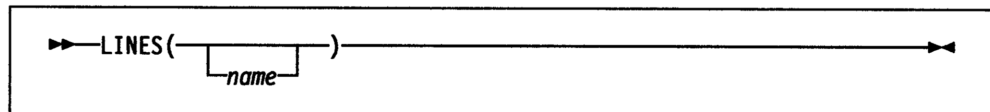
```
LINEOUT(,'Display this')          /* Writes string to the default    */
                                  /* output stream (normally, the    */
                                  /* display); returns 0 if          */
                                  /* successful                      */

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.

LINES (Lines Remaining)



returns 1 if any data remains between the current read position and the end of the character input stream *name*; returns 0 if no data remains. In effect, LINES reports whether a read action that CHARIN (see page 4-11) or LINEIN (see page 4-25) performs will succeed. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.)

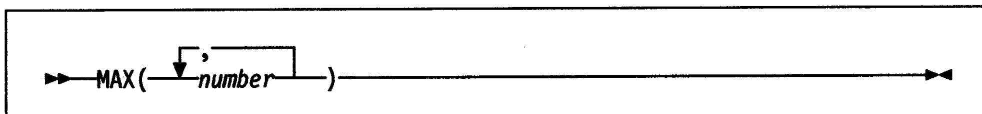
See page 8-1 for a list of device names. If you omit *name*, then the presence or absence of data in the default input stream (STDIN:) is returned. For OS/2 devices, LINES always returns 1. For QUEUE: the actual number of lines is returned.

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.

MAX (Maximum)



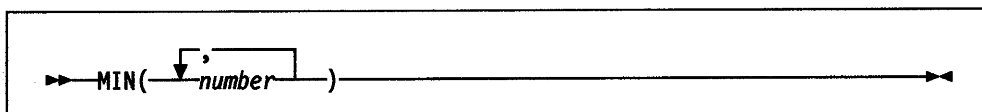
returns the largest number from the list specified, formatted according to the current setting of NUMERIC DIGITS.

Here are some examples:

```
MAX(12,6,7,9)                                                  ->    12
MAX(17.3,19,17.03)                                             ->    19
MAX(-7,-3,-4.3)                                                ->    -3
MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,MAX(20,21)) ->   21
```

Implementation maximum: You can specify up to 20 *numbers*, and can nest calls to MAX if more arguments are needed.

MIN (Minimum)



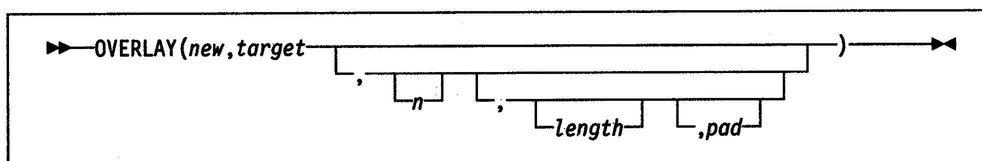
returns the smallest number from the list specified, formatted according to the current setting of NUMERIC DIGITS.

Here are some examples:

```
MIN(12,6,7,9)                                                  ->     6
MIN(17.3,19,17.03)                                             ->    17.03
MIN(-7,-3,-4.3)                                                ->    -7
MIN(21,20,19,18,17,16,15,14,13,12,11,10,9,8,7,6,5,4,3,MIN(2,1)) ->   1
```

Implementation maximum: You can specify up to 20 *numbers*, and can nest calls to MIN if more arguments are needed.

OVERLAY

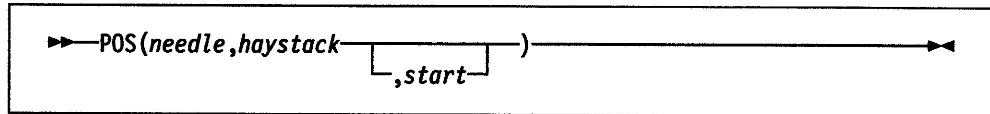


returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. If you specify *length*, it must be positive or 0. The default value for *length* is the length of *new*. If *n* is greater than the length of the target string, padding is added before the *new* string. The default *pad* character is a blank, and the default value for *n* is 1. If you specify *n*, it must be a positive whole number.

Here are some examples:

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

POS (Position)

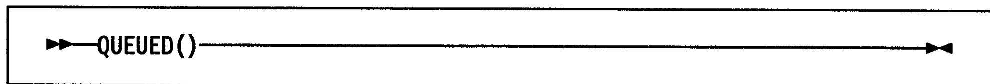


returns the position of one string, *needle*, in another, *haystack*. (See also the LASTPOS function.) Returns 0 if *needle* is the null string or is not found. By default the search starts at the first character of *haystack* (that is, the value of *start* is 1). You can override this by specifying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

Here are some examples:

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

QUEUED

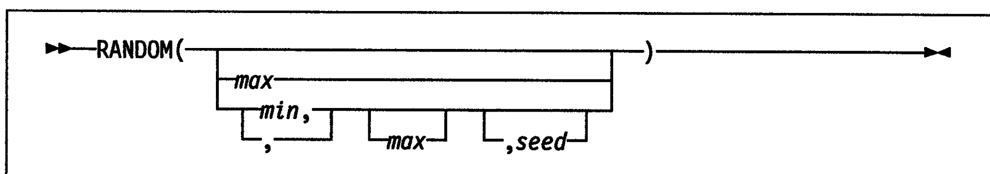


returns the number of lines remaining in the external data queue at the time when the function is invoked. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.)

Here is an example:

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

RANDOM



returns a quasi-random nonnegative whole number in the range *min* to *max* inclusive. If you specify *max* or *min* or both, *max* minus *min* cannot exceed 100000. *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument, as described in Note 1 on page 4-31. This *seed* must be a whole number.

Here are some examples:

```
RANDOM()          ->    305
RANDOM(5,8)       ->      7
RANDOM(2)         ->      0  /*  0  to  2    */
RANDOM(2,)        ->    747  /*  2  to  999  */
RANDOM(,,1983)    ->    123  /* reproducible */
```

Notes:

1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

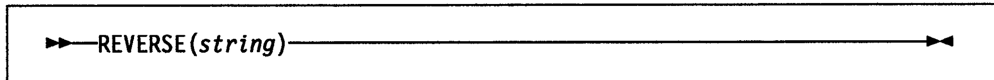
   ```
   sequence = RANDOM(1,6,12345)  /* any number would */
                                 /* do for a seed    */
   do 39
      sequence = sequence RANDOM(1,6)
      end
   say sequence
   ```

   The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence. If you do not supply a *seed*, the first time RANDOM is called, one is randomly assigned; your program usually gives different results each time it is run.

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

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

REVERSE

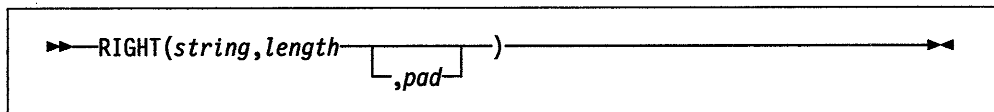


returns *string*, swapped end for end.

Here are some examples:

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

RIGHT



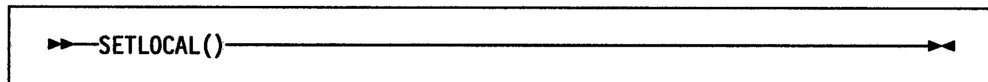
returns a string of length *length* containing the rightmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the left as needed. The default *pad* character is a blank. *length* must be nonnegative.

Here are some examples:

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

SETLOCAL
(Non-SAA Function)

SETLOCAL is a non-SAA built-in function provided by the OS/2 operating system.



SETLOCAL saves the current working drive and directory and the current values of the OS/2 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 (see page 4-41). To restore the drive, directory, and environment, use the ENDLOCAL function (see page 4-21).

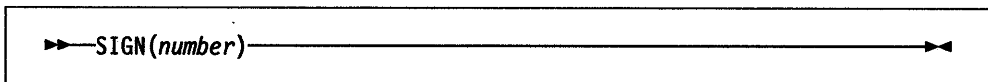
SETLOCAL returns a value of 1 if the initial drive, directory and environment are successfully saved, a value of 0 if unsuccessful. If SETLOCAL is not followed by an ENDLOCAL function in a procedure, then the initial environment saved by SETLOCAL will be restored upon exiting the procedure.

The following 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.

SIGN

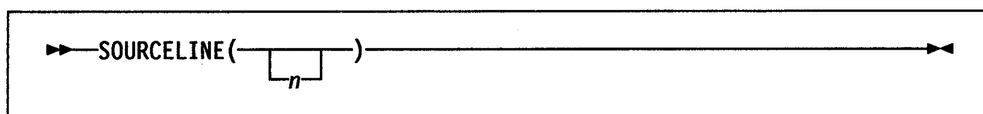


returns a number that indicates the sign of *number*. *number* is first rounded according to standard REXX rules, just as though the operation number+0 had been carried out. Returns -1 if *number* is less than 0; returns 0 if it is 0; and returns 1 if it is greater than 0.

Here are some examples:

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

SOURCELINE

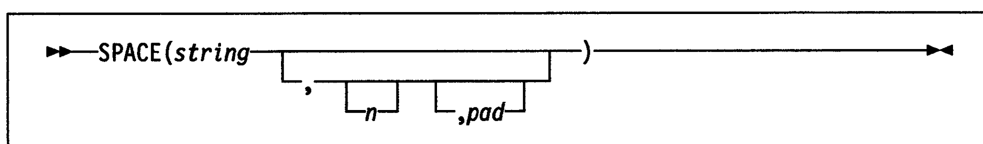


returns the line number of the final line in the source file if you omit *n*, or 0 if no source lines are available. If you specify *n*, returns the *n*th line in the source file if available at the time of execution; otherwise, returns the null string. If specified, *n* must be a positive whole number and must not exceed the number that a call to SOURCELINE with no arguments returns.

Here are some examples:

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

SPACE

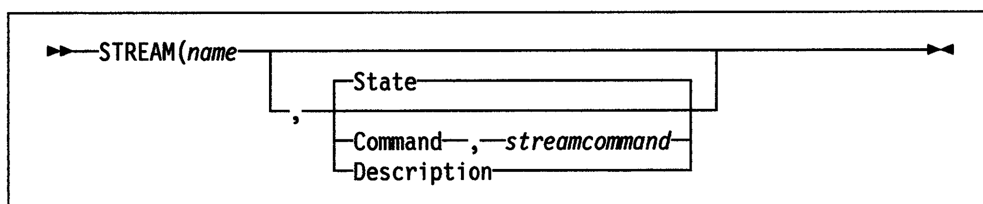


returns the blank-delimited words in *string* with *n pad* characters between each word. If you specify *n*, it must be nonnegative. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1, and the default *pad* character is a blank.

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'
```

STREAM



returns a string describing the state of, or the result of an operation upon, the character stream *name*. (See Chapter 8, "Input and Output Streams" on page 8-1 for a discussion of REXX input and output.) 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), describing the action to be carried out:

`Command` — an operation (specified by the *streamcommand* given as the third argument) is applied to the selected input or output stream. The string that is returned depends on the command performed and may be the null string. The possible input strings for the *streamcommand* argument are described below.

`Description` — 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—see "Errors During Input and Output" on page 8-6). You may be able to obtain additional information about the error by invoking the STREAM function with a request for the 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. (See page 8-6.) 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 may 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. In OS/2 implementations, this generally means that the stream is closed (or has not yet been opened).

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

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 — and only 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.

For most error conditions, the additional information is in the form of a numeric return code. These return codes are defined and set by the OS/2 file system. For information on the meaning of specific codes see the *OS/2 Command Reference*.

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 strea is successfully opened or an appropriate error message if unsuccessful.

  Examples:

  ```
  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 ("").

  Example:

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

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

  Note: On the OS/2 operating system, the read and write positions are the same. (See page 8-1 for a discussion of read and write positions in a persistent stream.) 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 or an appropriate error message otherwise.

Examples:

```
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 or a null string otherwise.

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

A sample output might be:

```
C:\CONFIG.SYS
```

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

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

A sample output might be:

```
1305
```

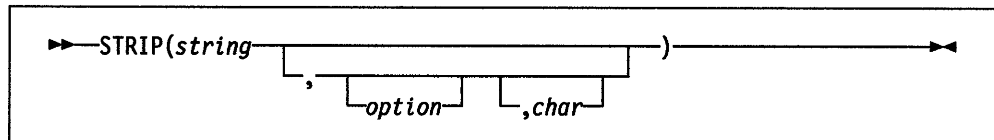
'QUERY DATETIME' returns the date and time stamps of a stream.

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

A sample output might be:

```
12-11-91 03:29:12
```

STRIP



returns *string* with leading or trailing characters or both removed, based on the *option* you specify. The following are valid *options*. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Both removes both leading and trailing characters from *string*. This is the default.

Leading removes leading characters from *string*.

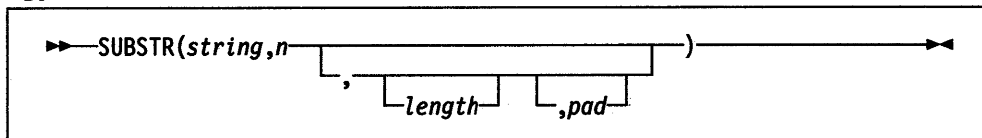
Trailing removes trailing characters from *string*.

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

Here are some examples:

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

SUBSTR (Substring)



returns the substring of *string* that begins at the *n*th character and is of length *length*, padded with *pad* if necessary. *n* must be a positive whole number. If *n* is greater than LENGTH(string), then only pad characters are returned.

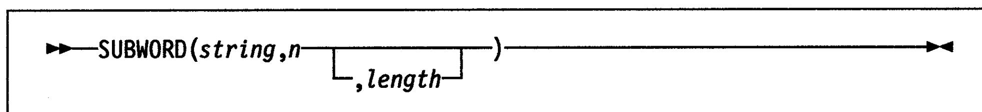
If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

Here are some examples:

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

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

SUBWORD

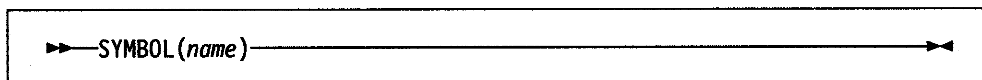


returns the substring of *string* that starts at the *n*th word, and is of length *length*, blank-delimited words. *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes 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)      ->    ''
```

SYMBOL



returns the state of the symbol named by *name*. Returns BAD if *name* is not a valid REXX symbol. Returns VAR if it is the name of a variable (that is, a symbol that has

been assigned a value). Otherwise returns LIT, indicating that it is either a constant symbol or a symbol that has not yet been assigned a value (that is, a literal).

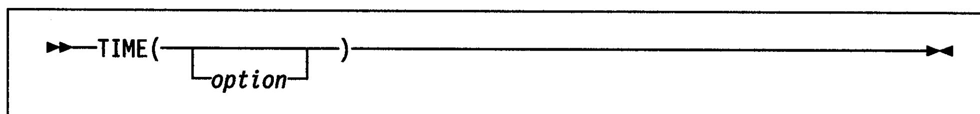
As with symbols in REXX expressions, lowercase characters in *name* are translated to uppercase and substitution in a compound name occurs if possible.

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

Here are some examples:

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

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* to obtain alternative formats, or to gain access to the elapsed-time clock. (Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Civil
: returns the time in Civil format: hh:mmxx. The hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters am or pm to distinguish times in the morning (midnight 12:00am through 11:59am) from noon and afternoon (noon 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.

Elapsed
: returns sssssssss.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.

REXX calculates elapsed time by subtracting the time the elapsed time clock was started or reset from the current time. On the OS/2 operating system, it is possible to change the system time clock while the system is running. This means that the elapsed time value REXX calculates might not be a true elapsed time. If the time is changed so that the system time is earlier than when the REXX elapsed time clock was started (so that the elapsed time would appear negative), REXX raises an error and disables the elapsed time clock. To restart the elapsed time clock, trap the error through SIGNAL ON SYNTAX. To change the system time clock, use the OS/2 TIME command. The clock can also by changed by programs on the system. Many LAN-attached programs synchronize the system time clock with the system time clock of the server during startup. This causes the REXX elapsed time function to be unreliable during LAN initialization.

Hours returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of 0).

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

Minutes returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of 0).

Normal returns the time in the default format hh:mm:ss, as described previously. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59; all these are always two digits. Any fractions of seconds are ignored (times are never rounded up). **This is the default.**

Reset returns sssssssss.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.

Refer to the **Elapsed** option for more information on resetting the system time clock.

Seconds returns up to five characters giving the number of seconds since midnight in the format: sssss (no leading zeros or blanks, except for a result of 0).

Here are some examples:

```
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') return 0.

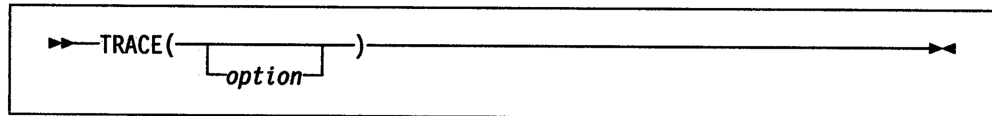
The clock is saved across internal routine calls, which is to say that an internal routine inherits the time clock its caller started. Any timing the caller is doing is not affected, even if an internal routine resets the clock. An example of the elapsed-time clock:

```
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 note under DATE about consistency of times within a single clause. The elapsed-time clock is synchronized to the other calls to TIME and DATE, so multiple calls to the elapsed-time clock in a single clause always return the same result. For the same reason, the interval between two normal TIME/DATE results may be calculated exactly using the elapsed-time clock.

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

TRACE



returns trace actions currently in effect.

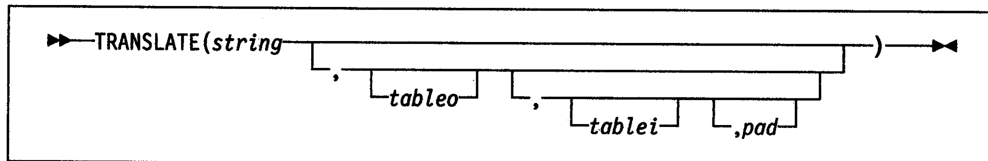
If *option* is supplied, it must be the valid prefix ? or one of the alphabetic character options (that is, starting with A, C, E, F, I, L, N, O, or R) associated with the TRACE instruction or both. (See the TRACE instruction in "TRACE" on page 3-37 for full details.) The function uses *option* to alter the effective trace action (such as tracing Labels, and so forth).

Unlike the TRACE instruction, the TRACE function alters the trace action even if interactive debug is active. Also unlike the TRACE instruction, *option* cannot be a number.

Here are some examples:

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

TRANSLATE



returns *string* with each character translated to another character or unchanged. You can also use this function to reorder the characters in *string*.

The output table is *tableo* and the input translation table is *tablei*. TRANSLATE searches *tablei* for each character in *string*. If the character is found, then the corresponding character in *tableo* is used in the result string; if there are duplicates in *tablei*, the first (leftmost) occurrence is used. If the character is not found, the original character in *string* is used. The result string is always the same length as *string*. The tables can be of any length.

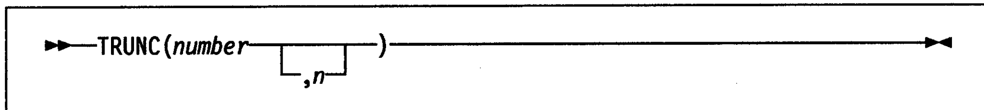
If you specify neither translation table, *string* is simply translated to uppercase (that is, lowercase a–z to uppercase A–Z). Otherwise, *tablei* defaults to XRANGE('00'x,'FF'x), and *tableo* defaults to the null string and is padded with *pad* or truncated as necessary. The default *pad* is a blank.

Here are some examples:

```
TRANSLATE('abcdef')                   ->   'ABCDEF'
TRANSLATE('abbc','&','b')             ->   'a&&c'
TRANSLATE('abcdef','12','ec')         ->   'ab2d1f'
TRANSLATE('abcdef','12','abcd','.')   ->   '12..ef'
TRANSLATE('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.

TRUNC (Truncate)



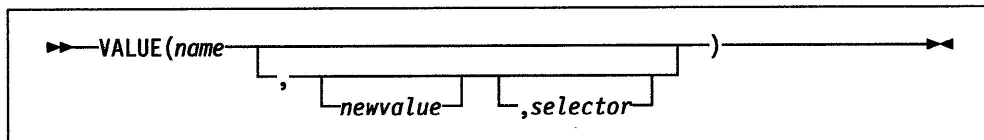
returns the integer part of *number*, and *n* decimal places. The default *n* is 0 and returns an integer with no decimal point. If you specify *n*, it must be a nonnegative whole number. The *number* is first rounded according to standard REXX rules, just as though the operation number+0 had been carried 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 the function processes it.

VALUE



returns the value of the symbol that *name* (often constructed dynamically) represents and optionally assigns it a new value. By default, VALUE refers to the current REXX-variables environment, 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 symbol's derived name (see "Compound Symbols" on page 2-14). 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.

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 */
```

To use the VALUE to manipulate OS/2 environment variables, *selector* must be the string "OS2ENVIRONMENT" or an expression so evaluated. In this case, the variable *name* need not be a valid REXX symbol. When VALUE is used to set or change the value of an environment variable, the new value is retained after the REXX procedure ends.

Restriction: The values assigned to the variables must not contain any character that is a hexadecimal zero ('00'X). For example, `Call VALUE 'MYVAR', 'FIRST' || '00'X || 'SECOND', 'OS2ENVIRONMENT'` sets MYVAR to "FIRST", truncating the '00'x and 'SECOND'.

Examples:

```
/* Given that an external variable FRED has a value of 4       */
share = 'OS2ENVIRONMENT'
say VALUE('fred',7,share)      /* says '4' and assigns         */
                               /* FRED a new value of 7        */

say VALUE('fred',,share)       /* says '7'                     */

/* After this procedure ends, FRED again has a value of 4      */

/* Accessing and changing OS/2 environment entries              */
env = 'OS2ENVIRONMENT'
new = 'C:\LIST\PROD;'
say value('prompt',,env)   /* says '$i[$p]' (perhaps) */       */
say value('path',new,env)  /* says 'C:\EDIT\DOCS;' (perhaps)   */
                           /* and sets PATH = 'C:\LIST\PROD'   */

say value('path',,env)     /* says 'C:\LIST\PROD'              */

/* When this procedure ends, PATH = 'C:\LIST\PROD'             */
```

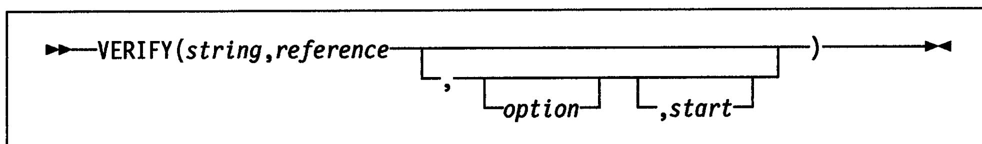
Notes:

1. If the VALUE function refers to an uninitialized REXX variable then the default value of the variable is always returned; the NOVALUE condition is not raised. A reference to an external collection of variables never raises NOVALUE.

2. The VALUE function is used when a variable contains the name of another variable, or when a name is constructed dynamically. If you specify the *name* as a single literal string and omit *newvalue* and *selector*, the symbol is a constant and so the string between the quotation marks can usually replace the whole function call. (For example, `fred=VALUE('k');` is identical with the assignment `fred=k;`, unless the NOVALUE condition is being trapped. (See Chapter 7, "Conditions and Condition Traps" on page 7-1.)

3. To effect *temporary* changes to environment variables, use the SETLOCAL and ENDLOCAL functions.

VERIFY



returns a number that, by default, indicates whether *string* is composed only of characters from *reference*; returns `0` if all characters in *string* are in *reference*, or returns the position of the first character in *string not* in *reference*.

The 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 upper- or lowercase, as usual. If you specify **`Match`**, returns the position of the first character in *string* that *is* in *reference*, or returns `0` if none of the characters are found.

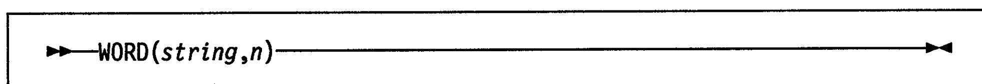
The default for *start* is 1, thus, the search starts at the first character of *string*. You can override this by specifying a different *start* point, which must be a positive whole number.

Always returns `0` if *string* is null, or if *start* is greater than `LENGTH(string)`. If *reference* is null, returns `0` if you specify **`Match`**, otherwise returns `1`.

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
```

WORD

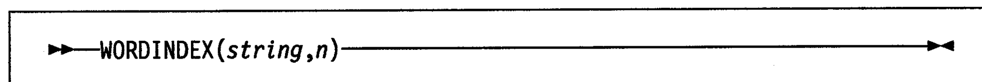


returns the *n*th blank-delimited word in *string* or returns the null string if fewer than *n* words are in *string*. *n* must be a positive whole number. This function is exactly equivalent to `SUBWORD(string,n,1)`.

Here are some examples:

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

WORDINDEX



returns the position of the first character in the *n*th blank-delimited word in *string* or returns `0` if fewer than *n* words are in *string*. *n* must be a positive whole number.

Here are some examples:

```
WORDINDEX('Now is the time',3)    ->    8
WORDINDEX('Now is the time',6)    ->    0
```

WORDLENGTH

```
>>--WORDLENGTH(string,n)-----------------------------------><
```

returns the length of the *n*th blank-delimited word in *string* or returns 0 if fewer than *n* words are in *string*. *n* must be a positive whole number.

Here are some examples:

```
WORDLENGTH('Now is the time',2)       ->    2
WORDLENGTH('Now comes the time',2)    ->    5
WORDLENGTH('Now is the time',6)       ->    0
```

WORDPOS (Word Position)

```
>>--WORDPOS(phrase,string--+--------+--)------------------><
                           '-,start-'
```

returns the word number of the first word of *phrase* found in *string* or returns 0 if *phrase* contains no words or if *phrase* is not found. Multiple blanks between words in either *phrase* or *string* are treated as a single blank for the comparison, but otherwise the words must match exactly.

By default the search starts at the first word in *string*. You can override this by specifying *start* (which must be positive), the word at which to start the search.

Here are some examples:

```
WORDPOS('the','now is the time')              -> 3
WORDPOS('The','now is the time')              -> 0
WORDPOS('is the','now is the time')           -> 2
WORDPOS('is   the','now is the time')         -> 2
WORDPOS('is   time ','now is   the time')     -> 0
WORDPOS('be','To be or not to be')            -> 2
WORDPOS('be','To be or not to be',3)          -> 6
```

WORDS

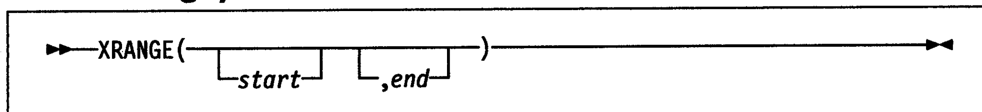
```
>>--WORDS(string)-----------------------------------------><
```

returns the number of blank-delimited words in *string*.

Here are some examples:

```
WORDS('Now is the time')    ->    4
WORDS(' ')                  ->    0
```

XRANGE (Hexadecimal Range)

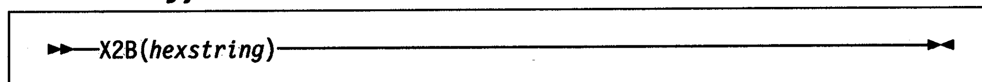


returns a string of all 1-byte codes between and including the values *start* and *end*. The default value for *start* is `'00'x`, and the default value for *end* is `'FF'x`. If *start* is greater than *end*, the values wrap from `'FF'x` to `'00'x`. If specified, *start* and *end* must be single characters.

Here are some examples:

```
XRANGE('a','f')       ->   'abcdef'
XRANGE('03'x,'07'x)   ->   '0304050607'x
XRANGE(,'04'x)        ->   '0001020304'x
XRANGE('FE'x,'02'x)   ->   'FEFF000102'x
XRANGE('i','j')       ->   'ij'                  /* ASCII  */
```

X2B (Hexadecimal to Binary)



returns a string, in character format, that represents *hexstring* converted to binary. The *hexstring* is a string of hexadecimal characters. It can be of any length. Each hexadecimal character is converted to a string of four binary digits. You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

The returned string has a length that is a multiple of four, and does not include any blanks.

If *hexstring* is null, returns a null string.

Here are some examples:

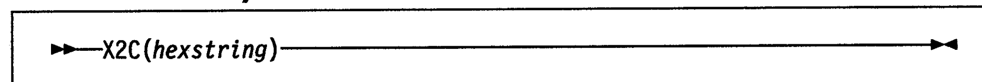
```
X2B('C3')        ->  '11000011'
X2B('7')         ->  '0111'
X2B('1 C1')      ->  '000111000001'
```

You can combine X2B with the functions D2X and C2X to convert numbers or character strings into binary form.

For example:

```
X2B(C2X('C3'x))  ->  '11000011'
X2B(D2X('129'))  ->  '10000001'
X2B(D2X('12'))   ->  '1100'
```

X2C (Hexadecimal to Character)



returns a string, in character format, that represents *hexstring* converted to character. The returned string is half as many bytes as the original *hexstring*. *hexstring* can be

of any length. You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

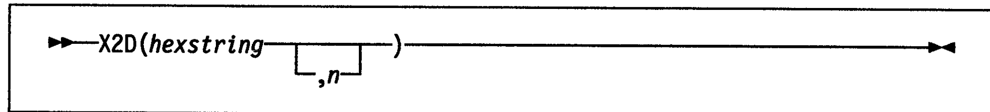
If *hexstring* is null, returns a null string.

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' is unprintable EBCDIC */
```

X2D (Hexadecimal to Decimal)



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

You can optionally add blanks to *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, returns `0`.

If you do not specify *n*, *hexstring* is processed as an unsigned binary number.

Here are some examples:

```
X2D('0E')       ->    14
X2D('81')       ->    129
X2D('F81')      ->    3969
X2D('FF81')     ->    65409
X2D('c6 f0'X)   ->    240
```

If you specify *n*, the given *hexstring* is padded on the left with `0` 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`, 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
```

OS/2 Applications Programming Interface 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.

- See "External Functions" on page 9-18 for a full discussion of the external-function interfaces
- See "Queue Interface" on page 8-4 for a full discussion of applications queuing services.

RXFUNCADD

```
▶▶──RXFUNCADD(name,module,procedure)──────────────────────────────▶◀
```

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

```
rxfuncadd('SysLoadFuncs,'REXXUTIL', 'SysLoadFuncs')     -> 0
                                   /* if not already registered */
```

RXFUNCDROP

```
▶▶──RXFUNCDROP(name)──────────────────────────────────────────────▶◀
```

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

```
rxfuncdrop('SysLoadFuncs')         -> 0 /* if not already registered */
```

RXFUNCQUERY

```
▶▶──RXFUNCQUERY(name)─────────────────────────────────────────────▶◀
```

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.

```
rxfuncquery('SysLoadFuncs')        -> 0 /* if not already registered */
```

Queue Interface

RXQUEUE

```
▶▶──RXQUEUE──(──┬─"Get"──────────────────────┬─)──────────────────▶◀
                ├─"Set"──newqueuename────────┤
                ├─"Delete"──queuename────────┤
                └─"Create"──┬──────────────┬─┘
                            └─,queuename───┘
```

creates and deletes external data queues; also sets and queries their names.

Parameters

Create Creates a queue with the name, queuename (if specified); if no name is specified, then REXX will provide a name. Returns the name of the queue in either case.

Many queues may exist at the same time, and most systems have sufficient resources available to support several hundred queues at once. If a queue name is given and a queue with that name already exists, a queue is still created; but REXX chooses the name and returns the actual name.

Delete Deletes the named queue; returns 0 if successful, a nonzero number if an error occurs; the possible return values are:

0	Queue has been deleted.
5	Not a valid queue name.
9	Queue named does not exist.
10	Queue is busy; wait is active.
12	A memory failure has occurred.
1000	Initialization error; check file OS2.INI.

Get Returns the name of the queue currently in use.

Set Sets the name of the current queue to newqueuename and returns the previous name of the queue.

The first parameter determines the function. Only the first character of the first parameter is significant. The parameter may be entered in any case. The syntax for a valid queue name is the same as for a valid REXX symbol.

The second parameter specified for Create, Set, and Delete must abide by the same syntax rules as for REXX variable names. (There is no connection, however, between queue names and variable names. A program can have a variable and a queue with a common name, and there is no connection between them.) The actual name of the queue is the uppercase value of the name requested.

The queue name must be a valid REXX symbol. However, there is no connection between queue names and variable names. A program can have a variable and a queue with a common name.

Named queues prevent different REXX programs that are running in a single session from interfering with each other. Named queues allow REXX programs running in different sessions to synchronize execution and pass data. LINEOUT('QUEUE:') is especially useful because the calling program will cease execution until another program places a line on the queue.

```
                             /* default queue */
rxqueue('Get')             -> 'SESSION'
                             /* assuming FRED does not already exist */
rxqueue('Create', 'Fred'   -> 'FRED'
                             /* assuming SESSION had been active  */
rxqueue('Set', 'Fred')     -> 'SESSION'
                             /* assuming FRED did not exist */
rxqueue('delete', 'Fred')  -> '0'
```

REXX Utilities (RexxUtil)

RexxUtil is a Dynamic Link Library (DLL) package of OS/2 operating system REXX functions. These operating system functions:

- Manipulate OS/2 operating system files and directories
- Manipulate OS/2 classes and objects.
- Perform text screen input and output.

To use a RexxUtil function, you must first register the function with the REXX RxFuncAdd function:

Add RexxUtil Function

```
call RxFuncAdd 'SysCls', 'RexxUtil', 'SysCls'
```

The example above registers the SysCls function. You can now use the SysCls function in your REXX programs.

The SysLoadFuncs RexxUtil function automatically loads the other RexxUtil functions. The following instructions in a REXX program will register all of the RexxUtil functions.

Load RexxUtil Function

```
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
```

Once registered, the RexxUtil functions are available from all OS/2 operating system sessions. If you use the RexxUtil functions frequently, you should place a call to SysLoadFuncs in your STARTUP.CMD file.

RxMessageBox

action = RxMessageBox(text, [title], [button], [icon])

Displays a Presentation Manager message box. RxMessageBox can only be used from a REXX program running under PMREXX or called from a Presentation Manager application.

Parameters

text The message box text.

title The message box title. The default title is "Error!."

button The message box button style. The allowed styles are:

OK
A single OK button.

OKCANCEL
An OK button and a CANCEL button.

CANCEL
A single CANCEL button.

ENTER
A single ENTER button.

ENTERCANCEL
An ENTER button and a CANCEL button.

RETRYCANCEL
A RETRY button and a CANCEL button.

ABORTRETRYCANCEL
An ABORT button, a RETRY button and a CANCEL button.

YESNO
A YES button and a NO button.

YESNOCANCEL
A YES button, a NO button and a CANCEL button.

The default button style is OK.

icon The message box icon style. The allowed styles are:

NONE
No icon is displayed.

HAND
A hand icon is displayed.

QUESTION
A question mark icon is displayed.

EXCLAMATION
An exclamation mark icon is displayed.

ASTERISK
An asterisk icon is displayed.

INFORMATION
An information icon is displayed.

QUERY
A query icon is displayed.

WARNING
A warning icon is displayed.

ERROR
An error icon is displayed.

action The selected message box button. Possible values are:

1 The OK button was pressed

2 The CANCEL button was pressed

3 The ABORT button was pressed

4 The RETRY button was pressed

5 The IGNORE button was pressed

6 The YES button was pressed

7 The NO button was pressed

8 The ENTER button was pressed

Example

```
                                      /* Give option to quit        */
 if RxMessageBox("Shall we continue",, "YesNo", "Query") = 7
   Then Exit                          /* quit option given, exit    */
```

SysCls

SysCls()

Clears the screen quickly.

Example

```
call SysCls
```

SysCreateObject

result = SysCreateObject(classname, title, location <,icon>)

Creates a new instance of an object class.

Parameters

classname The registered object class name.

title The object title.

location The object location. This can be either a descriptive path (for example, OS/2 System Folder\System Configuration) or a file system path (for example, C:\bin\mytools).

icon The name of an icon .ICO file with the object icon.

result The WinCreateObject return code. This will return 1 (TRUE) if the object was created and 0 (FALSE) if the object was not created.

Example

```
 /* Code    */
  if \SysCreateObject("NewObject","NEWDLL","C:\tools\bin") then
    say 'Install successfully completed for NewObject'
```

SysCurPos

pos = SysCurPos(row, col)

Moves the cursor to the specified row and column and queries the cursor position.

Parameters

row The new cursor row postion.

col The cursor column postion.

pos The current cursor position. SysCurPos returns the position in the form 'row col'.

Note: Position (0,0) is the upper left corner. You may call SysCurPos without a column and row position to obtain the cursor position without moving the cursor.

Example

```
/* Code */
call SysCls
parse value SysCurPos() with row col
say 'Cursor position is 'row', 'col

/* Output */
Cursor position is 0, 0
```

SysCurState

SysCurState state

Hides or displays the cursor.

Parameters

state The new cursor state. Allowed states are:

- **ON** display the cursor
- **OFF** hide the cursor

SysDeregisterObjectClass

result = SysDeregisterObjectClass(classname)

Deregisters an object class definition from the system.

Parameters

classname The object class name.

result The WinDeregisterObjectClass return code. SysDeregisterObjectClass will return 1 (TRUE) if the class was deregistered and 0 (FALSE) if the class was not deregistered.

Example

```
/* Code    */
 call SysDeregisterObjectClass "OldObjectClass"
```

SysDriveInfo

info = SysDriveInfo (drive)

Returns drive information.

Parameters

info Drive information returned in the following form: *'drive: free total label'*.

Where:

drive: is the drive letter identifier.
free is the drive unused space.
total is the total size of the drive.
label is the drive label.

If the drive is not accessible, then *info* will equal ''.

drive The drive of interest, 'C:'.

Example

```
/* Code */
say 'Disk='SysDriveInfo('C:')

/* Output */
Disk=C: 33392640 83687424 TRIGGER_C
```

SysDriveMap

map = SysDriveMap ([drive], [opt])

Returns accessible drives in the form: 'C: D: . .'

Parameters

drive The first drive letter of the drive map. The default is 'C:'.

opt The drivemap option. The accepted options are:

USED
Returns the drives that are accessible or in use, including all local and remote drives. This is the default.

FREE
Returns drives that are free or not in use.

LOCAL
Returns only local drives.

REMOTE
Returns only remote drives, such as redirected LAN resources or installable file system (IFS) attached drives.

DETACHED
Returns detached LAN resources.

map A string of blank separated drive letters.

Example

```
/* Code */
say 'Used drives include:'
say SysDriveMap('C:', 'USED')

/* Output */
Used drives include:
C: D: E: F: W:
```

SysDropFuncs

call SysDropFuncs

Drops all RexxUtil functions. Once a REXX program call SysDropFuncs, the RexxUtil functions will not be availablee in any OS/2 operating system sessions.

SysFileDelete

rc = SysFileDelete(file)

Deletes a file. SysFileDelete does not support wildcard file specifications.

Parameters

file The name of the deleted file.

rc The return code from SysFileDelete. The following return codes are commonly returned:

0	File deleted successfully.
2	File not found.
3	Path not found.
5	Access denied.
26	Not DOS disk.
32	Sharing violation.
36	Sharing buffer exceeded.
87	Invalid parameter
206	Filename exceeds range error

Unlike the OS/2 operating system ERASE or DELETE commands, SysFileDelete does not issue an error message if the file doesn't exist.

Example

```
parse arg InputFile OutputFile
call SysFileDelete OutputFile   /* unconditionally erase output file */
```

SysFileTree

rc = SysFileTree(filespec, stem, [options], [tattrib], [nattrib])

Finds all files that match a file specification. SysFileTree returns the file descriptions (date, time, size, attributes, and file specification) in a REXX stem variable collection.

Parameters

filespec The search file specification.

stem The result stem variable name. SysFileTree sets REXX variable *stem.0* to the number of files and directories found and variables *stem.1* to *stem.n* to the individual file descriptions.

options Any combination of the following:

- **F** Search only for files.
- **D** Search only for directories.
- **B** Search for both files and directories. This is the default.
- **S** Search file subdirectories also.
- **T** Return the time and date in the form: YY/MM/DD/HH/MM.
- **O** Returns only the file specifications. Be default, SysFileTree returns the date, time, size, attributes, and file specification for each file.

tattrib The target attribute mask for file specification matches. Only files that match the target mask will be returned. The default mask is '*****' which returns all files regardless of the Archive, Directory, Hidden, Read-Only, and System attribute settings. The target mask attributes must appear in the order 'ADHRS'.

Target Mask Options

- *. The file attribute may be any state.
- + The file attribute must be set.
- - The file attribute must be cleared.

Target Mask Examples

'***+*'	Find all files with the Read-Only attribute set.
'+**+*'	Find all files with the Read-Only and Archive attributes set.
'*++**'	Find all hidden subdirectories.
'---+-'	Find all files with only the Read-Only attribute set.

nattrib The new attribute mask used to set the attributes of each matching file. The default mask, '*****', leaves the Archive, Directory, Hidden, Read-Only, and System attributes unchanged. The target mask attributes must appear in the order 'ADHRS'.

New Atrribute Mask Options

- * The file attribute will not be changed.
- + The file attribute will be set.
- - The file attribute will be cleared.

New Attribute Mask Examples

'***+*'	Sets the Read-Only attribute on all files.

'-**+*' Sets the Read-Only attribute and clear the Archive attribute of each file.

'+*+++' Sets all file attributes, excluding the directory attribute.

'-----' Clears all attributes on all files.

Note: You cannot set the directory bit on non-directory files. SysFileTree returns the file attribute settings after the new attribute mask have been applied.

rc The SysFileTree return code. The following return codes are of particular interest:

0 Successful.

2 Not enough memory.

Examples

```
/* Find all subdirectories on C: */
 call SysFileTree 'c:\*.*', 'file', 'SD'

/* Find all Read-Only files */
 call SysFileTree 'c:\*.*', 'file', 'S', '***+*'

/* Clear Archive and Read-Only bits of files which have them set */
 call SysFileTree 'c:\*.*', 'file', 'S', '+**+*', '-**-*'


/****<< Sample Code and Output Example.>>********/

/* Code */
 call SysFileTree 'c:\os2*.', 'file', 'B'
 do i=1 to file.0
   say file.i
 end

/* Actual Output */
12:15:89  12:00a       4096  A-HRS  C:\OS2LDR
12:15:89  12:00a      29477  A-HRS  C:\OS2KRNL
 5:24:89   4:59p          0  -D---  C:\OS2
```

SysFileSearch

call SysFileSearch target, file, stem, [options]

Finds all file lines containing the target string. SysFileSearch returns the file lines in a REXX stem variable collection.

Parameters

target The target search string.

file The searched file.

stem The result stem variable name. SysFileSearch sets REXX variable *stem.0* to the number of lines returned and variables *stem.1* to *stem.n* to the individual file lines.

options Any combination of the following one-character options:

- **C** Conducts a case-sensitive search.
- **N** Also return file line numbers.

The default is a case insensitive search without line numbers.

Return Codes

0 Successful.

2 Not enough memory.

3 Error opening file.

Examples

```
/* Find DEVICE statements in CONFIG.SYS */
call SysFileSearch 'DEVICE', 'C:\CONFIG.SYS', 'file.'
do i=1 to file.0
 say file.i
end

/* Output */
DEVICE=C:\OS2\DOS.SYS
DEVICE=C:\OS2\PMDD.SYS
DEVICE=C:\OS2\COM02.SYS
SET VIDEO_DEVICES=VIO_IBM8514A
SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
DEVICE=C:\OS2\POINTDD.SYS
DEVICE=C:\OS2\MSPS202.SYS
DEVICE=C:\OS2\MOUSE.SYS TYPE=MSPS2$


/* Find DEVICE statements in CONFIG.SYS (along with */
/* line numbers)                                    */
call SysFileSearch 'DEVICE', 'C:\CONFIG.SYS', 'file.', 'N'
do i=1 to file.0
 say file.i
end

/* Output */
20 DEVICE=C:\OS2\DOS.SYS
21 DEVICE=C:\OS2\PMDD.SYS
22 DEVICE=C:\OS2\COM02.SYS
33 SET VIDEO_DEVICES=VIO_IBM8514A
34 SET VIO_IBM8514A=DEVICE(BVHVGA,BVH8514A)
40 DEVICE=C:\OS2\POINTDD.SYS
41 DEVICE=C:\OS2\MSPS202.SYS
42 DEVICE=C:\OS2\MOUSE.SYS TYPE=MSPS2$
```

SysGetEA

result = SysGetEA(file, name, variable)

Reads a file extended attribute.

Parameters

file The file name.

name The extended attribute name.

variable The REXX variable where SysGetEA places the extended attribute value.

result The function result. When *result* is 0, the SysGetEA retrieved the extended attribute and placed the extended attribute value in *variable*. When *result* is non-zero, *result* is the OS/2 operating system return code.

Example

```
/* Code    */
if (SysGetEA("C:\CONFIG.SYS", ".type", "TYPEINFO") = 0 then
  parse var typeinfo 11 type
  say type

/* Output */
OS/2 Command File
```

SysGetKey

key = SysGetKey([opt])

Reads the next key from the keyboard buffer. If the keyboard buffer is empty, SysGetKey will wait until a key is pressed. Unlike the REXX CHARIN built-in function, SysGetKey() does not wait until the Enter key is pressed.

Parameters

key The pressed key.

opt An option controlling screen echoing. Allowed values are:

ECHO Echo the pressed key to the screen. ECHO is the default.
NOECHO Do not echo the pressed key.

SysGetMessage

msg = SysGetMessage(num, [file] [str1],...[str9])

Retrieves a message from an OS/2 operating system message file. SysGetMessage can insert up to 9 message text variables.

Message files are created with the OS/2 2.0 Toolkit MKMSGF utility. MKMSGF is documented in the toolkit *Online Tools Reference*.

Parameters

num The message number.

file The message file containing the message. The default message file is OSO001.MSG. SysGetMessage searches for message files in the system root directory (C:\), the current directory, and along the current DPATH.

str1,...str9 Insertion text variables. If the message contains insertion fields %1 to %9, SysGetMessage will insert the optional parameters *str1* through *str9* into the message.

msg The retrieved message, with variable substitutions performed.

Example

```
/*** Sample code segment using SysGetMessage and insertion text vars ***/
msg =  SysGetMessage(34, , 'A:', 'diskette labeled "Disk 2"', 'SER0002')
say msg
```

```
/** Output **/
The wrong diskette is in the drive.
Insert diskette labeled "Disk 2" (Volume Serial Number: SER0002)
into drive A:.
```

SysIni

result = SysIni ([inifile], app, key, val, stem)

Allows limited access to profile variables. Variables are stored in profiles as **Application Names** with associated **Key Names** or keywords. The SysIni can store application profile information.

Note: SysIni can retrieve and store all types of profile data (text, numeric, binary). Care should be used when changing application profile information.

Parameters

inifile The profile file name containing the profile variables. *inifile* can be a file specification, or one of the following:

- **USER** The user profile file (usually C:\OS2\OS2.INI). This is the default.
- **SYSTEM** The system INI file (usually C:\OS2\OS2SYS.INI).
- **BOTH** Search both the user and system profile files. When setting variables, SysIni will write to the user profile file.

app The application name used to store profile information.

key The keyword name holding profile information.

val The application keyword value.

stem The name of a REXX stem variable collection to receive profile information. SysIni will set REXX variable *stem.0* to the number of profile variables returned. Variables *stem.1* to *stem.n* are set to the individual profile variable values.

result When SysIni successfully sets a profile variable, *result* will equal ''. When SysIni successfully retrieves a profile variable, *result* result will be the application keyword value. When SysIni successfully deletes a profile variable, *result* will equal ''.

SysIni will return the error string ERROR: when an error occurs. Possible error conditions include:

- An an application/key pair does not exist.
- An error opening the profile file occurred. You may have specified the current user or system INI file with a relative file specification. The full file specificaiton must be used.

SysIni has six modes. The modes and the syntax variations are as follows:

Set a single key value.
SysIni([inifile], app, key, val)

Retrieve a single key value.
SysIni([inifile], app, key)

Delete a single key.
SysIni([inifile], app, key, 'DELETE:')

Delete an application and all associated keys.
SysIni([inifile], app, ['DELETE:'])

Retrieve all keys for an application.
SysIni([inifile], app, 'ALL:', 'stem')

Retrieve the names of all applications.
SysIni([inifile], 'ALL:', 'stem')

Example

```
/* Sample code segments */

/***  Save the user entered name under the key 'NAME' of *****
****  the application 'MYAPP'.                              ****/
pull name .
call SysIni , 'MYAPP', 'NAME', name /* Save the value        */
say  SysIni(, 'MYAPP', 'NAME')      /* Query the value       */
call SysIni , 'MYAPP'               /* Delete all MYAPP info */
exit

/****  Type all OS2.INI file information to the screen  *****/
call SysIni , 'ALL', 'Apps'                /* Stem Var=Apps */
if Result = 'ERROR:' then
 do i=1 to Apps.0
   call SysIni , Apps.i, 'ALL', 'Keys'     /* Stem Var=Keys */
   if Result = 'ERROR:' then
    do j=1 to Keys.0
      val = SysIni( , Apps.i, Keys.j)
      say left(Apps.i, 20) left(Keys.j, 20),
            'Len=x'left(d2x(length(val)),4) left(val, 20)
    end
 end
exit
```

SysMkDir

rc = SysMkDir(dirspec)

Creates a file directory.

Parameters

dirspec The directory specification.

rc The SysMkDir return code. The following return codes are of particular interest:

- **0** Directory creation was successful.
- **2** File not found.
- **3** Path not found.
- **5** Access denied.
- **26** Not a DOS disk.
- **87** Invalid parameter.
- **108** Drive locked.
- **206** Filename exceeds range.

Unlike the OS/2 operating system MD (Make Directory) command, the SysMkDir function does not issue an error message when the directory cannot be created.

Example

```
call SysMkDir 'c:\rexx'
```

SysOS2Ver

```
ver = SysOS2Ver()
```

Returns the OS/2 operating system version information.

Parameters

ver A string containing the OS/2 operating system version information in the form 'x.xx'

SysPutEA

```
result = SysPutEA(file, name, value)
```

Writes a named extended attribute to a file.

Parameters

file The file where SysPutEA will write the extended attribute.

name The extended attribute name.

value The new extended attribute value.

result The function result. When *result* is 0, SysPutEA has written the extended attribute to the file. When *result* is non-zero, *result* contains an OS/2 operating system error code.

Example

```
                                      /* update a security classification */
call SysPutEA "C:\CONFIG.SYS", "SECURITY", "Unclassified"
```

Many file extended attributes contain binary data. Care should be used when changing extended attributes with SysPutEA.

SysQueryClassList

```
call SysQueryClassList stem
```

Retrieves the complete list of registered object classes.

Parameters

stem A REXX stem variable name. SysQueryClassList will set REXX variable *stem.0* to the number of object classes and variables *stem.1* to *stem.n* to the individual class names.

Example

```
                    /* type the list of object classes */
call SysQueryClassList "list."
do i = 1 to list.0
  say 'Class' i 'is' list.i
end
```

SysRegisterObjectClass

result = SysRegisterObjectClass(classname, modulename)

Registers a new object class definition.

Parameters

classname The new object class name.

modulename The module file containing the object definition.

result The WinRegisterObjectClass return code. SysRegisterObjectClass will return 1 (TRUE) if the class was successfully registered and 0 (FALSE) if the new class was not registered.

Example

```
/* Code    */
 if SysRegisterObjectClass("NewObject","NEWDLL") then
   say 'Install successfully completed for NewObject'
```

SysRmDir

rc = SysRmDir(dirspec)

Deletes a file directory.

Parameters

dirspec The directory specification.

rc The SysRmDir return code. The following return codes are of particular interest:

- **0** Directory removal was successful.
- **2** File not found.
- **3** Path not found.
- **5** Access denied.
- **16** Current directory.
- **26** Not a DOS disk.
- **87** Invalid parameter.
- **108** Drive locked.
- **206** Filename exceeds range.

Example

```
call SysRmDir 'c:\rexx'
```

SysSearchPath

filespec = SysSearchPath(path, filename)

Searches a file path for a file. When a file is found, SysSearchPath returns the full file specification. When the file is not found, SysSearchPath returns ''.

Parameters

path An environment variable name. The environment variable must contain a list of file directories separated by semicolons.

filename The requested file.

filespec The returned file specification. When the file is found, *filespec* will be the fully qualified file name. When the file is not found, *filespec* will be ''.

Example

```
/* Code */
fspec = SysSearchPath('PATH', 'CMD.EXE')
say "CMD.EXE is located at" fspec

/* Output */
CMD.EXE is located at C:\OS2\CMD.EXE
```

SysSetIcon

result = SysSetIcon(filename, iconfilename)

Associates an icon with a file.

Parameters

filename The file to receive the icon.

iconfilename An .ICO file containing icon data.

result The WinSetIcon return code. SysSetIcon returns 1 (TRUE) when the icon is successfully set and 0 (FALSE) when the new icon could not be set.

Example

```
/* Code    */
 if SysSetIcon(file, "NEW.ICO") then
   say 'Install successfully completed for' file
```

SysSleep

call SysSleep secs

Pauses a REXX program for a specified time interval.

Parameters

secs The number of seconds to pause.

SysTempFileName

file = SysTempFileName(template, [filter])

Returns a unique file or directory name. SysTempFileName is useful when a program requires a unique temporary file.

Parameters

template *template* is the location and base form of the temporary file or directory name. The *template* name is a file or directory specification with up to 5 filter characters.

filter The filter character used in *template*. SysTempFileName replaces each filter character in *template* with a numeric value. The resulting file or directory name will not exist on the specified drive. The default filter character is ?.

file A file or directory that does not currently exist. If an error occurred or SysTempFileName could not create a unique name from the template, a null string is returned.

Example

```
/* Code */
say SysTempFileName('C:\TEMP\MYEXEC.???')
say SysTempFileName('C:\TEMP\??MYEXEC.???')
say SysTempFileName('C:\MYEXEC@.@@@', '@')

/* Output */
C:\TEMP\MYEXEC.251
C:\TEMP\10MYEXEC.392
C:\MYEXEC6.019
```

SysTempFileName generates the filter character replacements with a random number algorithm. If the resulting file or directory already exists, SysTempFileName increments the replacement value until all possibilities have been exhausted.

SysTextScreenRead

string = SysReadScreen(row, col, [len])

Reads characters from a specified screen location.

Parameters

row The screen row.

col The screen column.

len The number of characters to read. The default is the end of the screen.

string The characters read from the screen. This includes any carriage return and linefeed characters on the screen.

Note: Limitations: This function only reads screen characters. When restoring a character string to the screen with SAY or the CHAROUT built-in function, the previous color settings will be lost.

Examples

```
/* Example which reads in the entire screen */
 screen = SysTextScreenRead( 0, 0 )

/* Example which reads in one line */
 line = SysTextScreenRead( 2, 0, 80 )
```

SysTextScreenSize

result = SysTextScreenSize()

Returns the screen size.

Parameters

result The screen size. The result string format is 'row col'.

Example

```
/* Example */
parse value SysScreenSize() with row col
say 'Rows='row', Columns='col
```

SysWaitNamedPipe

result = SysWaitNamedPipe(name, [timeout])

Performs a timed wait on a named pipe.

Parameters

name The named pipe name. Pipe names must have the form "\PIPE\pipename."

timeout The number of microseconds to wait. If *timeout* is omitted or zero, SysWaitNamedPipe uses the pipe default timeout value. A *timeout* of -1 will wait until the named pipe is no longer busy.

result The DosWaitNmPipe return code. The following return codes are of particular interest:

- **0** The named pipe is no longer busy.
- **2** The named pipe was not found.
- **231** The wait timed out before the pipe became available.

Example

```
Parse value stream(PipeName,'C','OPEN') with PipeState ':' OS2RC
If OS2RC=231 then call SysWaitNamedPipe(PipeName, -1)
```

Chapter 5. Parsing

The parsing instructions are ARG, PARSE, and PULL (see "ARG" on page 3-4, "PARSE" on page 3-25, and "PULL" on page 3-29).

The data to parse is a *source string*. Parsing splits up the data in a source string and assigns pieces of it into the variables named in a template. A *template* is a model telling how to split the source string. The simplest kind of template consists of only a list of variable names. Here is an example:

```
variable1 variable2 variable3
```

This kind of template parses the source string into blank-delimited words. More complicated templates contain patterns in addition to variable names.

String patterns Match characters in the source string to tell where to split it. (See "Templates Containing String Patterns" on page 5-3 for details.)

Positional patterns Indicate the character positions at which to split the source string. (See "Templates Containing Positional (Numeric) Patterns" on page 5-4 for details.)

Parsing is essentially a two-step process.

1. Parse the source string into appropriate substrings using patterns.
2. Parse each substring into words.

Simple Templates for Parsing into Words

Here is a parsing instruction:

```
parse value 'time and tide' with var1 var2 var3
```

The template in this instruction is: `var1 var2 var3`. The data to parse is between the keywords `PARSE VALUE` and the keyword `WITH`, the source string `'time and tide'`. Parsing divides the source string into blank-delimited words and assigns them to the variables named in the template as follows:

```
var1='time'
var2='and'
var3='tide'
```

In this example, the source string to parse is a literal string, `time and tide`. In the next example, the source string is a variable.

```
/* PARSE VALUE using a variable as the source string to parse    */
string='time and tide'
parse value string with var1 var2 var3          /* same results */
```

(PARSE VALUE does not convert alphabetic characters in the source string to uppercase (lowercase `a`–`z` to uppercase `A`–`Z`). If you want to convert characters to uppercase, use PARSE UPPER VALUE. A summary of the effect of parsing instructions on case is in "Using UPPER" on page 5-8.)

All of the parsing instructions assign the parts of a source string into the variables named in a template. There are various parsing instructions because of differences in the nature or origin of source strings. (A summary of all the parsing instructions is on page 5-8.)

The PARSE VAR instruction is similar to PARSE VALUE except that the source string to parse is always a variable. In PARSE VAR, the name of the variable containing the source string follows the keywords `PARSE VAR`. In the next example, the variable `stars` contains the source string. The template is `star1 star2 star3`.

```
/* PARSE VAR example                                          */
stars='Sirius Polaris Rigil'
parse var stars star1 star2 star3           /* star1='Sirius'  */
                                            /* star2='Polaris' */
                                            /* star3='Rigil'   */
```

All variables in a template receive new values. If there are *more variables in the template than words in the source string*, the leftover variables receive null (empty) values. This is true for all parsing: for parsing into words with simple templates and for parsing with templates containing patterns. Here is an example using parsing into words.

```
/* More variables in template than (words in) the source string */
satellite='moon'
parse var satellite Earth Mercury             /* Earth='moon'  */
                                              /* Mercury=''    */
```

If there are *more words in the source string than variables in the template*, the last variable in the template receives all leftover data. Here is an example:

```
/* More (words in the) source string than variables in template */
satellites='moon Io Europa Callisto...'
parse var satellites Earth Jupiter            /* Earth='moon'  */
                                /* Jupiter='Io Europa Callisto...'*/
```

Parsing into words removes leading and trailing blanks from each word before it is assigned to a variable. The exception to this is the word or group of words assigned to the last variable. The last variable in a template receives leftover data, *preserving extra leading and trailing blanks*. Here is an example:

```
/* Preserving extra blanks                                      */
solar5='Mercury Venus  Earth   Mars     Jupiter  '
parse var solar5 var1 var2 var3 var4
/* var1  ='Mercury'                                             */
/* var2  ='Venus'                                               */
/* var3  ='Earth'                                               */
/* var4  ='  Mars     Jupiter  '                                */
```

In the source string, `Earth` has two leading blanks. Parsing removes both of them (the word-separator blank and the extra blank) before assigning `var3='Earth'`. `Mars` has three leading blanks. Parsing removes one word-separator blank and keeps the other two leading blanks. It also keeps all five blanks between `Mars` and `Jupiter` and both trailing blanks after `Jupiter`.

Parsing removes *no* blanks if the template contains only one variable. For example:

```
parse value '   Pluto   ' with var1        /* var1='   Pluto   '*/
```

The Period as a Placeholder

A period in a template is a placeholder. It is used instead of a variable name, but it receives no data. It is useful:

- As a "dummy variable" in a list of variables
- Or to collect unwanted information at the end of a string.

The period in the first example is a placeholder. Be sure to separate adjacent periods with spaces; otherwise, an error results.

```
/* Period as a placeholder                                        */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars . . brightest .         /* brightest='Sirius' */

/* Alternative to period as placeholder                           */
stars='Arcturus Betelgeuse Sirius Rigil'
parse var stars drop junk brightest rest   /* brightest='Sirius' */
```

A placeholder saves the overhead of unneeded variables.

Templates Containing String Patterns

A *string pattern* matches characters in the source string to indicate where to split it. A string pattern can be a:

Literal string pattern One or more characters within quotation marks.

Variable string pattern A variable within parentheses with no sign before the left parenthesis. (See page 5-7 for details.)

Here are two templates: a simple template and a template containing a literal string pattern:

```
var1 var2          /* simple template                               */
var1 ', ' var2     /* template with literal string pattern          */
```

The literal string pattern is: `', '`. This template:

- Puts characters from the start of the source string up to (but not including) the first character of the match (the comma) into `var1`
- Puts characters starting with the character after the last character of the match (the character after the blank that follows the comma) and ending with the end of the string into `var2`.

A template with a string pattern can omit some of the data in a source string when assigning data into variables. The next two examples contrast simple templates with templates containing literal string patterns.

```
/* Simple template                                                */
name='Smith, John'
parse var name ln fn                   /* Assigns: ln='Smith,' */
                                       /*          fn='John'   */
```

Notice that the comma remains (the variable `ln` contains `'Smith,'`). In the next example the template is `ln ', ' fn`. This removes the comma.

```
/* Template with literal string pattern                           */
name='Smith, John'
parse var name ln ', ' fn              /* Assigns: ln='Smith'  */
                                       /*          fn='John'   */
```

First, the language processor scans the source string for `', '`. It splits the source string at that point. The variable `ln` receives data starting with the first character of the source string and ending with the last character before the match. The variable `fn` receives data starting with the first character *after* the match and ending with the end of string.

A template with a string pattern omits data in the source string that matches the pattern. (There is a special case (on page 5-11) in which a template with a string

pattern does *not* omit matching data in the source string.) We used the pattern ', ' (with a blank) instead of ',' (no blank) because, without the blank in the pattern, the variable fn receives ' John' (including a blank).

If the source string *does not contain a match for a string pattern*, then the variables preceding the unmatched string pattern get all the data in question. Any variables after that pattern receive the null string.

A null string is never found. It always matches the end of the source string.

Templates Containing Positional (Numeric) Patterns

A *positional pattern* is a number that identifies the character position at which to split data in the source string. The number must be a whole number.

An *absolute positional pattern* is

- A number with no + or - preceding it or with an equal sign preceding it
- A variable in parentheses with an equal sign before the left parenthesis. (See page 5-8 for details on *variable positional patterns.*)

The number specifies the absolute character position at which to split the source string.

Here is a template with absolute positional patterns:

```
variable1 11 variable2 21 variable3
```

The numbers 11 and 21 are absolute positional patterns. The number 11 refers to the 11th position in the input string, 21 to the 21st position. This template:

- Puts characters 1 through 10 of the source string into variable1
- Puts characters 11 through 20 into variable2
- Puts characters 21 to the end into variable3.

Positional patterns are probably most useful for working with a file of records, such as:

```
           character positions:
           1         11         21                  40
          ┌──────────┬──────────┬───────────────────┐end of
 FIELDS:  │LASTNAME  │FIRST     │PSEUDONYM          │record
          └──────────┴──────────┴───────────────────┘
```

The following example uses this record structure.

```
/* Parsing with absolute positional patterns in template        */
record.1='Clemens   Samuel    Mark Twain          '
record.2='Evans     Mary Ann  George Eliot        '
record.3='Munro     H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                        /* Says 'By George!' after record 2 */
```

The source string is first split at character position 11 and at position 21. The language processor assigns characters 1 to 10 into lastname, characters 11 to 20 into firstname, and characters 21 to 40 into pseudonym.

The template could have been:

```
1 lastname 11 firstname 21 pseudonym
```

instead of

```
  lastname 11 firstname 21 pseudonym
```

Specifying the 1 is optional.

Optionally, you can put an equal sign before a number in a template. An equal sign is the same as no sign before a number in a template. The number refers to a particular character position in the source string. These two templates work the same:

```
lastname  11 first  21 pseudonym

lastname =11 first =21 pseudonym
```

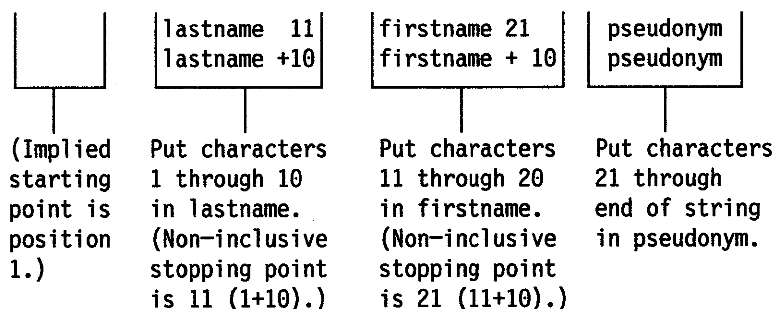
A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. (It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; see page 5-8 for details on variable positional patterns.)

The number specifies the relative character position at which to split the source string. The plus or minus indicates movement right or left, respectively, from the start of the string (for the first pattern) or the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for absolute positional patterns done with relative positional patterns:

```
/* Parsing with relative positional patterns in template        */
record.1='Clemens   Samuel    Mark Twain          '
record.2='Evans     Mary Ann  George Eliot        '
record.3='Munro     H.H.      Saki                '
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end                                               /* same results  */
```

Blanks between the sign and the number are insignificant. Therefore, +10 and + 10 have the same meaning. Note that +0 is a valid relative positional pattern.

Absolute and relative positional patterns are interchangeable (except in the special case (on page 5-11) when a string pattern precedes a variable name and a positional pattern follows the variable name). The templates from the examples of absolute and relative positional patterns give the same results.



Only with positional patterns can a matching operation *back up* to an earlier position in the source string. Here is an example using absolute positional patterns:

```
/* Backing up to an earlier position (with absolute positional)  */
string='astronomers'
parse var string 2 var1 4 1 var2 2 4 var3 5 11 var4
say string 'study' var1||var2||var3||var4
/* Displays: "astronomers study stars"                           */
```

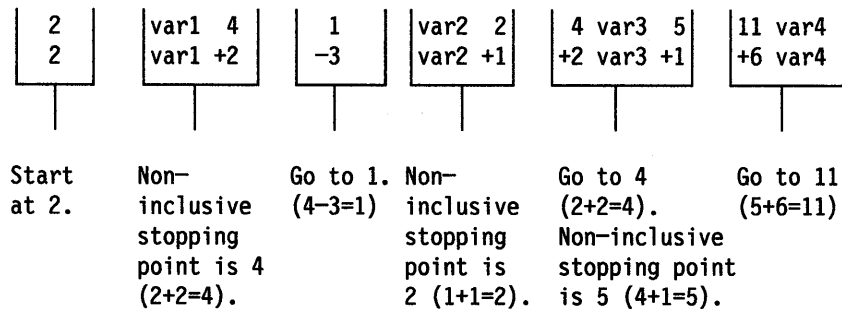
The absolute positional pattern 1 backs up to the first character in the source string.

With relative positional patterns, a number preceded by a minus sign backs up to an earlier position. Here is the same example using relative positional patterns:

```
/* Backing up to an earlier position (with relative positional)  */
string='astronomers'
parse var string 2 var1 +2 -3 var2 +1 +2 var3 +1 +6 var4
say string 'study' var1||var2||var3||var4      /* same results   */
```

In the previous example, the relative positional pattern -3 backs up to the first character in the source string.

The templates in the last two examples are equivalent.



You can use templates with positional patterns to make multiple assignments:

```
/* Making multiple assignments                                   */
books='Silas Marner, Felix Holt, Daniel Deronda, Middlemarch'
parse var books 1 Eliot 1 Evans
/* Assigns the (entire) value of books to Eliot and to Evans.    */
```

Combining Patterns and Parsing Into Words

What happens when a template contains patterns that divide the source string into sections containing multiple words? String and positional patterns divide the source string into substrings. The language processor then applies a section of the template to each substring, following the rules for parsing into words.

```
/* Combining string pattern and parsing into words               */
name='    John      Q.    Public'
parse var name fn init '.' ln        /* Assigns: fn='John'       */
                                     /*          init='     Q'   */
                                     /*          ln='    Public'  */
```

The pattern divides the template into two sections:

- `fn init`
- `ln`

The matching pattern splits the source string into two substrings:

- `'    John      Q'`
- `'    Public'`

The language processor parses these substrings into words based on the appropriate template section.

John had three leading blanks. All are removed because parsing into words removes leading and trailing blanks except from the last variable.

Q has six leading blanks. Parsing removes one word-separator blank and keeps the rest because init is the last variable in that section of the template.

For the substring '   Public', parsing assigns the entire string into ln without removing any blanks. This is because ln is the only variable in this section of the template. (For details about treatment of blanks, see page 5-2.)

```
/* Combining positional patterns with parsing into words          */
string='R E X X'
parse var string var1 var2 4 var3 6 var4   /* Assigns: var1='R'  */
                                           /*          var2='E'  */
                                           /*          var3=' X' */
                                           /*          var4=' X' */
```

The pattern divides the template into three sections:

- var1 var2
- var3
- var4

The matching patterns split the source string into three substrings that are individually parsed into words:

- 'R E'
- ' X'
- ' X'

The variable var1 receives 'R'; var2 receives 'E'. Both var3 and var4 receive ' X' (with a blank before the X) because each is the only variable in its section of the template. (For details on treatment of blanks, see page 5-2.)

Parsing with Variable Patterns

You may want to specify a pattern by using the value of a variable instead of a fixed string or number. You do this by placing the name of the variable in parentheses. This is a *variable reference*. Blanks are not necessary inside or outside the parentheses, but you can add them if you wish.

The template in the next parsing instruction contains the following literal string pattern '. '.

```
parse var name fn  init '. ' ln
```

Here is how to specify that pattern as a variable string pattern:

```
strngptrn='. '
parse var name fn init (strngptrn) ln
```

If no equal, plus, or minus sign precedes the parenthesis that is before the variable name, the value of the variable is then treated as a string pattern. The variable can be one that has been set earlier in the same template.

Example:

```
/* Using a variable as a string pattern                        */
/*  The variable (delim) is set in the same template           */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/90 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
            /* Sets: month='11'; delim='/'; day='15'; year='90'  */
```

If an equal, a plus, or a minus sign precedes the left parenthesis, then the value of the variable is treated as an absolute or relative positional pattern. The value of the variable must be a nonnegative whole number.

The variable can be one that has been set earlier in the same template. In the following example, the first two fields specify the starting character positions of the last two fields.

Example:

```
/* Using a variable as a positional pattern                    */
dataline = '6 20 Samuel ClemensMark Twain'
parse var dataline pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname='Samuel Clemens'; pseudonym='Mark Twain'    */
```

Using UPPER

Specifying UPPER on any of the PARSE instructions converts characters to uppercase (lowercase a–z to uppercase A–Z) before parsing. The following table summarizes the effect of the parsing instructions on case.

Converts alphabetic characters to uppercase before parsing	Maintains alphabetic characters in case entered
ARG PARSE UPPER ARG	PARSE ARG
PARSE UPPER LINEIN	PARSE LINEIN
PULL PARSE UPPER PULL	PARSE PULL
PARSE UPPER SOURCE	PARSE SOURCE
PARSE UPPER VALUE	PARSE VALUE
PARSE UPPER VAR	PARSE VAR
PARSE UPPER VERSION	PARSE VERSION

The ARG instruction is simply a short form of PARSE UPPER ARG. The PULL instruction is simply a short form of PARSE UPPER PULL. If you do not desire uppercase translation, use PARSE ARG (instead of ARG or PARSE UPPER ARG) and use PARSE PULL (instead of PULL or PARSE UPPER PULL).

Parsing Instructions Summary

Remember: *All* parsing instructions assign parts of the source string into the variables named in the template. The following table summarizes where the source string comes from.

Instruction	Where the source string comes from
ARG PARSE ARG	Arguments you list when you invoke the program or arguments in the call to a subroutine or function.
PARSE LINEIN	Next line in the default input stream.
PULL PARSE PULL	The string at the head of the external data queue. (If queue empty, uses default input, typically the terminal.)
PARSE SOURCE	System-supplied string giving information about the executing program.
PARSE VALUE	Expression between the keyword VALUE and the keyword WITH in the instruction.
PARSE VAR *name*	Parses *name*.
PARSE VERSION	System-supplied string telling the language, language level, and (three-word) date.

Parsing Instructions Examples

All examples in this section parse source strings into words.

ARG

```
/* ARG with source string named in program invocation          */
/*  Program name is PALETTE.  Specify 2 primary colors (yellow, */
/*   red, blue) on call.   Assume call is: palette red blue     */
arg var1 var2             /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1                      /* entered duplicates */
END
Say new; exit                             /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit
```

ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in "Parsing Multiple Strings" on page 5-10.

PARSE ARG works the same as ARG except that PARSE ARG does not convert alphabetic characters to uppercase before parsing.

PARSE LINEIN

```
parse linein 'a=' num1 'b=' num2      /* Assume: 8 and 9        */
sum=num1+num2                         /* Enter: a=8 b=9 as input */
say sum                               /*  Displays: "17"         */
```

PARSE PULL

```
PUSH '80 7'                 /* Puts data on queue                        */
parse pull fourscore seven  /* Assigns: fourscore='80'; seven='7'        */
SAY fourscore+seven         /* Displays: "87"                            */
```

PARSE SOURCE

```
parse source sysname .
Say sysname                          /* Displays:      "OS/2"    */
```

PARSE VALUE example is on page 5-1.

PARSE VAR examples are throughout the chapter, starting on page 5-2.

PARSE VERSION

```
parse version . level .
say level                                   /* Displays: "4.00" */
```

PULL works the same as PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.

Advanced Topics in Parsing

This section includes parsing multiple strings and flow charts depicting a conceptual view of parsing.

Parsing Multiple Strings

Only ARG and PARSE ARG can have more than one source string. To parse *multiple strings*, you can specify multiple comma-separated templates. Here is an example:

```
parse arg template1, template2, template3
```

This instruction consists of the keywords PARSE ARG and three comma-separated templates. (For an ARG instruction, the source strings to parse come from arguments you specify when you invoke a program or CALL a subroutine or function.) Each comma is an instruction to the parser to move on to the next string.

Example:

```
/* Parsing multiple strings in a subroutine                         */
num='3'
musketeers="Porthos, Athos, Aramis, D'Artagnon"
CALL Sub num,musketeers  /* Passes num and musketeers to sub        */
SAY total; say fourth /* Displays: "4" and " D'Artagnon"            */
EXIT

Sub:
parse arg subtotal, . . . fourth
total=subtotal+1
RETURN
```

Note that when a REXX program is started as a command, only one argument string is recognized. You can pass multiple argument strings for parsing:

- When one REXX program invokes another REXX program with the CALL instruction or a function call.
- When programs written in other languages start a REXX program.

If there are more templates than source strings, each variable in a leftover template receives a null string. If there are more source strings than templates, the language processor ignores leftover source strings. If a template is empty (two commas in a row) or contains no variable names, parsing proceeds to the next template and source string.

Combining String and Positional Patterns: A Special Case

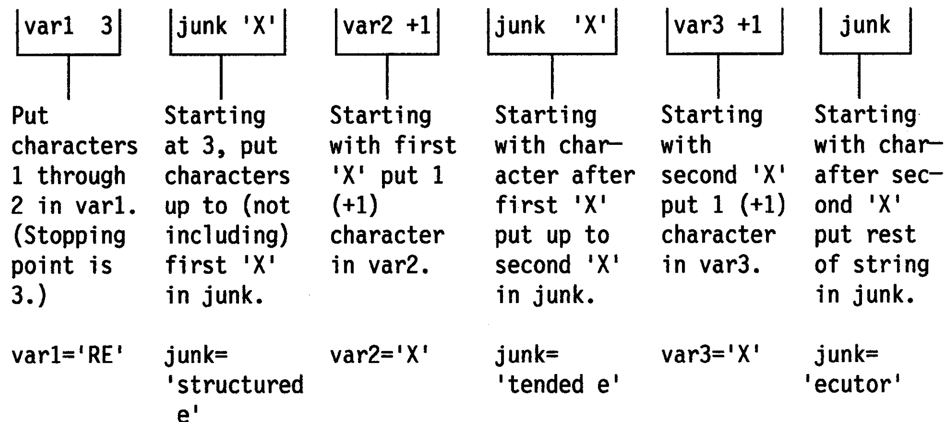
There is a special case in which absolute and relative positional patterns do not work identically. We have shown how string patterns skip over data in the source string (see page 5-3). But a template containing the sequence:

- string pattern
- variable name
- *relative* positional pattern

does *not* skip over any data. A relative positional pattern moves relative to the first character of a string pattern. As a result, assignment includes the data that is in the string pattern. Thus, the variable receives characters including the matching data.

```
/* Template containing string pattern, then variable name, then  */
/*  relative positional pattern does not skip over any data.     */
string='REstructured eXtended eXecutor'
parse var string var1 3 junk 'X' var2 +1 junk 'X' var3 +1 junk
say var1||var2||var3 /* Concatenates variables; displays: "REXX" */
```

Here is how this template works:



Parsing with DBCS Characters

Parsing with DBCS characters generally follows the same rules as parsing with SBCS characters. Literal strings can contain DBCS characters, but numbers and variable names must be in SBCS characters. See "PARSE" on page B-3 for examples of DBCS parsing.

Details of Steps in Parsing

The three figures that follow are to help you understand the concept of parsing. Please note that the figures do not include error cases.

The figures include terms whose definitions are as follows:

string start is the beginning of the source string (or substring).

string end is the end of the source string (or substring).

length is the length of the source string.

match start is in the source string and is the first character of the match.

match end is in the source string. For a string pattern, it is the first character after the end of the match. For a positional pattern, it is the same as match start.

match position is in the source string. For a string pattern, it is the first matching character. For a positional pattern, it is the position of the matching character.

token is a distinct syntactic element in a template, such as a variable, a period, a pattern, or a comma.

value is the numeric value of a positional pattern. This can be either a constant or the resolved value of a variable.

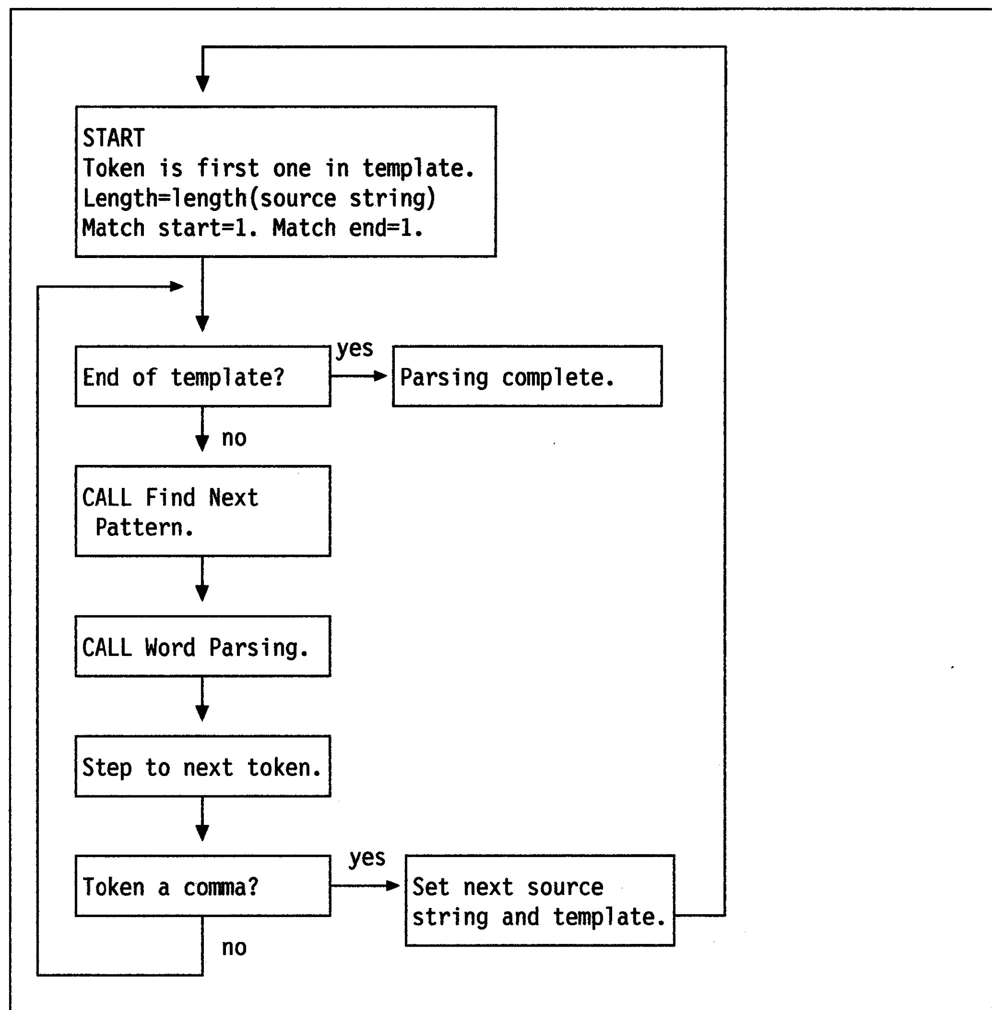


Figure 5-1. Conceptual Overview View of Parsing

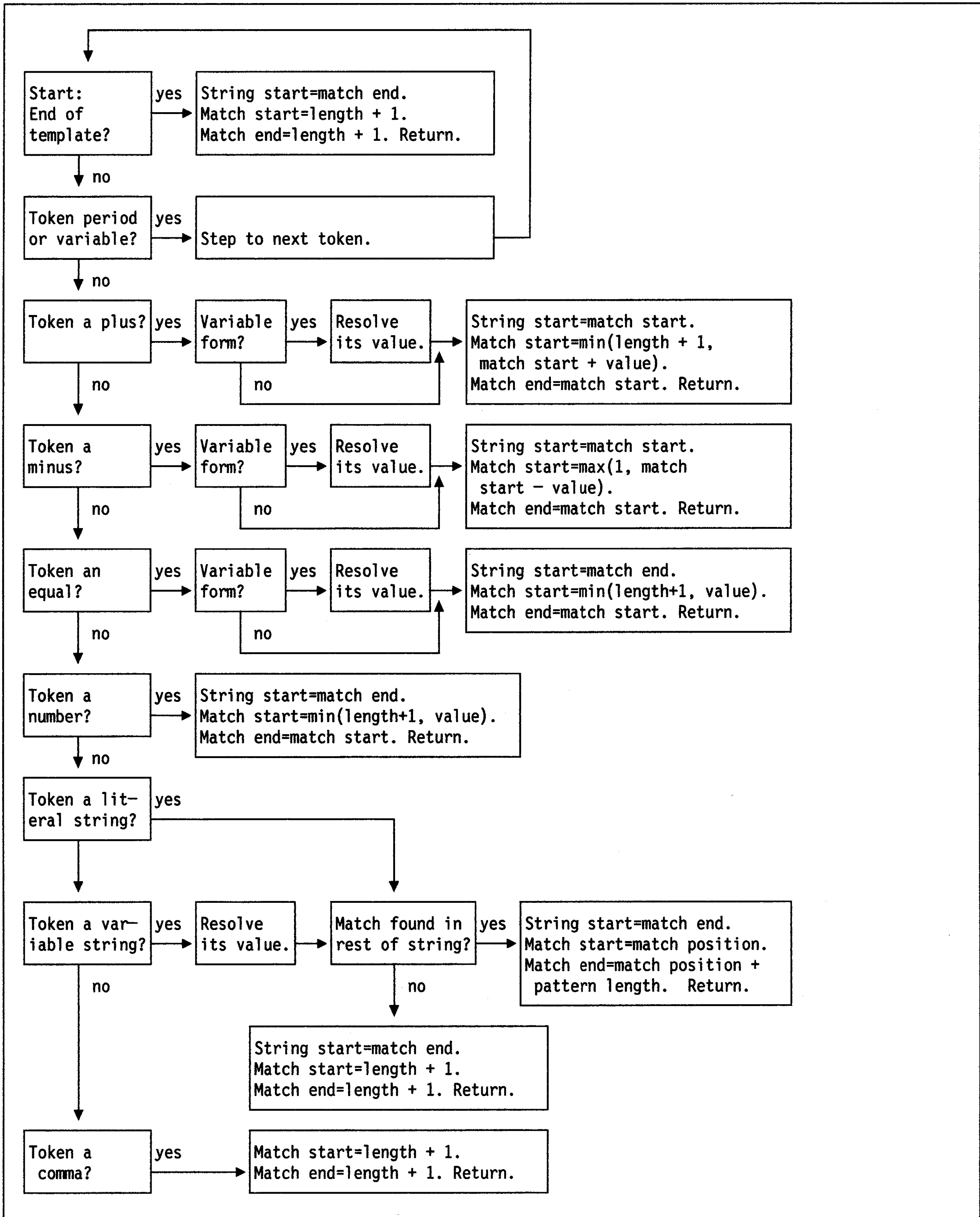


Figure 5-2. Conceptual View of Finding Next Pattern

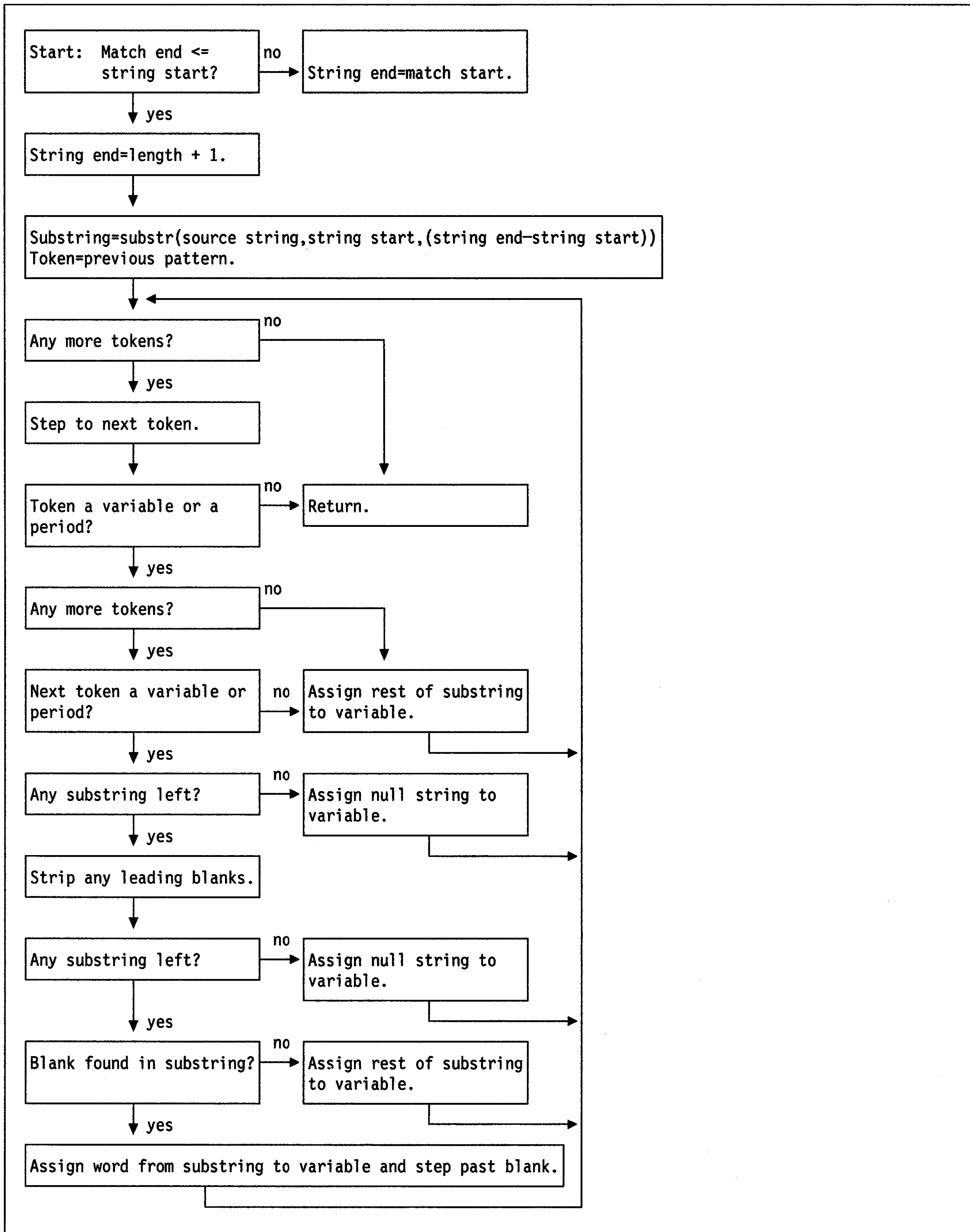


Figure 5-3. Conceptual View of Word Parsing

Chapter 6. Numbers and Arithmetic

REXX defines the usual arithmetic operations (addition, subtraction, multiplication, and division) in as natural a way as possible. What this really means is that the rules followed are those that are conventionally taught in schools and colleges.

During the design of these facilities, however, it was found that unfortunately the rules used vary considerably (indeed much more than generally appreciated) from person to person and from application to application and in ways that are not always predictable. The arithmetic described here is, therefore, a compromise that (although not the simplest) should provide acceptable results in most applications.

Introduction

Numbers (that is, character strings used as input to REXX arithmetic operations and built-in functions) can be expressed very flexibly. Leading and trailing blanks are permitted, and exponential notation can be used. Some valid numbers are:

```
   12        /* a whole number                   */
 '-76'       /* a signed whole number            */
   12.76     /* decimal places                   */
' +  0.003 ' /* blanks around the sign and so forth */
   17.       /* same as "17"                     */
     .5      /* same as "0.5"                    */
   4E9       /* exponential notation             */
    0.73e-7  /* exponential notation             */
```

(Exponential notation means that the number includes a power of ten following an E that indicates how the decimal point is shifted. Thus 4E9 above is just a short way of writing 4000000000, and 0.73e-7 is short for 0.000000073.)

The **arithmetic operators** include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, there are two further division operators: integer divide (%) divides and returns the integer part; remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see the "Definition" section for full details):

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus if a result requires more than 9 digits, it would normally be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.666666667 (it would require an infinite number of digits for perfect accuracy).
- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros). So, for example:

  ```
  2.40 + 2    ->    4.40
  2.40 - 2    ->    0.40
  2.40 * 2    ->    4.80
  2.40 / 2    ->    1.2
  ```

  This behavior is desirable for most calculations (especially financial calculations).

If necessary, you can remove trailing zeros with the STRIP function (see page 4-36), or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number will be expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12           /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11  /* not 0.0000000000333333333 */
```

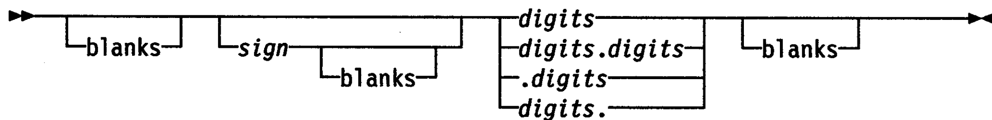
Definition

A precise definition of the arithmetic facilities of the REXX language is given here.

Numbers

A **number** in REXX is a character string that includes one or more decimal digits, with an optional decimal point. (See "Exponential Notation" on page 6-7 for an extension of this definition.) The decimal point may be embedded in the number, or may be prefixed or suffixed to it. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:



Where:

blanks
: are one or more spaces

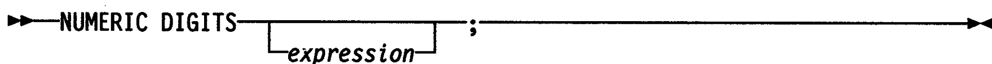
sign
: is either + or –

digits
: are one or more of the decimal digits 0 – 9.

Note that a single period alone is not a valid number.

Precision

The maximum number of significant digits that can result from an operation is controlled by the instruction:



expression is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify *expression* in this instruction, or if no NUMERIC DIGITS instruction has been executed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. Use small values, however, with care—the loss of precision and rounding thus requested affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

Arithmetic Operators

REXX arithmetic is performed by the operators +, -, *, /, %, //, and ** (add, subtract, multiply, divide, integer divide, remainder, and power), which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving just one zero if all the digits in the number are zeros). They are then truncated (if necessary) to DIGITS + 1 significant digits (the extra digit is a "guard" digit) before being used in the computation. The operation is then carried out under up to double that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Every operation is carried out in such a way that no errors will be introduced except during the final rounding of the result to the specified significance. (That is, input data is first truncated to the appropriate significant digit (NUMERIC DIGITS+1) before being used in the computation, and then divisions and multiplications are carried out to double that precision, as needed.)

Rounding is done in the traditional manner. The digit to the right of the least significant digit in the result (the "guard digit") is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is, therefore, not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, trailing zeros are removed after rounding.

The FORMAT built-in function (see page 4-23) allows a number to be represented in a particular format if the standard result provided does not meet your requirements.

Arithmetic Operation Rules—Basic Operators

The basic operators (addition, subtraction, multiplication, and division) operate on numbers as follows. All numbers have insignificant leading zeros removed before being used in computation.

Addition and Subtraction

If either number is 0, the other number, rounded to NUMERIC DIGITS digits, if necessary, is used as the result (with sign adjustment as appropriate). Otherwise, the two numbers are extended on the right and left as necessary, up to a total maximum of DIGITS + 1 digits (the number with the smaller absolute value may, therefore, lose some or all of its digits on the right) and are then added or subtracted as appropriate.

Example:

```
xxx.xxx  +  yy.yyyyy
```

becomes:

```
  xxx.xxx00
+ 0yy.yyyyy
-----------
  zzz.zzzzz
```

The result is then rounded to the current setting of NUMERIC DIGITS if necessary (taking into account any extra "carry digit" on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted), and any insignificant leading zeros are removed.

The prefix operators are evaluated using the same rules; the operations `+number` and `-number` are calculated as `0+number` and `0-number`, respectively.

Multiplication

The numbers are multiplied together ("long multiplication") resulting in a number that may be as long as the sum of the lengths of the two operands.

Example:

```
xxx.xxx * yy.yyyyy
```

becomes:

```
zzzzz.zzzzzzzz
```

The result is then rounded, counting from the first significant digit of the result, to the current setting of NUMERIC DIGITS.

Division

For the division:

```
yyy / xxxxx
```

the following steps are taken: First the number yyy is extended with zeros on the right until it is larger than the number xxxxx (with note being taken of the change in the power of ten that this implies). Thus, in this example, yyy might become yyy00. Traditional long division then takes place. This might be written:

```
         zzzz
        ------
xxxxx  | yyy00
```

The length of the result (zzzz) is such that the rightmost z is at least as far right as the rightmost digit of the (extended) y number in the example. During the division, the y number is extended further as necessary. The z number may increase up to NUMERIC DIGITS+1 digits, at which point the division stops and the result is rounded. Following completion of the division (and rounding if necessary), insignificant trailing zeros are removed.

Basic Operator Examples

Following are some examples that illustrate the main implications of the rules just described.

```
/* With:  Numeric digits 5 */
12+7.00      ->    19.00
1.3-1.07     ->     0.23
1.3-2.07     ->    -0.77
1.20*3       ->     3.60
7*3          ->    21
0.9*0.8      ->     0.72
1/3          ->     0.33333
2/3          ->     0.66667
5/2          ->     2.5
1/10         ->     0.1
12/12        ->     1
8.0/2        ->     4
```

Note: With all the basic operators, the position of the decimal point in the terms being operated upon is arbitrary. The operations may be carried out as integer operations with the exponent being calculated and applied afterwards. Therefore, the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

Arithmetic Operation Rules—Additional Operators

The power (**), integer divide (%), and remainder (//) operators rules follow.

Power

The ** **(power) operator** raises a number to a power, which may be positive, negative, or 0. The power must be a whole number. If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the result, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by one).

In practice (see Note 1 on page 6-6 for rationale), the result is calculated by the process of left-to-right binary reduction. For `x**n`: `n` is converted to binary, and a temporary accumulator is set to 1. If `n = 0` the calculation is complete. (Thus, `x**0` = 1 for all *x*, including `0**0`.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by `x`. If all bits have now been inspected, the calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the normal REXX arithmetic combination rules with the initial calculation (the multiplications) using precision of DIGITS + L + 1 digits (where L is the length in digits of the whole number `n`) and the final division using the usual NUMERIC DIGITS digits.

Integer Division

The **% (integer divide) operator** divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result if normal division were used.

The result returned will have no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed simply by digits within the precision set by the NUMERIC DIGITS instruction, the operation is in error and will fail. For example, 10000000000%3 requires 10 digits for the result (3333333333) and would, therefore, fail if `NUMERIC DIGITS 9` were in effect.

Remainder

The **// (remainder) operator** returns the remainder from integer division, which is defined as being the residue of the dividend after the operation of calculating integer division as just described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation will fail under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

Additional Operator Examples

Following are some examples using the power, integer divide, and remainder operators just described:

```
/* Again with:  Numeric digits 5 */
2**3        ->     8
2**-3       ->     0.125
1.7**8      ->    69.758
2%3         ->     0
2.1//3      ->     2.1
10%3        ->     3
10//3       ->     1
-10//3      ->    -1
10.2//1     ->     0.2
10//0.3     ->     0.1
```

Notes:

1. A particular algorithm for calculating powers is used, because it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It, therefore, gives better performance than the simpler definition of repeated multiplication. Because results may differ from those of repeated multiplication, the algorithm is defined here.
2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

Numeric Comparisons

The comparison operators are listed in "Comparison" on page 2-9. You can use any of these for comparing numeric strings. However, you should not use ==, \==, ¬==, >>, \>>, ¬>>, <<, \<<, and ¬<< to compare numeric values because leading and trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

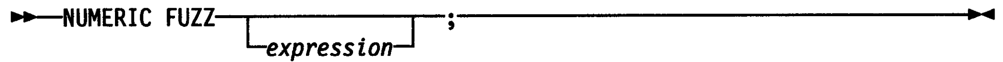
```
A ? B
```

where ? is any numeric comparison operator, is identical with:

```
(A - B) ? '0'
```

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

Comparison of two numbers is affected by a quantity called **fuzz**, which is set by the instruction:



Here *expression* must result in a whole number that is 0 or positive. This FUZZ number controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison operation. That is, the numbers are subtracted under a precision of DIGITS – FUZZ digits during the comparison. Clearly FUZZ must be less than DIGITS.

Thus if DIGITS = 9, and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though NUMERIC DIGITS 8 had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say  4.9999 = 5     /* Displays "0"   */
say  4.9999 < 5     /* Displays "1"   */
Numeric fuzz 1
say  4.9999 = 5     /* Displays "1"   */
say  4.9999 < 5     /* Displays "0"   */
```

Exponential Notation

The preceding description of numbers describes “pure” numbers, in the sense that the character strings that describe numbers could be very long. For example:

```
10000000000 * 10000000000
```

would give

```
100000000000000000000
```

and

```
.00000000001 * .00000000001
```

would give

```
0.0000000000000000000001
```

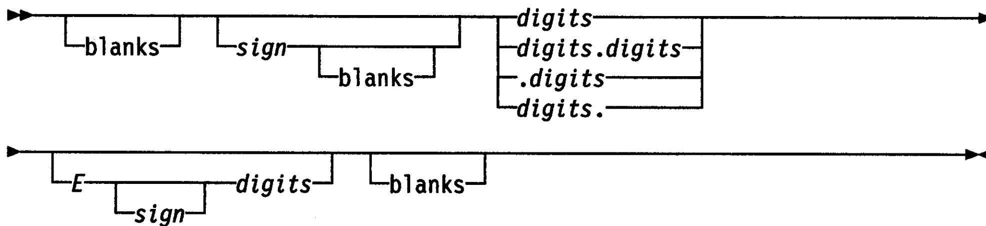
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the “simple” form would give misleading information.

For example:

```
numeric digits 5
say 54321*54321
```

would display `2950800000` if long form were used. This is clearly misleading, and so the result is expressed as `2.9508E+9` instead.

The definition of numbers is, therefore, extended as:



The integer following the `E` represents a power of ten that is to be applied to the number, and the `E` can be in uppercase or lowercase.

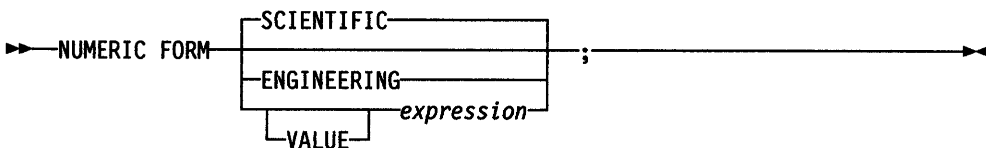
Here are some examples:

```
12E11  =  1200000000000
12E-5  =  0.00012
-12e4  =  -120000
```

The preceding numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form depending on the setting of DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form generated by REXX always has a sign following the `E` in order to improve readability. An exponential part of `E+0` will never be generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in "long" form, by using the FORMAT built-in function, described in "FORMAT" on page 4-23.

You can control whether Scientific or Engineering notation is to be used by using the instruction:



The default setting of FORM is `SCIENTIFIC`.

Scientific notation adjusts the power of 10 so there is a single nonzero digit to the left of the decimal point. Engineering notation causes powers of 10 to always be expressed as a multiple of 3: the integer part may, therefore, range from 1 through 999.

```
/* after the instruction */
Numeric form scientific

123.45 * 1e11     ->     1.2345E+13

/* after the instruction */
Numeric form engineering

123.45  * 1e11    ->    12.345E+12
```

Whole Numbers

Within the set of numbers REXX understands, it is useful to distinguish the subset defined as **whole numbers**. A whole number in REXX is a number that has a decimal part that is all zeros (or that has no decimal part). In addition, it must be possible to express its integer part simply as digits within the precision set by the NUMERIC DIGITS instruction. REXX would express larger numbers in exponential notation, after rounding, and, therefore, these could no longer be safely described or used as whole numbers.

Numbers Used Directly by REXX

As discussed, numbers are always rounded (if necessary) according to the setting of NUMERIC DIGITS during any arithmetic operation. Similarly, when REXX directly uses a number (which has not necessarily been involved in an arithmetic operation), the same rounding is also applied.

In the following cases, the number used must be a whole number with the following limits:

Power values (right hand operand of the power operator)	999999999
Values of *exprr* and *exprf* in the DO instruction	The current numeric precision (up to 999999999).
Values given for DIGITS or FUZZ in the NUMERIC instruction	999999999 (Note: FUZZ must always be less than DIGITS).
Positional patterns in parsing templates	999999999
Number given for *option* in the TRACE instruction.	999999999

Errors

Two types of errors may occur during arithmetic:

- Overflow/Underflow

  This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is 9, the OS/2 operating system supports exponents in the range -999999999 through 999999999.

  Because this allows for (very) large exponents, overflow or underflow is treated as a terminating syntax error.

- Insufficient storage

  Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail due to lack of storage. This is considered a terminating error as usual, rather than an arithmetic error.

Chapter 7. Conditions and Condition Traps

CALL and SIGNAL modify the flow of execution in a REXX program by using condition traps. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see "CALL" on page 3-6 and "SIGNAL" on page 3-35).

condition and *trapname* are symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified condition occurs, control passes to the routine or label *trapname*. SIGNAL or CALL is used, depending on whether the most recent trap for the condition was set using SIGNAL ON or CALL ON, respectively.

The conditions and their corresponding events that can be trapped are:

ERROR
: raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE is set. The condition is raised at the end of the clause that invoked the command, but is ignored if the ERROR condition trap is already in the delayed state.

FAILURE
: raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that invoked the command, but is ignored if the FAILURE condition trap is already in the delayed state.

An attempt to issue a command to an unknown subcommand environment will also raise a FAILURE condition.

HALT
: raised if an external attempt is made to interrupt and terminate execution of the program. The condition is usually raised at the end of the clause that was being executed when the external interruption occurred. When a REXX program is running in an OS/2 full-screen or command prompt session, the Ctrl+Break key sequence raises the halt condition. However, if Ctrl+Break is pressed while a command or non-REXX external function is running, the command or function ends. The REXX program is also ended by without raising the halt condition. When a REXX program is running under PMREXX, PMREXX provides a pull-down control for raising the halt condition. This affects only the REXX program, not non-REXX programs which may have been called by the REXX program.

Note: Application programs that use the REXX language processor might use the RXHALT exit or the RexxStart programming interface to halt execution of a REXX macro. See "System Exits" on page 9-26.

NOTREADY
raised if an error occurs during an input or output operation. See "Errors During Input and Output" on page 8-6. This condition is ignored if the NOTREADY condition trap is already in the delayed state.

NOVALUE
raised if an uninitialized variable is used:

- As a term in an expression
- As the *name* following the VAR subkeyword of the PARSE instruction
- As a variable reference in a parsing template, a PROCEDURE, or a DROP instruction.

This condition may be specified only for SIGNAL ON.

SYNTAX
raised if any language processing error is detected. This includes all kinds of processing errors, including true syntax errors and "run-time" errors, such as attempting an arithmetic operation on non-numeric terms. This condition may be specified only for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or delayed, and any *trapname*) of that condition trap. Thus, a SIGNAL ON HALT replaces any current CALL ON HALT, a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, any OFF reference disables the trap for CALL or SIGNAL, and so on.

Action Taken When a Condition Is Not Trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

- For HALT and SYNTAX, the execution of the program ends, and a message (see Appendix A, "Error Numbers and Messages" on page A-1) describing the nature of the event that occurred usually indicates the condition.
- For all other conditions, the condition is ignored and its state remains OFF.

Action Taken When a Condition Is Trapped

When a condition trap is currently enabled (ON) and the specified condition occurs, instead of the usual flow of control, a CALL *trapname* or SIGNAL *trapname* instruction is executed automatically (that is, passes control to a label or routine). The label or routine given control depends on whether you used the NAME *trapname* option when you enabled the condition trap.

If you did not explicitly specify a *trapname*, control is passed to the label or routine that matches the name of the *condition* itself (ERROR, FAILURE, HALT, NOTREADY, NOVALUE, or SYNTAX).

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

If you specified *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction, control is passed to the label or routine specified, rather than to the name of the *condition*.

The sequence of events, once a condition has been trapped, varies depending on whether a SIGNAL or CALL is executed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see page 3-35).

  If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it once the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then if the SIGNAL ON SYNTAX label name is not found, a normal syntax error termination occurs.

- If the action taken is a CALL, the CALL is made in the usual way (see page 3-6) except that the special variable RESULT is not affected by the call. If the routine should RETURN any data, then the returned character string is ignored.

  Note that CALL ON can only occur at clause boundaries. Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

  Before the CALL is made, the condition trap is put into a delayed state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is trapped again any action (including the updating of the condition information) is delayed until one of the following events occurs:

  1. A CALL ON or SIGNAL ON, for the delayed condition, is executed. In this case a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been executed.
  2. A CALL OFF or SIGNAL OFF, for the delayed condition, is executed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
  3. A RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

  On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

  Notes:

  1. In all cases, the condition is raised immediately upon detection. If SIGNAL ON traps the condition, the current instruction is terminated, if necessary. Therefore, the instruction during which an event occurs may be only partly executed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that the CALL for ERROR, FAILURE, HALT, and NOTREADY traps can occur only at clause boundaries. If these conditions arise in the middle of an INTERPRET instruction, execution of INTERPRET may be interrupted and later resumed. Similarly, other instructions, for example, DO or SELECT, may be temporarily interrupted by a CALL at a clause boundary.

2. The state (ON, OFF, or delayed, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction (page 3-6) for details of other information that is saved during a subroutine call.
3. The state of condition traps is not affected when an external routine is invoked by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.
4. While user input is executed during interactive tracing, all conditions are set OFF so that unexpected transfer of control does not occur should (for example) the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program, but is trapped specially and then ignored after a message is given.
5. Certain execution errors are detected by the system interface either before execution of the program starts or after the program has exited. SIGNAL ON SYNTAX cannot trap these errors.
6. If a trap is enabled using CALL ON, the routine can be an internal, built-in, or external function.

Note that **labels** are clauses consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition Information

When any condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see "CONDITION" on page 4-14).

The condition information includes:

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

The descriptive string varies, depending on the condition trapped.

ERROR The string that was processed and resulted in the error condition.

FAILURE The string that was processed and resulted in the failure condition.

HALT Any string associated with the halt request. This can be the null string if no string was provided.

NOTREADY The name of the stream being manipulated when the error occurred and the NOTREADY condition was raised. If the stream was a default stream with no defined name, then the null string may be returned.

NOVALUE The derived name of the variable whose attempted reference caused the NOVALUE condition. The NOVALUE condition trap can be enabled only using SIGNAL ON.

SYNTAX Any string the language processor associated with the error. This can be the null string if no specific string is provided. Note that the special variables RC and SIGL provide information on the nature and position of the processing error. The SYNTAX condition trap can be enabled only using SIGNAL ON.

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is saved and restored across subroutine or function calls, including one due to a CALL ON trap. A routine invoked by a CALL ON, therefore, can access the appropriate condition information. Any previous condition information is still available after the routine returns.

The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code before control is transferred to the condition label. For SIGNAL ON SYNTAX, RC is set to the syntax error number.

The Special Variable SIGL

When any transfer of control due to a SIGNAL or CALL takes place, the line number of the clause currently executing is stored in the REXX special variable SIGL. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control an editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then invoke an editor to edit the source file positioned at the line in error. Note that in this case the program has to be reinvoked before any changes made in the editor can take effect.

Alternatively, SIGL can be used to help determine the cause of an error (such as the occasional failure of a function call) as in the following example:

```
/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl':' errortext(rc)
  say sourceline(sigl)
  trace ?r; nop
```

This code first displays the error code, line number, and error message. It then displays the line in error, and finally drops into debug mode to let you inspect the values of the variables used at the line in error.

Chapter 8. Input and Output Streams

REXX defines only simple, character oriented, forms of input and output. In general, communication to or from the user is in the form of a stream of characters. These streams may be manipulated either character-by-character or line-by-line. In addition to these character streams, an external data queue is defined for inter-program communication. This queue can only be accessed on a line-by-line basis.

In this discussion, input and output will be described as though communicating with a human user, but a character stream might, in fact, have a variety of sources or destinations. These may include files, serial interfaces, displays, or networks. A character stream may therefore be:

- **transient**, or dynamic; for example, data sent or received over a serial interface

  or

- **persistent**, in a static form; for example, a file or data object.

Housekeeping for the character streams (opening and closing files, for example) is not explicitly part of the language since in most environments these operations will be automatic; however, a function is provided for miscellaneous stream commands for those operating environments that require them.

It is assumed that there is one default input stream and one default output stream.

A stream name can be a file, a named pipe, or any OS/2 device. If the name is a file, it can use a valid drive, path, and file specification, including network file names. If the stream is a named pipe, it must follow the OS/2 conventions for named pipes.

Some valid OS/2 devices for streams include:

COM1:/COM2:	Communication ports
CON:	Display screen input and output
KBD:	Keyboard input
LPT1:/LPT2:	Printer devices
PRN:	Current default printer output
STDERR:	Standard error output
STDIN:	Standard input stream
STDOUT:	Standard output stream
QUEUE:	REXX external data queue.

The default input and output streams are STNIN (the standard OS/2 input stream) and STDOUT (the standard OS/2 output stream). The appropriate default input or output stream is used when no name is specified.

Notes:

1. For additional information on specific devices, refer to the MODE command in the *OS/2 Command Reference*.
2. While the form of a stream name is necessarily defined by the OS/2 environment, it is still possible to write programs that use input and output functions and yet are effectively independent of the OS/2 program.

The Input and Output Model

The model of input and output for REXX consists of three logically distinct parts, namely:

- One or more character input streams
- One or more character output streams
- One or more external data queues.

These three elements are manipulated by the REXX instructions and built-in routines as follows.

Character Input Streams

Input to REXX programs takes the form of a serial character stream generated by user interaction, or having the characteristics of a stream so generated. Characters may be added to the end of some streams asynchronously; other streams may be static or synchronous.

The instructions that govern the use of input streams are:

- Any named input stream can be read directly as characters by the CHARIN function or as lines by the LINEIN function.
- The default input character stream (STDIN:) can be read as lines by the PULL and PARSE PULL instructions if the external data queue is empty (PULL is the same as PARSE PULL except that uppercase translation takes place).
- The PARSE LINEIN instruction can be used to read lines from the default input character stream regardless of the state of the external data queue. Normally, however, the default input stream is read by using PULL or PARSE PULL.

In a persistent stream, the REXX language processor maintains a current *read position.*

- The CHARS function returns the number of characters currently available in an input character stream from the read position through the end of the stream (including any line-end characters, if these are defined for the stream).
- The LINES function is used to determine if any data remains between the current read position and the end of the stream.
- The read position itself can be manipulated to an arbitrary point in the stream by means of the SEEK command of the STREAM function.

In a transient stream, no read position is available.

- The CHARS and LINES function can determine only if data is present in the stream. For OS/2 devices, the return value is 1 for either of these functions.
- The SEEK command of the STREAM function is not applicable to transient streams.

Character Output Streams

Character output streams provide for output from a REXX program.

- Any output stream can be written in character form with the CHAROUT function.
- Any output stream can be written as lines using the LINEOUT function.
- The default output stream (STDOUT:) can also be written as lines with the SAY instruction.

Both LINEOUT and SAY provide the appropriate line-end sequence at the end of each line. Depending on the stream being written, other modifications or formatting may be applied to output lines by the operating system or the hardware; however the output data remains a single logical line.

The current *write position* in a stream (identical to the read position) is also maintained by the REXX language processor. This position is usually the end of the stream (as for example when the stream is first opened), so that data can be appended to the end of the stream. For persistent files, however, the write position can be set to the beginning of the stream to overwrite existing data by giving a value of 1 for the `start` parameter of the CHAROUT function or for the `line` parameter of the LINEOUT function. Or, the STREAM function can be used to direct sequential output to some arbitrary point in the stream.

Note: Once data has been placed in a transient character output stream (for example, a network or serial link), it is no longer accessible to REXX. Normally the actual use of the characters in the stream, by the system, is asynchronous.

The STREAM Function

The built-in STREAM function is used to determine the state of an input or output stream and to carry out specific operations, described by *stream commands*. This stream command mechanism allows REXX programs to open and close selected streams for read-only or read and write operations, to move the read and write positions within a stream, and to access the size and the date of last update, see "STREAM" on page 4-33.

External Data Queue

The external data queue is a list of character strings that can only be accessed by line operations. It is external to REXX programs in that other REXX programs can have access to the queue.

The external data queue therefore forms a REXX-defined channel of communication between programs. Data in the queue is arbitrary; no characters have any special meaning or effect.

Apart from the explicit REXX operations described here, no detectable change to the queue occurs during the execution of a REXX program except when control leaves the program (as, for example, when an external command or routine is called). The REXX queuing operations are:

- Lines may be removed from the queue using the PULL or PARSE PULL instructions. When the queue is empty, these instructions will read lines from the default character input stream (STDIN:). In this way, the external data queue may be used as a source for user input, provided that the input is read as lines with PULL or PARSE PULL. Optionally, queue items can also be

removed by using the LINEIN function to address the queue as a device, such as `LINEIN('QUEUE:')`.

- Lines can be stacked at the head of the queue using the PUSH instruction.
- Lines can be added to the tail of the queue using the QUEUE instruction, or using the LINEOUT function to address the queue as a device, such as `LINEOUT('QUEUE:','string')`.
- The QUEUED function returns the number of lines currently in the queue as does the function call `LINES('QUEUE:')`.

Implementation

Usually, the dialog between a REXX program with the user takes place on a line-by-line basis and is therefore carried out with the SAY and PULL (or PARSE PULL) instructions. This technique considerably enhances the usability of many programs, as they may be converted to programmable dialogs by using the external data queue to provide the input normally typed by the user. The PARSE LINEIN instruction should only be used when it is necessary to bypass the external data queue.

When a dialog is not on a line-by-line basis, use the explicitly serial interfaces provided by the CHARIN and CHAROUT functions. These functions are especially important for input and output in transient character streams, such as keyboards, printers, or network environments.

Opening and closing of persistent streams, such as files, is largely automatic. Generally speaking, a stream is opened upon the first call of a line or character function and remains open until explicitly closed with the CHAROUT, LINEOUT or STREAM functions, or until the program ends. A stream can also be opened or closed explicitly. This can be done with the STREAM function, or through specific use of the other I/O functions. For example, invoking the LINEOUT function with just the name of a stream (and no output line) closes the named stream.

A stream opened by the CHARIN, CHAROUT, LINEIN or LINEOUT functions is open for both reading and writing. The STREAM function, however, can be used to open a stream for read-only or write-only operations.

Queue Interface

REXX provides queuing services entirely separate from the OS/2 Inter-Process Communications queues. The queues discussed here are solely for the use of REXX programs.

REXX queues are manipulated within a program by these instructions:

PUSH Stacks a string on top of the queue (LIFO).

QUEUE Adds a string to the tail of the queue (FIFO).

PULL Reads a string from the head of the queue. If the queue is empty, input is taken from the console (STDIN:).

To get the number of items remaining in the queue, use the function QUEUED.

Access to Queues

There are two kinds of queues in REXX. Both kinds are accessed and processed by name.

Session Queues

One *session queue* is automatically provided for each OS/2 session in operation. Its name is always SESSION and it is created by REXX the first time information is put on the queue by a program or procedure. All processes (programs and procedures) in a session can access the session queue. However, a given process can only access the session queue defined for its session and the session queue is not unique to any single process in the session.

Private Queues

Private queues are created (and deleted) by your program. You can name the queue yourself or leave the naming to REXX. In order for your program to use any queue, it must know the name of the queue.

The following is an example:

```
/*                                                                        */
                              /* push/pull WITHOUT multiprogramming support   */
/*                                                                        */
push date() time()                                  /* push date and time      */
do 1000                                             /* lets pass some time     */
  nop                                               /* doing nothing           */
end                                                 /* end of loop             */
pull a b                                            /* pull them               */
say 'Pushed at ' a b ', Pulled at ' date() time() /* say now and then        */

/*                                                                        */
/*              push/pull WITH multiprogramming support                   */
/*            (no error recovery, or unsupported env tests)               */
/*                                                                        */
newq = RXQUEUE('Create')                            /* create a unique queue   */
oq = RXQUEUE('Set',newq)                            /* establish new queue     */
push date() time()                                  /* push date and time      */
do 1000                                             /* lets spend some time    */
  nop                                               /* doing nothing           */
end                                                 /* end of loop             */
pull a b                                            /* get pushed info         */
say 'Pushed at ' a b ', Pulled at ' date() time() /* tell user               */
call RXQUEUE 'Delete',newq                     /* destroy unique queue created */
call RXQUEUE 'Set',oq                /* reset to default queue (not required)  */
```

Figure 8-1. Sample REXX Procedure Using a Queue

Special Considerations

1. External programs that must communicate with a REXX procedure by means of defined data queues can use the default queue or the session queue, or they can receive the data queue name by some interprocess communication technique. This could include: parameter passing, placement on a prearranged logical queue, or use of normal OS/2 Inter-Process Communication mechanisms (for example, pipes, shared memory, or IPC queues).

2. Named queues are available across the entire system; therefore, the names of queues must be unique within the system. If a queue named `os2que` exists and this function is issued:

   ```
   newqueue = RXQUEUE('Create', 'OS2QUE')
   ```

   a new queue is created with a randomly chosen name, and *newqueue* is assigned that new name.

Detached Processes

1. Detached processes will access a *detached* session queue that is unique for each detached process. Note, however, that this detached session queue is *not* the same as the session queue of the starting session.
2. REXX programs that are to be run as detached processes cannot perform any PULL or PARSE PULL instructions that involve terminal I/O. However, PULL and PARSE PULL instructions that act on a queue are permitted in detached processes.

Multi-Programming Considerations

This data queue mechanism differs from the OS/2 standard API queueing in the following ways:

1. The queue is *not* owned by a specific process and as such any process is entitled to modify the queue at any time. The operations that effect the queue are atomic, in that the resource is serialized by the subsystem such that no data integrity problems can be encountered.

   However, synchronization of requests such that two processes accessing the same queue get the data in the order it was placed on the queue is a *user* responsibility.
2. Such a queue is defined as an *element copy data queue*.
3. A regular OS/2 IPC queue is owned (created) by a specific process. When that process terminates, the queue is destroyed. Conversely, the queues created by the `RxQueue('Create', queuename)` call will exist until *explicitly* deleted. Termination of a program or procedure that created a private queue does not force the deletion of the private queue. Any data on the queue when the process *creating* it terminates will remain on the queue until either the queue is deleted, by way of the REXX function call `RxQueue('Delete', queuename)`, or until the data is read.

Data queues *must be explicitly deleted* by some procedure or program (not necessarily the creator). Deletion of a queue with remaining items, destroys those items.

Errors During Input and Output

The REXX language offers considerable flexibility in the handling of errors during input or output. It is provided in the form of a NOTREADY condition that may be trapped by the CALL ON and SIGNAL ON instructions and further information can be elicited by the STREAM function. See Chapter 7, "Conditions and Condition Traps," for a more detailed discussion of SIGNAL ON and CALL ON.)

When an error occurs during an input or output operation, the function being called will normally continue without interruption (with, for example, a nonzero count being returned by an output function). Depending on the nature of the operation, a

program has the option of raising the NOTREADY condition. The NOTREADY condition is similar to the ERROR and FAILURE conditions associated with commands in that it does not cause a terminating error if the condition is raised but is not trapped.

Once NOTREADY has been raised, the following possibilities exist:

- The NOTREADY condition is not being trapped; in this case execution continues without interruption; the NOTREADY condition remains in the OFF state.
- The NOTREADY condition is being trapped by SIGNAL ON NOTREADY; in this case, the NOTREADY condition is raised, execution of the current clause ceases immediately, and the SIGNAL takes place as usual for condition traps.
- The NOTREADY condition is being trapped by CALL ON NOTREADY; in this case the NOTREADY condition is raised, but execution of the current clause is not halted. The NOTREADY condition is put into the delayed state, and execution continues until the end of the current clause. While execution continues, input functions that refer to the same stream may return the null string and output functions may return an appropriate count, depending on the form and timing of the error. At the end of the current clause, the CALL takes place as usual for condition traps.
- The NOTREADY condition is being trapped (by CALL ON NOTREADY) but is already in the DELAY state (due to NOTREADY already having been raised); in this case execution continues, and the NOTREADY condition remains in the DELAY state.

Once the NOTREADY condition has been raised and is in DELAY state, the CONDITION function will return (for a *description* invocation) the name of the stream being processed when the stream error occurred. If the stream is a default stream and has no defined name, then the null string may be returned in this case.

The STREAM function will then usually show that the state of the stream is ERROR or NOTREADY, and additional information on the state of the stream will normally be available by way of the *description* option of the STREAM function.

Examples of Input and Output

In most circumstances, communication with a user running a REXX program will be by way of the default input and output streams. For a question and answer dialog, the recommended technique is to use the SAY and PULL instructions (using PARSE PULL if case-sensitive input is required).

More generally, though, it is necessary to write to or read from streams other than the default. For example, to copy the contents of one file to another one might use the following program:

```
/* FILECOPY.CMD */
/* This routine copies the stream or file named by  */
/* the first argument to the stream or file named   */
/* by the second, as lines.                         */
parse arg inputname, outputname

signal on notready

do forever
  call lineout outputname, linein(inputname)
end

notready:
```

As long as lines remain in the named input stream, a line is read and is then immediately written out to the named output stream. It is easy to modify this program so that it filters the lines in some way before they are written.

To illustrate how character and line operations might be mixed in a communications program, consider the following example in which a character stream is converted into lines:

```
/* COLLECT.CMD */
/* This routine collects characters from the stream */
/* named by the first argument until a line is      */
/* complete, and then places the line on the        */
/* external data queue.                             */
/* The second argument is the single character that */
/* identifies the end of a line.                    */
parse arg inputname, lineendchar

buffer=''       /* zero-length character accumulator */
do forever
  nextchar=charin(inputname)
  if nextchar=lineendchar then leave
  buffer=buffer||nextchar          /* add to buffer */
  end
queue buffer /* place it on the external data queue */
```

Here each line is built up in a variable called BUFFER. When the line is complete (for example, when the Enter key is pressed) the loop is ended and the contents of BUFFER are placed on the external data queue. The program then ends.

Summary of Instructions and Functions

CHARIN Reads zero or more characters from a character input stream. A start position may be specified for persistent streams. See "CHARIN" on page 4-11.

CHAROUT Writes zero or more characters to a character output stream. A start position may be specified for persistent streams. See "CHAROUT" on page 4-12.

CHARS Returns the number of characters currently remaining in a character input stream. See "CHARS" on page 4-13.

LINEIN Reads zero or one line from a character input stream. See "LINEIN" on page 4-25.

LINEOUT Writes zero or one line to a character output stream. See "LINEOUT" on page 4-27.

LINES Returns 1 if any data currently remains in a character input stream. See "LINES" on page 4-28.

PARSE LINEIN Reads one line from the default character input stream. See "PARSE LINEIN" on page 3-25.

PARSE PULL Reads one line from the external data queue. If the queue is empty it reads a line from the default character input stream instead. See "PARSE PULL" on page 3-25.

PULL — The same as PARSE PULL except that the string read is translated to uppercase. See "PULL" on page 3-29.

PUSH — Writes one line to the head of the external data queue, as in a stack. See "PUSH" on page 3-30.

QUEUE — Writes one line to the tail of the external data queue. See "QUEUE" on page 3-31.

QUEUED — Returns the number of lines currently available in the external data queue. See "QUEUED" on page 4-30.

SAY — Writes one line to the default character output stream. See "SAY" on page 3-33.

STREAM — Returns a string describing the state of, or the result of an operation upon, a named character stream. See "STREAM" on page 4-33.

Chapter 9. Application Programming Interface

Other chapters of this book are addressed to professional systems and application programmers who are writing REXX programs. This chapter is addressed to professional systems and application programmers involved with the following:

- Interfacing applications to REXX
- Extending REXX language facilities by writing external functions.

Programmers attempting these tasks have typically written applications in the C language, or a similar language. They will, therefore, be familiar with the interface conventions described here. Those programmers not involved in these tasks can refer to the other sections of this book for an understanding of the REXX language.

This chapter describes:

- RXSTRINGs
- Invoking the REXX Interpreter
- Subcommand Handlers
- External Functions
- System Exits
- Variable Pool Interface
- Macrospace Interface
- Halt and Trace Functions

In this chapter, the term *application* refers to programs written in languages other than REXX. The features described here allow an application to extend many parts of the REXX language or extend an application with REXX. This includes creating handlers for subcommands, external functions and system exits.

Subcommands are commands issued from a REXX program. A REXX expression is evaluated and the result is passed as a command to the currently "addressed" subcommand handler. Subcommands are used in REXX programs running as application macros.

Functions are direct extensions of the REXX language. An application can create functions that extend the native REXX function set. Functions may be general purpose extensions or specific to an application.

System Exits are programmer-defined variations of the operating system. The application programmer can tailor the REXX interpreter behavior by replacing OS/2 for REXX system requests.

Subcommand, function and exit handlers have similar coding, compilation and packaging characteristics.

In addition, applications can manipulate the variables in REXX programs (the Variable Pool Interface), and execute REXX routines directly from memory (the Macrospace Interface).

General Characteristics

The basic requirements for subcommand, function and system exit handlers are:

- REXX handlers must use the system linkage convention. Handler functions should be declared with the appropriate type definition from the REXXSAA.H include file:
  - RexxSubcomHandler
  - RexxFunctionHandler
  - RexxExitHandler
- A REXX handler must be packaged as either:
  - An exported routine within an OS/2 Dynamic Link Library (a *dynalink* or DLL)
  - An entry point within an executable (EXE) module.
- A handler must be registered with REXX before it can be used. REXX uses the registration information to locate and call the handler. For example, external function registration of a dynamic link library external function identifies both the dynamic link library and routine that contains the external function. Also note:
  - Dynamic link library handlers are global to the OS/2 system; they can be called from any REXX program.
  - EXE file handlers are local to the registering process; handlers packaged within an EXE module can only be called by a REXX program running in the same process as the EXE module.

RXSTRINGs

Many of the REXX interfaces pass REXX character strings to and from a REXX procedure. The RXSTRING data structure is used to describe REXX character strings. An RXSTRING is a content-insensitive, flat model character string with a theoretical maximum length of 4 gigabytes. The following structure defines an RXSTRING:

```
typedef struct {
   ULONG          strlength;          /*  length of string          */
   PCH            strptr;             /*  pointer to string         */
} RXSTRING;

typedef RXSTRING *PRXSTRING;          /* pointer to an RXSTRING      */
```

Figure 9-1. RXSTRING Data Structure

Notes:

1. The REXXSAA.H include file contains a number of convenient macros for setting and testing RXSTRING values.
2. An RXSTRING may have a value (including the null string, "") or it may be empty.
   - If an RXSTRING has a value, the *strptr* field will be non-NULL. The RXSTRING macro RXVALIDSTRING(string) will return TRUE.
   - If an RXSTRING is the REXX null string (""), the *strptr* field will be non-NULL and the *strlength* field will be zero. The RXSTRING macro RXZEROLENSTRING(string) will return TRUE.
   - If an RXSTRING is empty, the field *strptr* will be NULL. The RXSTRING macro RXNULLSTRING(string) will return TRUE.
3. When the REXX interpreter passes an RXSTRING to a subcommand handler, external function, or exit handler, the interpreter adds a null character (hexadecimal zero) at the end of the RXSTRING data. The C string library functions can be used on these strings. However, the RXSTRING data may also contain null characters. There is no guarantee that the first null character encountered in an RXSTRING marks the end of the string. The C string functions should only be used when null characters are not expected in the RXSTRINGs (such a file names passed to external functions). The *strlength* field in the RXSTRING does not include the terminating null character.
4. On calls to subcommand and external functions handlers, as well as some of the exit handlers, the REXX interpreter expects an RXSTRING value returned. The REXX interpreter provides a default RXSTRING with a *strlength* of 256 for the returned information. If the returned data is shorter than 256 characters, the handler can copy the data into the default RXSTRING and set the *strlength* field to the length returned.

   If the returned data is longer than 256 characters, a new RXSTRING can be allocated using DosAllocMem. The *strptr* field must point to the new storage and the *strlength* must be set to the string length. The REXX interpreter will return the newly allocated storage to the system for the handler routine.

Invoking the REXX Interpreter

A REXX program may be executed directly by the operating system or from within an application program.

From the OS/2 operating system

The standard OS/2 CMD.EXE command shell calls the REXX interpreter for the user:

- at OS/2 command prompts
- in calls from CMD (batch) files

  Note: Use the OS/2 CALL command to invoke a REXX program in a batch file if you want control to return to the caller.
- from the object that represents the program.

From Within an Application

The REXX interpreter is an OS/2 dynamic link library (DLL) routine. Any application may call the REXX interpreter to execute a REXX program. The interpreter is fully re-entrant and supports REXX procedures running on multiple threads within the same process.

A C-language prototype for calling REXX is in the OS/2 Developer's Toolkit REXXSAA.H include file.

The RexxStart Function

RexxStart

RexxStart invokes the REXX interpreter to execute a REXX procedure.

RexxStart (ArgCount, ArgList, ProgramName, Instore, EnvName, CallType, Exits, ReturnCode, Result)

Parameters

ArgCount *(LONG) - input*
The number of elements in the *ArgList* array. This is the value that will be returned by the *ARG*() built-in function in the REXX program. *ArgCount* includes RXSTRINGs which represent omitted arguments. Omitted arguments will be empty RXSTRINGs (*strptr* will be NULL).

ArgList *(PRXSTRING) - input*
An array of RXSTRING structures that are the REXX program arguments.

ProgramName *(PSZ) - input*
Address of the ASCIIZ name of the REXX procedure. If *Instore* is NULL, string must contain at least the file name of the REXX procedure. An extension, drive, and path specification may also be provided. If a file extension is not specified, a default of ".CMD" is supplied. A REXX program can use any extension. If the path and drive are not provided, the REXX interpreter uses the normal OS/2 file search (current directory, then environment path).

If *Instore* is not NULL, *ProgramName* is the name used in the PARSE SOURCE instruction. If *Instore* requests a REXX procedure from the macrospace, *ProgramName* is the macrospace function name. (see "Macrospace Interface" on page 9-56).

Instore *(PRXSTRING) - input*
An array of two (2) RXSTRING descriptors for in-storage REXX procedures. If the *strptr* fields of both RXSTRINGs are NULL, the interpreter searches for REXX procedure *ProgramName* in the REXX macrospace (see "Macrospace Interface" on page 9-56). If the procedure is not in the macrospace, the call to RexxStart terminates with an error return code.

If either *Instore strptr* field is not NULL, *Instore* is used to execute a REXX procedure directly from storage.

Instore[0] An RXSTRING describing a memory buffer containing the REXX procedure source. The source must be an exact image of a REXX procedure disk file (complete with carriage returns, line feeds, and end-of-file characters).

Instore[1] An RXSTRING containing the tokenized image of the REXX procedure. If *Instore[1]* is empty, the REXX interpreter will return the tokenized image in *Instore[1]* when the REXX procedure finishes executing. The tokenized image may be used in *Instore[1]* on subsequent RexxStart calls.

If *Instore[1]* is not empty, interpreter will execute the tokenized image directly. The program source provided in *Instore[0]* is only used if the REXX procedure uses the SOURCELINE built-in function. *Instore[0]* may be empty if SOURCELINE is not used. If *Instore[0]* is empty and the SOURCELINE built-in function is used, SOURCELINE will return null strings for the REXX procedure source lines.

If *Instore[1]* is not empty, but does not contain a valid REXX tokenized image, unpredictable results can occur. The REXX interpreter may be be able to determine that the tokenized image is incorrect and retokenize the source.

Instore[1] is both an input and an output parameter.

If the procedure is executed from disk, the *Instore pointer* must be NULL. If the first argument string in *Arglist* contains the string "//T" and the *CallType* is RXCOMMAND, the interpreter will tokenize the procedure source and return the tokenized image without running the program.

The program calling RexxStart must release *Instore[1]* using DosFreeMem when the tokenized image is no longer needed.

The format of the tokenized image of a REXX program is not a programming interface. The tokenized image can only be executed by the same interpreter version used to create the image. Therefore, a tokenized image should not be moved to other OS/2 systems or saved for later use. The tokenized image may, however, be used multiple times during a single application instance.

EnvName *(PSZ) - input*
Address of the ASCIIZ initial ADDRESS environment name. The ADDRESS environment is a subcommand handler registered using RexxRegisterSubcomExe or RexxRegisterSubcomDll. *EnvName* is used as the initial setting for the REXX ADDRESS instruction.

If *EnvName* is NULL, the file extension is used as the initial ADDRESS environment. The environment name cannot be longer than 250 characters.

CallType *(LONG) - input*
The type of REXX procedure execution. Allowed execution types are:

`RXCOMMAND` The REXX procedure is an OS/2 or application command. REXX commands normally have a single argument string. The REXX PARSE SOURCE instruction will return `COMMAND` as the second token.

`RXSUBROUTINE` The REXX procedure is a subroutine of another program. The subroutine may have multiple arguments and does not need to return a result. The REXX PARSE SOURCE instruction will return `SUBROUTINE` as the second token.

RXFUNCTION — The REXX procedure is a function called from another program. The subroutine may have multiple arguments and must return a result. The REXX PARSE SOURCE instruction will return FUNCTION as the second token.

Exits *(PRXSYSEXIT) - input*
An array of RXSYSEXIT structures defining exits the REXX interpreter will use. The RXSYSEXIT structures have the following form:

```
typedef struct {
   PSZ          sysexit_name;      /* name of exit handler        */
   LONG         sysexit_code;      /* system exit function code   */
} RXSYSEXIT;
```

Figure 9-2. RXSYSEXIT Data Structure

The *sysexit_name* is the address of an ASCIIZ exit handler name registered with RexxRegisterExitExe or RexxRegisterExitDll. *sysexit_code* is a code identifying the handler exit type. See "System Exits" on page 9-26 for exit code definitions. The system exit list end is identified by an RXENDLST entry. *Exits* must be NULL if exits are not used.

ReturnCode *(PLONG) - output*
The integer form of the *Result* string. If the *Result* string is a whole number in the range -(2**15) to 2**15-1, it will be converted to an integer and and also returned in *ReturnCode.*

Result *(PRXSTRING) - output*
The string returned from the REXX procedure with the REXX RETURN or EXIT instruction. A default RXSTRING may be provided for the returned result. If a default RXSTRING is not provided or the default is too small for the returned result, the REXX interpreter will allocate an RXSTRING using DosAllocMem. The caller of RexxStart is responsible for releasing the RXSTRING storage with DosFreeMem.

The REXX interpreter does not add a terminating null to *Result.*

Returns

The possible RexxStart return codes are:

negative — Interpreter errors. See Appendix A, "Error Numbers and Messages" on page A-1 for the list of REXX errors.

0 — No errors occurred. The REXX procedure executed normally.

positive — An OS/2 return code indicating problems finding or loading the interpreter. See the return codes for the OS/2 functions DosLoadModule and DosQueryProcAddr for details.

When a called macrospace REXX procedure is not loaded in the macrospace, the return code is -3 ("Program is unreadable").

Example

```
LONG      return_code;                /* interpreter return code    */
RXSTRING  argv[1];                    /* program argument string    */
RXSTRING  retstr;                     /* program return value       */
LONG      rc;                         /* converted return code      */
CHAR      return_buffer[250];         /* returned buffer            */

                                      /* build the argument string  */
  MAKERXSTRING(argv[0], macro_argument,
      strlen(macro_argument));
                                      /* set up default return      */
  MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));

  return_code = RexxStart(1,          /* one argument               */
                         argv,        /* argument array             */
                         "CHANGE.ED", /* REXX procedure name        */
                         NULL,        /* use disk version           */
                         "Editor",    /* default address name       */
                         RXCOMMAND,   /* calling as a subcommand    */
                         NULL,        /* no exits used              */
                         &rc,         /* converted return code      */
                         &retstr);    /* returned result            */

                                      /* process return value       */
 :
                                      /* need to return storage?    */
  if (RXSTRPTR(retval) != return_buffer)
    DosFreeMem(RXSTRPTR(retval));     /* release the RXSTRING       */
```

Figure 9-3. Sample Call to the REXX Interpreter

Subcommand Interfaces

An application can create named handlers to process commands from a REXX programs. Once created, the subcommand handler name can be used with the RexxStart function or the REXX ADDRESS instruction. Subcommand handlers must be registered with the RexxRegisterSubcomExe or RexxRegisterSubcomDll function before use.

Registering Subcommand Handlers

A subcommand handler can reside in the same module (EXE or DLL) as an application, or it can reside in a separate dynamic link library. An application that executes REXX procedures with RexxStart should use RexxRegisterSubcomExe to register subcommand handlers. The REXX interpreter passes commands to the application subcommand handler entry point. Subcommand handlers created using RexxRegisterSubcomExe are available only to REXX programs invoked from the registering application.

The RexxRegisterSubcomDll interface creates subcommand handlers which reside in a dynamic link library. A dynamic link library subcommand handler can be accessed by any REXX program using the REXX ADDRESS instruction. A dynamic link library subcommand handler can also be registered directly from a REXX program using the RXSUBCOM command.

Creating Subcommand Handlers

The following example is a sample subcommand handler definition.

```
ULONG command_handler(
  PRXSTRING Command,    /* Command string from REXX              */
  PUSHORT   Flags,      /* Returned Error/Failure flags          */
  PRXSTRING Retstr);    /* Returned RC string                    */
```

Figure 9-4. Sample Definition of a Subcommand Handler

Where:

Command The command string created by REXX.

command is a null-terminated RXSTRING containing the issued command.

Flags Subcommand completion status. The subcommand handler can indicate success, error, or failure status. The subcommand handler can set *Flags* to one of the following value:

RXSUBCOM_OK
The subcommand completed normally. No errors occurred during subcommand processing and the REXX procedure will continue when the subcommand handler returns.

RXSUBCOM_ERROR
A subcommand error occurred. RXSUBCOM_ERROR indicates a subcommand error occurred, for example, incorrect command options or syntax.

If the subcommand handler sets *Flags* to RXSUBCOM_ERROR, the REXX interpreter will raise an ERROR condition if SIGNAL ON ERROR or CALL ON ERROR traps have been created. If TRACE

ERRORS has been issued, REXX will trace the command when the subcommand handler returns.

RXSUBCOM_FAILURE
A subcommand failure occurred. RXSUBCOM_FAILURE indicates that general subcommand processing errors have occurred. For example, unknown commands normally return RXSUBCOM_FAILURE.

If the subcommand handler sets *Flags* to RXSUBCOM_FAILURE, the REXX interpreter will raise a FAILUREcondition if SIGNAL ON FAILURE or CALL ON FAILURE traps have been created. If TRACE FAILURES has been issued, REXX will trace the command when the subcommand handler returns.

Retstr Address of an RXSTRING for the return code. *Retstr* is a character string return code that will be assigned to the REXX special variable RC when the subcommand handler returns to REXX. The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING may allocated with DosAllocMem if the return string is longer than the default RXSTRING. If the subcommand handler sets *Retval* to an empty RXSTRING (a NULL *strptr*), REXX will assign the string "0" to RC.

Example

```
ULONG Edit_Commands(
  PRXSTRING Command,    /* Command string passed from the caller    */
  PUSHORT   Flags,      /* pointer to short for return of flags     */
  PRXSTRING Retstr)     /* pointer to RXSTRING for RC return        */
{
  LONG      command_id;                 /* command to process        */
  LONG      rc;                         /* return code               */
  PSZ       scan_pointer;               /* current command scan      */
  PSZ       target;                     /* general editor target     */

  scan_pointer = command->strptr;       /* point to the command      */
                                        /* resolve command           */
  command_id = resolve_command(&scan_pointer);

  switch (command_id) {                 /* process based on command  */

    case   LOCATE:                      /* locate command            */

                                        /* validate rest of command  */
      if (rc = get_target(&scan_pointer, &target)) {
        *Flags = RXSUBCOM_ERROR;        /* raise an error condition  */
        break;                          /* return to REXX            */
      }
      rc = locate(target);              /* look target in the file   */
      *Flags = RXSUBCOM_OK;             /* not found is not an error */
      break;                            /* go finish up              */

:

    default:                            /* unknown command           */
      rc = 1;                           /* return code for unknown   */
      *Flags = RXSUBCOM_FAILURE;        /* this is a command failure */
      break;
  }

  sprintf(Retstr->strptr, "%d", rc);    /* format return code string */
                                        /* and set the correct length */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                             /* processing completed      */
}
```

Figure 9-5. Sample Subcommand Handler

Subcommand Interface Functions

The functions for registering and using subcommand handlers are:

RexxRegisterSubcomDll

RexxRegisterSubcomDll registers a subcommand handler that resides in a dynamic link library routine.

RexxRegisterSubcomDll (EnvName, ModuleName, EntryPoint, UserArea, DropAuth)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the subcommand handler routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynamic link library procedure name. *EntryPoint* is the name of the exported routine within *ModuleName* that REXX will call as a subcommand handler.

UserArea *(PUCHAR) - input*
Address of an eight-byte area of user defined information. The eight-bytes addressed by *UserArea* will be saved with the subcommand handler registration. *UserArea* may be NULL if there is no user information to save. The saved user information can be retrieved with the RexxQuerySubcom function.

DropAuth *(ULONG) - input*
The drop authority. *DropAuth* identifies the processes that can deregister the subcommand handler. The possible *DropAuth* values are:

RXSUBCOM_DROPPABLE
Any process can deregister the subcommand handler with RexxDeregisterSubcom.

RXSUBCOM_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterSubcom.

Returns

0	RXSUBCOM_OK
10	RXSUBCOM_DUP
1002	RXSUBCOM_NOEMEM
1003	RXSUBCOM_BADTYPE

Remarks

EntryPoint may be either a 16-bit or a 32-bit routine. REXX will invoke the handler in the correct addressing mode.

RexxRegisterSubcomExe

RexxRegisterSubcomExe registers a subcommand handler that resides within application code.

RexxRegisterSubcomExe (EnvName, EntryPoint, UserArea)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ subcommand handler name.

EntryPoint *(PFN) - input*
Address of the subcommand handler entry point within the application EXE code.

UserArea *(PUCHAR) - input*
Address of an eight-byte area of user defined information. The eight-bytes addressed by *UserArea* will be saved with the subcommand handler registration. *UserArea* may be NULL if there is no user information to save. The user information can be retrieved with the RexxQuerySubcom function.

Returns

0	RXSUBCOM_OK
10	RXSUBCOM_DUP
30	RXSUBCOM_NOTREG
1002	RXSUBCOM_NOEMEM
1003	RXSUBCOM_BADTYPE

Remarks

If *EnvName* is same as a subcommand handler already registered with RexxRegisterSubcomDll, RexxRegisterSubcomExe will return RXSUBCOM_DUP. This is not an error condition. RexxRegisterSubcomExe has successfully registered the new subcommand handler.: A REXX procedure can register dynamic link library subcommand handlers with the RXSUBCOM command. For example:

```
                                /* register Dialog Manager      */
                                /* subcommand handler           */
'RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR'
Address ispcir                  /* send commands to dialog mgr  */
```

The RXSUBCOM command registers the Dialog Manager subcommand handler ISPCIR as routine ISPCIR in the ISPCIR dynamic link library.

Example

```
WORKAREARECORD  *user_info[2];        /* saved user information     */

user_info[0] = global_workarea;       /* save global work area for  */
user_info[1] = NULL;                  /* re-entrancy                */

rc = RexxRegisterSubcomExe("Editor", /* register editor handler    */
    &Edit_Commands,                   /* located at this address    */
    user_info);                       /* save global pointer        */
```

Figure 9-6. Sample Subcommand Handler Registration

RexxDeregisterSubcom

RexxDeregisterSubcom deregisters a subcommand handler.

RexxDeregisterSubcom (EnvName, ModuleName)

Parameters

EnvName *(PSZ) - input*
: Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
: Address of an ASCIIZ dynalink library name. *ModuleName* is the name of the dynalink library containing the registered subcommand handler. When *ModuleName* is NULL, RexxDeregisterSubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxDeregisterSubcom does not find a RexxRegisterSubcomExe handler, it will search the RexxRegisterSubcomDll subcommand handler list.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
40	RXSUBCOM_NOCANDROP
1003	RXSUBCOM_BADTYPE

Remarks

The handler is removed from the active subcommand handler list.

RexxQuerySubcom

RexxQuerySubcom queries a subcommand handler and retrieves saved user information.

RexxQuerySubcom (EnvName, ModuleName, Flag, UserWord)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ subcommand handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* restricts the query to a subcommand handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxQuerySubcom searches the RexxRegisterSubcomExe subcommand handler list for a handler within the current process. If RexxQuerySubcom does not find a RexxRegisterSubcomExe handler, it will search the RexxRegisterSubcomDll subcommand handler list.

Flag *(PUSHORT) - output*
Subcommand handler registration flag. *Flag* is the *EnvName* subcommand handler registration status. When RexxQuerySubcom returns RXSUBCOM_OK, the *EnvName* subcommand handler is currently registered. When RexxQuerySubcom returns RXSUBCOM_NOTREG, the *EnvName* subcommand handler is not registered.

UserWord *(PUCHAR) - output*
Address of an eight-byte area to receive the user information saved with RexxRegisterSubcomExe or RexxRegisterSubcomDll. *UserWord* can be NULL if the saved user information is not required.

Returns

0	RXSUBCOM_OK
30	RXSUBCOM_NOTREG
1003	RXSUBCOM_BADTYPE

Example

```
ULONG Edit_Commands(
  PRXSTRING Command,     /* Command string passed from the caller    */
  PUSHORT   Flags,       /* pointer to short for return of flags     */
  PRXSTRING Retstr)      /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information    */
  WORKAREARECORD   global_workarea;    /* application data anchor   */
  USHORT           query_flag;         /* flag for handler query    */


  rc = RexxQuerySubcom("Editor",       /* retrieve application work */
      NULL,                            /* area anchor from REXX.    */
      &query_flag,
      user_info);

  global_workarea = user_info[0];      /* set the global anchor     */
```

Figure 9-7. Sample Subcommand Handler Query

Return Codes

RXSUBCOM_ERROR	0x01	An error in subcommand execution has occurred; the interpreter raises an ERROR condition.
RXSUBCOM_FAILURE	0x02	A failure in subcommand execution has occurred; the interpreter raises a FAILURE condition.
RXSUBCOM_NOEMEM	1002	There is insufficient memory available to complete this request.
RXSUBCOM_OK	0	A subcommand has execute successfully.
RXSUBCOM_DUP	10	A duplicate handler name has been successfully registered; there is either: • an EXE handler with the same name registered in another process, or • a DLL handler with the same name registered in another DLL; to address this subcommand, its library name must be specified.
RXSUBCOM_NOTREG	30	This indicates: • registration was unsuccessful due to duplicate handler and dynalink names (RexxRegisterSubcomExe or RexxRegisterSubcomDll) • the subroutine environment is not registered (other REXX subcommand functions).
RXSUBCOM_NOCANDROP	40	The subcommand handler has been registered as "not droppable."
RXSUBCOM_LOADERR	50	An error has occurred while loading a dynalink library; most commonly caused by a missing dynalink library file.
RXSUBCOM_NOPROC	127	The registered subcommand handler routine was not found; ensure the dynalink routine name has been exported during linking.

External Functions

There are two types of REXX external functions:

1. Routines written in REXX
2. Routines written in other OS/2-supported languages.

External functions written in the REXX language are not registered with REXX. the REXX functions are found by a disk search for a REXX procedure file that matches the function name. Functions written in other languages, however, must be registered with the REXX interpeter.

Registering External Functions

An external function can reside in the same module (EXE or DLL) as an application, or it can reside in a separate dynamic link library. RexxRegisterFunctionExe registers external functions within an application module. External functions registered with RexxRegisterFunctionExe are available only to REXX programs invoked from the registering application.

The RexxRegisterFunctionDll interface registers external functions that reside in a dynamic link library. Once registered, a dynamic link library external function can be accessed by any REXX program. A dynamic link library external function can also be registered directly from a REXX program using the REXX RxFuncAdd built-in function.

Creating External Functions

The following is a sample external function definition:

```
LONG SysLoadFuncs(
     PSZ       Name,                    /* name of the function       */
     LONG      Argc,                    /* number of arguments        */
     RXSTRING  Argv[],                  /* list of argument strings   */
     PSZ       Queuename,               /* current queue name         */
     PRXSTRING Retstr)                  /* returned result string     */
```

Figure 9-8. Sample External Function Definition

Where

Name Address of ASCIIZ function name used to call the external function.

Argc The size of the argument list. *Argv* will contain *Argc* RXSTRINGs.

Argv An array of null-terminated RXSTRINGs for the function arguments.

Queue The name of the currently defined REXX external data queue.

Retstr Address of an RXSTRING for the returned value. *Retstr* is a character string function or subroutine return value. When a REXX program calls an external function with the REXX CALL instruction, *Retstr* is assigned to the REXX special variable RESULT. When the REXX program calls an external function with a function call, *Retstr* is used directly within the REXX expression.

The REXX interpreter provides a default 256-byte RXSTRING in *Retstr*. A longer RXSTRING may allocated with DosAllocMem if the returned string is longer name 256 bytes. The REXX interpreter

releases *Retstr* with DosFreeMem when the external function completes.

Returns An integer return code from the function. When the external function returns zero, the function completed successfully. *Retstr* contains the function return value. When the external function returns a non-zero, the REXX interpreter raises REXX error 40 ("Invalid call to routine"). The *Retstr* value is ignored.

If the external function does not have a return value, the function should set *Retstr* to an an empty RXSTRING (NULL *strptr*). When an external function called as a function does not return a value, the interpreter raises error 44, "Function did not return data." When an external function called with the REXX CALL instruction does not return a value, the REXX interpreter drops (unassigns) the special variable RESULT.

Calling External Functions

RexxRegisterFunctionExe external functions are local to the registering process. Only REXX procedures running in the same process can call the registered external function. It is possible to register functions with the same external function name if they are registered from different processes. However, RexxRegisterFunctionDll functions are available from all processes. The function names cannot be duplicated.

Example

```
LONG SysMkDir(
     PSZ       Name,                    /* name of the function        */
     LONG      Argc,                    /* number of arguments         */
     RXSTRING  Argv[],                  /* list of argument strings    */
     PSZ       Queuename,               /* current queue name          */
     PRXSTRING Retstr)                  /* returned result string      */
{
  ULONG  rc;                            /* Return code of function     */

  if (Argc != 1)                        /* must be 1 argument          */
    return 40;                          /* incorrect call if not       */

                                        /* make the directory using    */
                                        /* the null-terminated         */
  rc = DosMkDir(Argv[0].strptr, 0L);    /* directly                    */

  sprintf(Retstr->strptr, "%d", rc);    /* result is return code       */
                                        /* set proper string length    */
  Retstr->strlength = strlen(Retstr->strptr);
  return 0;                             /* successful completion       */
}
```

Figure 9-9. Sample External Function Routine

External Function Interface Functions

The functions for registering and using external functions are:

RexxRegisterFunctionDll

RexxRegisterFunctionDll registers an external function that resides in a dynamic link library routine.

RexxRegisterFunctionDll (FuncName, ModuleName, EntryPoint)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the external function routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynamic link procedure name. *EntryPoint* is exported external function routine within *ModuleName*. *FuncName*.

Returns

0	RXFUNC_OK
10	RXFUNC_DEFINED
20	RXFUNC_NOMEM

Remarks

EntryPoint may be either a 16-bit or 32-bit routine. REXX will invoke the function in the correct addressing mode.: A REXX procedure can register dynamic link library subcommand handlers with the RxFuncAdd built-in function. For example:

```
                               /* register function SysLoadFuncs*/
                               /* in dynalink library REXXUTIL  */
Call RxFuncAdd 'SysLoadFuncs', 'REXXUTIL', 'SysLoadFuncs'
Call SysLoadFuncs              /* call to load other functions  */
```

The RxFuncAdd registers the external function SysLoadFuncs as routine SysLoadFuncs in the REXXUTIL dynamic link library. SysLoadFuncs registers additional functions in REXXUTIL.DLL with RexxRegisterFunctionDll. See the SysLoadFuncs routine below for a function registration example.

Example

```
static PSZ  RxFncTable[] =            /* function package list      */
{
      "SysCls",
      "SysCurpos",
      "SysCurState",
      "SysDriveInfo",
}


LONG RexxFunctionHandler SysLoadFuncs(
     PSZ       Name,                  /* name of the function       */
     LONG      Argc,                  /* number of arguments        */
     RXSTRING  Argv[],                /* list of argument strings   */
     PSZ       Queuename,             /* current queue name         */
     PRXSTRING Retstr)                /* returned result string     */
{
  INT    entries;                     /* Num of entries             */
  INT    j;                           /* Counter                    */

  Retstr->strlength = 0;              /* set null string return     */

  if (Argc > 0)                       /* check arguments            */
    return 40;                        /* too many, raise an error   */

                                      /* get count of arguments     */
  entries = sizeof(RxFncTable)/sizeof(PSZ);
                                      /* register each function in  */
  for (j = 0; j < entries; j++) {     /* the table                  */
    RexxRegisterFunctionDll(RxFncTable[j],
          "REXXUTIL", RxFncTable[j]);
  }
  return 0;                           /* successful completion      */
}
```

Figure 9-10. Function Package Load Routine

RexxRegisterFunctionExe

RexxRegisterFunctionExe registers an external function that resides within application code.

RexxRegisterFunctionExe (FuncName, EntryPoint)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name.

EntryPoint *(PFN) - input*
Address of the external function entry point within the application EXE file. Functions registered with RexxRegisterFunctionExe are *local* to the current process. REXX procedures in the same process as the RexxRegisterFunctionExe issuer can call local external functions.

Returns

0	RXFUNC_OK
10	RXFUNC_DEFINED
20	RXFUNC_NOMEM

RexxDeregisterFunction

RexxDeregisterFunction deregisters an external function.

RexxDeregisterFunction (FuncName)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name to deregister.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG

RexxQueryFunction

RexxQueryFunction queries the existence of a registered external function.

RexxQueryFunction (FuncName)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ external function name to query.

Returns

0	RXFUNC_OK
30	RXFUNC_NOTREG

Remarks

RexxQueryFunction will only return RXFUNC_OK if the requested function is available to the current process. If a function is not available to the current process, RexxQueryFunction search will search the RexxRegisterFunctionDll external function list.

Return Codes

RXFUNC_OK	0	The call to the function completed successfully.
RXFUNC_DEFINED	10	The requested function is already registered.
RXFUNC_NOMEM	20	There is not enough memory to register a new function.
RXFUNC_NOTREG	30	The requested function is not registered.
RXFUNC_MODNOTFND	40	The dynamic link library module could not be found.
RXFUNC_ENTNOTFND	50	The dynamic link library entry point could not be found.

System Exits

The REXX System Exits create user-define REXX interpreter operating environment. Application defined exit handlers process specified REXX interpreter activities.

Applications can create exits for:

- The administration of resources at the beginning and end of interpretation.
- Linkages to external functions and subcommand handlers.
- Special language features. For example, input and output to standard resources.
- Polling for halt and external trace events.

Exit handlers are similar to subcommand handlers and external functions:

- Applications must registers named exit handlers with the REXX interpreter.
- Exit handlers can reside in dynamic link libraries or within an application EXE module.

Writing System Exit Handlers

The following is a sample exit handler definition:

```
LONG Rexx_IO_exit(
     LONG  ExitNumber,    /* code defining the exit function    */
     LONG  Subfunction,   /* code defining the exit subfunction */
     PEXIT ParmBlock);    /* function dependent control block   */
```

Figure 9-11. Sample System Exit Handler Definition

Where:

ExitNumber The major function code defining the type of exit call.

Subfunction The subfunction code defining the exit event for the call.

ParmBlock A pointer to the exit parameter list.

The exit parameter list contains exit specific information. See the exit descriptions below parameter list formats.

Note: Some exit subfunctions do not have parameters. *ParmBlock* for exit subfunctions without parameters.

Exit Return Codes

Exit handlers return an integer value that signals one of three actions:

RXEXIT_HANDLED
The exit handler processed the exit subfunction and updated the subfunction parameter list as required. The REXX interpreter continues with normal processing.

RXEXIT_NOT_HANDLED
The exit handler did not process the exit subfunction. The REXX interpreter processes the subfunction as if the exit handler had not been called.

RXEXIT_RAISE_ERROR
A fatal error occurred in the exit handler. The REXX interpreter raises REXX error 48 ("Failure in system service").

For example, if an application creates an input/output exit handler:

- When the exit handler returns RXEXIT_NOT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter writes the output line to STDOUT.
- When the exit handler returns RXEXIT_HANDLED for an RXSIOSAY subfunction, the REXX interpreter assumes the exit handler has performed all required output. The interpreter will not write the output line to STDOUT.
- When the exit handler returns RXEXIT_RAISE_ERROR for an RXSIOSAY subfunction, the interpreter raise REXX error 48, "Failure in system service."

Exit Parameters

Each exit subfunction has a different parameter list. All RXSTRING exit subfunction parameters are passed as null-terminated RXSTRINGs. It is possible that the RXSTRING value may contain null characters also.

For some exit subfunctions, the exit handler may return an RXSTRING character result in the parameter list. The interpreter provides a default 256-byte for RXSTRING result strings. If the result is longer than 256 bytes, a new RXSTRING can be allocated using DosAllocMem. The REXX interpreter will return the RXSTRING storage for the exit handler.

Identifying Exit Handlers to REXX

System exit handlers must be registered with RexxRegisterExitDll or RexxRegisterExitExe. The system exit handler registration is similar to subcommand handler registration.

The REXX system exits are enabled with the RexxStart function parameter *Exits*. *Exits* is a pointer to an array of RXSYSEXIT structures. Each RXSYSEXIT structure in the array contains a REXX exit code and the address of an ASCIIZ exit handler name. The RXENDLST exit code marks the exit list end.

```
typedef struct {
   PSZ          sysexit_name;      /* name of exit handler         */
   LONG         sysexit_code;      /* system exit function code    */
} RXSYSEXIT;
```

Figure 9-12. RXSYSEXIT Data Structure

The REXX interpreter calls the registered exit handler named in *sysexit_name* for all of the *sysexit_code* subfunctions.

Example

```
WORKAREARECORD  *user_info[2];         /* saved user information    */
RXSYSEXIT exit_list[2];                /* system exit list          */

  user_info[0] = global_workarea;      /* save global work area for */
  user_info[1] = NULL;                 /* re-entrancy               */

  rc = RexxRegisterExitExe("EditInit", /* register exit handler     */
      &Init_exit,                      /* located at this address   */
      user_info);                      /* save global pointer       */

                                       /* set up for RXINI exit     */
  exit_list[0].sysexit_name = "EditInit";
  exit_list[0].sysexit_code = RXINI;
  exit_list[1].sysexit_code = RXENDLST;

  return_code = RexxStart(1,           /* one argument              */
                         argv,         /* argument array            */
                         "CHANGE.ED",  /* REXX procedure name       */
                         NULL,         /* use disk version          */
                         "Editor",     /* default address name      */
                         RXCOMMAND,    /* calling as a subcommand   */
                         exit_list,    /* no exits used             */
                         &rc,          /* converted return code     */
                         &retstr);     /* returned result           */

                                       /* process return value      */
:
}

LONG Init_exit(
     LONG  ExitNumber,     /* code defining the exit function    */
     LONG  Subfunction,    /* code defining the exit subfunction */
     PEXIT ParmBlock)      /* function dependent control block   */
{
  WORKAREARECORD  *user_info[2];       /* saved user information    */
  WORKAREARECORD   global_workarea;    /* application data anchor   */
  USHORT           query_flag;         /* flag for handler query    */


  rc = RexxQueryExit("EditInit",       /* retrieve application work */
      NULL,                            /* area anchor from REXX.    */
      &query_flag,
      user_info);

  global_workarea = user_info[0];      /* set the global anchor     */

  if (global_workarea->rexx_trace)     /* trace at start?           */
                                       /* turn on macro tracing     */
    RexxSetTrace(global_workarea->rexx_pid, global_workarea->rexx_tid);
  return RXEXIT_HANDLED;               /* successfully handled      */
}
```

Figure 9-13. Sample System Exit Usage

System Exit Definitions

The REXX interpreter supports the following system exits:

RXFNC External function call exit

- **RXFNCCAL** Call an external function

RXCMD Subcommand call exit

- **RXCMDHST** Call a subcommand handler

RXMSQ External data queue exit

- **RXMSQPLL** Pull a line from the external data queue.
- **RXMSQPSH** Place a line on the external data queue.
- **RXMSQSIZ** Return number of lines on the external data queue.
- **RXMSQNAM** Set active external data queue name.

RXSIO Standard input and output exit.

- **RXSIOSAY** Write a line to the standard output stream for the SAY instruction.
- **RXSIOTRC** Write a line to the standard error stream for REXX trace or REXX error messages.
- **RXSIOTRD** Read a line from the standard input stream for PULL or PARSE PULL.
- **RXSIODTR** Read a line from the standard input stream for interactive debug.

RXHLT Halt processing exit

- **RXHLTTST** Test for a HALT condition.
- **RXHLTCLR** Clear a HALT condition.

RXTRC External trace exit

- **RXTRCTST** Test for an external trace event.

RXINI Initialization exit

- **RXINIEXT** Allow additional REXX procedure initialization.

RXTER Termination exit

- **RXTEREXT** Process REXX procedure termination.

Each exit subfunction has the following characteristics:

- When REXX calls the exit handler.
- The default action when the exit is not provided or the exit handler does not process the subfunction.
- The subfunction parameter list.
- The service the subfunction provides.
- The state of the variable pool interface during the exit handler call. The variable pool interface is fully enabled for the RXCMD, RXFNC, RXINI, and RXTER exit handler calls. The variable pool interface is enabled for RXSHV_EXIT requests for RXHLT, RXCMD, RXFNC, RXSIO and RXMSQ exit handler calls.

System Exit Details

RXFNC

Process calls to external functions.

RXFNCCAL
Process calls to external functions.

When called: When REXX calls an external subroutine or function.

Default action: Call the external routine using the normal external function search order.

Exit Action: Call the external routine, if possible.

Continuation: If necessary, raise REXX error 40 ("Invalid call to routine"), 43 ("Routine not found"), or 44 ("Function did not return data").

Parameter list:

```
typedef struct {
   struct {
      unsigned rxfferr  : 1;          /* Invalid call to routine.   */
      unsigned rxffnfnd : 1;          /* Function not found.        */
      unsigned rxffsub  : 1;          /* Called as a subroutine if  */
                                      /* TRUE.  Return values are   */
                                      /* optional for subroutines,  */
                                      /* required for functions.    */
   } rxfnc_flags ;

   PUCHAR          rxfnc_name;        /* Pointer to function name.  */
   USHORT          rxfnc_namel;       /* Length of function name.   */
   PUCHAR          rxfnc_que;         /* Current queue name.        */
   USHORT          rxfnc_quel;        /* Length of queue name.      */
   USHORT          rxfnc_argc;        /* Number of args in list.    */
   PRXSTRING       rxfnc_argv;        /* Pointer to argument list.  */
                                      /* List mimics argv list for  */
                                      /* function calls, an array of */
                                      /* RXSTRINGs.                 */
   RXSTRING        rxfnc_retc;        /* Return value.              */
} RXFNCCAL_PARM;
```

The name of the external function is defined by *rxfnc_name* and *rxfnc_namel*. The arguments to the function are in *rxfnc_argc* and *rxfnc_argv*. If the named external function is invoked by the REXX CALL instruction (rather than as a function call), the flag *rxffsub* is TRUE.

The exit handler can set *rxfnc_flags* to indicate the external function call success. If neither *rxfferr* or *rxffnfnd* is TRUE, the exit hander successfully called the external function. The error flags are checked only when the exit handler handles the the request.

The exit handler sets *rxffnfnd* to TRUE when the exit handler could not locate the external function. The interpreter raises REXX error 43, "Routine not found." The exit handler sets *rxfferr* to TRUE when the exit handler located the external function, but the external function returned an error return code. The REXX interpreter raises error 40, "Invalid call to routine.."

The exit handler returns the external function result in the *rxfnc_retc* RXSTRING. The REXX interpreter will raise error 44, "Function did not return data." when the external routine is invoked as a function call and the exit handler does not return a result. When the external routine is by the REXX CALL instruction, the exit handler a result is not required.

Note: The variable pool interface is fully enabled during calls to the RXFNC exit handler.

RXCMD

Process calls to subcommand handlers.

RXCMDHST
Call a named subcommand handler.

When called: When a command is issued in a REXX procedure.

Default action: Call the named subcommand handler specified by the current REXX ADDRESS setting.

Exit Action: Process the call to a named subcommand handler.

Continuation: Raise the ERROR or FAILURE condition when indicated by the parameter list flags,

Parameter list:

```
typedef struct {
   struct {                                /* Condition flags              */
      unsigned rxfcfail : 1;               /* Command failed.  Trap with   */
                                           /* CALL or SIGNAL on FAILURE.   */
      unsigned rxfcerr  : 1;               /* Command ERROR occurred.      */
                                           /* Trap with CALL or SIGNAL on  */
                                           /* ERROR.                       */
   } rxcmd_flags;
   PUCHAR           rxcmd_address;         /* Pointer to address name.     */
   USHORT           rxcmd_addressl;        /* Length of address name.      */
   PUCHAR           rxcmd_dll;             /* dll name for command.        */
   USHORT           rxcmd_dll_len;         /* Length of dll name.  0 ==>   */
                                           /* .EXE file.                   */
   RXSTRING         rxcmd_command;         /* The command string.          */
   RXSTRING         rxcmd_retc;            /* Pointer to return code       */
                                           /* buffer.  User allocated.     */
} RXCMDHST_PARM;
```

The *rxcmd_command* field contains the issued command. *rxcmd_address*, *rxcmd_addressl*, *rxcmd_dll*, and *rxcmd_dll_len* fully define the current ADDRESS setting. *rxcmd_retc* is an RXSTRING for the return code value assigned to REXX special variable RC.

The exit handler can set *rxfcfail* or *rxfcerr* to TRUE to raise an ERROR or FAILURE condition.

Note: The variable pool interface function is fully enabled during calls to the RXCMD exit handlers.

RXMSQ

External data queue exit.

RXMSQPLL
Pull a line from the external data queue.

When called: When a REXX PULL instruction, PARSE PULL instruction, or LINEIN() built-in function reads a line from the external data queue.

Default action: Remove a line from the current REXX data queue.

Exit Action: Return a line from the exit handler provided data queue.

Parameter list:

```
typedef struct {
   RXSTRING          rxmsq_retc;       /* Pointer to dequeued entry   */
                                       /* buffer.  User allocated.    */
} RXMSQPLL_PARM;
```

The exit handler returns the queue line in the *rxmsq_retc* RXSTRING.

RXMSQPSH
Place a line on the external data queue.

When called: Called by the REXX PUSH instruction, QUEUE instruction, or LINEOUT() built-in function to add a line to the data queue.

Default action: Add the line to the current REXX data queue.

Exit Action: Add the line to the exit handler provided data queue.

Parameter list:

```
typedef struct {
   struct {                            /* Operation flag              */
      unsigned rxfmlifo : 1;           /* Stack entry LIFO when TRUE, */
                                       /* FIFO when FALSE.            */
   } rxmsq_flags;
   RXSTRING          rxmsq_value;      /* The entry to be pushed.     */
} RXMSQPSH_PARM;
```

The *rxmsq_value* RXSTRING contains the line added to the queue. It is the responsibility of the exit handler to truncate the string if the exit handler data queue has a maximum length restriction. *rxfmlifo* is the stacking order (LIFO or FIFO).

RXMSQSIZ
Return the number of lines in the external data queue.

When called: When the REXX QUEUED() built-in function requests the size of the external data queue.

Default action: Request the size from the current REXX data queue.

Exit Action: Return the size of the exit handler provided data queue.

Parameter list:

```
typedef struct {
   ULONG              rxmsq_size;        /* Number of Lines in Queue     */
} RXMSQSIZ_PARM;
```

The exit handler returns the number of queue lines in *rxmsq_size*.

RXMSQNAM
Set the name of the active external data queue.

When called: Called by the RXQUEUE("SET", *newname*) built-in function.

Default action: Change the current default queue to *newname*.

Exit Action: Change the default queue name for the exit handler provided data queue.

Parameter list:

```
typedef struct {
   RXSTRING           rxmsq_name;        /* RXSTRING containing          */
                                         /* queue name.                  */
} RXMSQNAM_PARM;
```

rxmsq_name contains the new queue name.

RXSIO

Standard input and output.

RXSIOSAY
Write a line to the standard output stream (STDOUT).

When called: By the SAY instruction to write a line to the standard output stream.

Default action: Write to the OS/2 standard output stream.

Exit Action: Write the line to the exit handler provided output stream.

Parameter list:

```
typedef struct {
   RXSTRING          rxsio_string;     /* String to display.         */
} RXSIOSAY_PARM;
```

The output line is contained in *rxsio_string*. The output line may be any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRC
Write trace and error message output to the standard error stream.

When called: To output lines of trace output and REXX error messages.

Default action: Write the line to the OS/2 standard error stream (STDERR).

Exit Action: Write tine to the exit handler provided error output stream.

Parameter list:

```
typedef struct {
  RXSTRING           rxsio_string;     /* Trace line to display.     */
} RXSIOTRC_PARM;
```

The output line is contained in *rxsio_string*. The output line may be of any length. It is the responsibility of the exit handler to truncate or split the line if necessary.

RXSIOTRD
Read from standard input stream.

When called: To read from the standard input stream for the REXX PULL and PARSE PULL instructions.

Default action: Read a line from the OS/2 standard input stream (STDIN).

Exit Action: Return a line from the exit handler provided standard input stream.

Parameter list:

```
typedef struct {
   RXSTRING          rxsiotrd_retc;    /* RXSTRING for output.       */
} RXSIOTRD_PARM;
```

The input stream line is returned in the *rxsiotrd_retc* RXSTRING.

RXSIODTR

Interactive debug input.

When called: To read from the debug input stream for interactive debug prompts.

Default action: Read a line from the OS/2 standard input stream (STDIN).

Exit Action: Return a line from the exit handler provided standard debug stream.

Parameters: Debug read from STDIN: stream.

```
typedef struct {
   RXSTRING         rxsiodtr_retc;    /* RXSTRING for output.       */
} RXSIODTR_PARM;
```

The input stream line is returned in the *rxsiodtr_retc* RXSTRING.

Note: The PARSE LINEIN instruction and the LINEIN, LINEOUT, LINES, CHARIN, CHAROUT, and CHARS built-in functions do not call the RXSIO exit handler.

RXHLT

HALT condition processing.

Note: Since the RXHLT exit handler is called after every REXX instruction, this exit will slow REXX program execution. The RexxSetHalt function may be used to halt a REXX program without between-instruction polling.

RXHLTTST
Test HALT indicator.

When called: RXHLTTST is called by the interpreter to poll externally raised HALT conditions. The exit will be called after completion of every REXX instruction.

Default action: The interpreter uses the system facilities for trapping Cntrl-Break signals.

Exit Action: Return the current state of the HALT condition (either TRUE or FALSE).

Continuation: Raise the REXX HALT condition if the exit handler returns TRUE.

Parameter list:

```
typedef struct {
   struct {                            /* Halt flag                  */
      unsigned rxfhhalt : 1;           /* Set if HALT occurred.      */
   } rxhlt_flags;
} RXHLTTST_PARM;
```

If the exit handler sets *rxfhhalt* to TRUE, the HALT condition will be raised in the REXX program.

When the exit handler has set *rxfhhalt* to TRUE, it can also use the RXSHV_EXIT operation of RexxVariablePool to return a string describing the HALT condition reason. The REXX program can retrieve the reason string using the CONDITION("D") built-in function.

RXHLTCLR
Clear HALT condition.

When called: To acknowledge processing of the HALT condition when the interpreter has recognized and raised a HALT condition

Default action: The interpreter resets the Cntrl-Break signal handlers.

Exit Action: Reset exit handler HALT state to FALSE.

Parameters: None.

RXTRC

Test external trace indicator.

Note: Since the RXTST exit is called after every REXX instruction, these exits will slow REXX procedure execution. The RexxSetTrace function may be used to turn on REXX tracing without the between-instruction polling.

RXTRCTST
Test external trace indicator.

When called: RXTRCTST is called by the interpreter to poll for an external trace event. The exit will be called after completion of every REXX instruction.

Default action: None.

Exit Action: Return the current state of external tracing (either TRUE or FALSE).

Continuation: When the exit handler switches from FALSE to TRUE, the REXX interpreter enters REXX interactive debug mode using TRACE ?R level of tracing. When the exit handler switches from TRUE to FALSE, the REXX interpreter will exit interactived debug mode.

Parameter list:

```
struct rxtrc_parm {
  struct {
    unsigned rxftrace : 1;       /* External trace setting       */
         } rxtrc_flags;
}
```

If the exit handler switches *rxftrace* to TRUE, REXX will switch on interactive debug mode. It the exit handler switches *rxftrace* to FALSE, REXX will switch off interactive debug mode.

RXINI

Initialization processing.

RXINIEXT
Initialization exit.

The RXINI exit is called as the last step of REXX program initialization. The exit handler may perform additional initialization. For example:

- Use RexxVariablePool to initialize application specific variables
- Use RexxSetTrace to switch on REXX interactive debug mode.

When called: Before the first instruction of the REXX procedure is interpreted.

Default action: None.

Exit Action: The exit handler may perform additional initialization. For example:

- Use RexxVariablePool to initialize application specific variables
- Use RexxSetTrace to switch on REXX interactive debug mode.

Parameters: None.

Note: The variable pool interface is fully enabled for this exit.

RXTER

Termination processing.

RXTEREXT
Termination exit

the RXINI exit is called as the first step of REXX program termination.

When called: After the last instruction of the REXX procedure has been interpreted.

Default action: None.

Exit Action: The exit handler may perform additional termination activities For example, the exit handler can use RexxVariablePool to retrieve REXX variables values.

Parameters: None.

Note: The variable pool interface is fully enabled for this exit.

System Exit Functions

The system exit functions are similar to the subcommand handler functions. The system exit functions are:

RexxRegisterExitDll

RexxRegisterExitDll registers an exit handler that resides in a dynalink library routine.

RexxRegisterExitDll (ExitName, ModuleName, EntryPoint, UserArea, DropAuth)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* is the DLL file containing the exit handler routine.

EntryPoint *(PSZ) - input*
Address of an ASCIIZ dynalink procedure name. *EntryPoint* is the routine within *ModuleName* that REXX will call as an exit handler.

UserArea *(PUCHAR) - input*
Address of an eight-byte area of user defined information. The eight-bytes addressed by *UserArea* will be saved with the exit handler registration. *UserArea* may be NULL if there is no user information to save. The saved user information can be retrieved with the RexxQueryExit function.

DropAuth *(ULONG) - input*
The drop authority. *DropAuth* identifies the processes that can deregister the exit handler. The possible *DropAuth* values are:

RXEXIT_DROPPABLE
Any process can deregister the subcommand handler with RexxDeregisterSubcom.

RXEXIT_NONDROP
Only a thread within the same process as the thread that registered the handler can deregister the handler with RexxDeregisterExit.

Returns

0	RXEXIT_OK
10	RXEXIT_DUP
1002	RXEXIT_NOEMEM
1003	RXEXIT_BADTYPE

Remarks

EntryPoint may be either a 16-bit or a 32-bit routine. REXX will invoke the exit handler in the correct addressing mode.

RexxRegisterExitExe

RexxRegisterExitExe registers an exit handler that resides within application code.

RexxRegisterExitExe(EnvName, EntryPoint, UserArea)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ exit handler name.

EntryPoint *(PFN) - input*
Address of the exit handler entry point within the application EXE file.

UserArea *(PUCHAR) - input*
Address of an eight-byte area of user defined information. The eight-bytes addressed by *UserArea* will be saved with the exit handler registration. *UserArea* may be NULL if there is no user information to save. The user information can be retrieved with the RexxQueryExit function.

Returns

0	RXEXIT_OK
10	RXEXIT_DUP
30	RXEXIT_NOTREG
1002	RXEXIT_NOEMEM
1003	RXEXIT_BADTYPE

Remarks

If *EnvName* has the same name as a handler registered with RexxRegisterExitDll, RexxRegisterExitExe will return RXEXIT_DUP. This is not an error and the new exit handler has been properly registered.

Example

```
WORKAREARECORD  *user_info[2];       /* saved user information     */

user_info[0] = global_workarea;      /* save global work area for  */
user_info[1] = NULL;                 /* re-entrancy                */

rc = RexxRegisterExitExe("IO_Exit",  /* register editor handler    */
    &Edit_IO_Exit,                   /* located at this address    */
    user_info);                      /* save global pointer        */
```

Figure 9-14. Sample Exit Handler Registration

RexxDeregisterExit

RexxDeregisterExit deregisters an exit handler.

RexxDeregisterExit(EnvName, ModuleName)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
Address of an ASCIIZ dynamic link library name. *ModuleName* restricts the query to an exit handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxDeregisterExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxDeregisterExit does not find a RexxRegisterExitExe handler, it will search the RexxRegisterExitDll exit handler list.

Returns

0	RXEXIT_OK
30	RXEXIT_NOTREG
40	RXEXIT_NOCANDROP
1003	RXEXIT_BADTYPE

Remarks

The handler is removed from the exit handler list.

RexxQueryExit

RexxQueryExit queries an exit handler and retrieves saved user information.

RexxQueryExit(EnvName, ModuleName, Flag, UserWord)

Parameters

EnvName *(PSZ) - input*
Address of an ASCIIZ exit handler name.

ModuleName *(PSZ) - input*
ModuleName restricts the query to an exit handler within the *ModuleName* dynamic link library. When *ModuleName* is NULL, RexxQueryExit searches the RexxRegisterExitExe exit handler list for a handler within the current process. If RexxQueryExit does not find a RexxRegisterExitExe handler, it will search the RexxRegisterExitDll exit handler list.

Flag *(PUSHORT) - output*
Exit handler registration flag. *Flag* is the *EnvName* exit handler registration status. When RexxQueryExit returns RXEXIT_OK, the *EnvName* exit handler is currently registered. When RexxQueryExit returns RXEXIT_NOTREG, the *EnvName* exit handler is not registered.

Flag *(PUSHORT) - output*
Exit handler registration flag. *Flag* indicates if the *EnvName* exit handler is registered. If *Flag* is RXEXIT_OK, the *EnvName* exit handler is not registered. If *Flag* is RXEXIT_NOTREG, the *EnvName* exit handler is registered.

UserWord *(PUCHAR) - output*
Address of an eight-byte area to receive the user information saved with RexxRegisterExitExe or RexxRegisterExitDll. *UserWord* can be NULL if the saved user information is not required.

Returns

0	RXEXIT_OK
30	RXEXIT_NOTREG
1003	RXEXIT_BADTYPE

Example

```
ULONG Edit_IO_Exit(
  PRXSTRING Command,     /* Command string passed from the caller    */
  PUSHORT   Flags,       /* pointer to short for return of flags     */
  PRXSTRING Retstr)      /* pointer to RXSTRING for RC return        */
{
  WORKAREARECORD  *user_info[2];       /* saved user information     */
  WORKAREARECORD   global_workarea;    /* application data anchor    */
  USHORT           query_flag;         /* flag for handler query     */


  rc = RexxQueryExit("IO_Exit",        /* retrieve application work  */
      NULL,                            /* area anchor from REXX.     */
      &query_flag,
      user_info);

  global_workarea = user_info[0];      /* set the global anchor      */
```

Figure 9-15. Sample Exit Handler Query

Variable Pool Interface

Application programs can use the REXX Variable Pool Interface to manipulate the variables of a currently active REXX procedure.

RexxVariablePool Interface Function

REXX procedure variables are accessed using the RexxVariablePool function.

RexxVariablePool

RexxVariablePool accesses variables of a currently active REXX procedure.

RexxVariablePool(RequestBlockList)

Parameters

RequestBlockList *(PSHVBLOCK) - input*
A linked list of shared variable request blocks (SHVBLOCK). Each shared variable request block in the linked list is a separate variable access request.: The SHVBLOCK has the following form:

```
typedef struct shvnode {
    struct shvnode    *shvnext;
    RXSTRING           shvname;
    RXSTRING           shvvalue;
    ULONG              shvnamelen;
    ULONG              shvvaluelen;
    UCHAR              shvcode;
    UCHAR              shvret;
}   SHVBLOCK;
```

Figure 9-16. SHVBLOCK Data Structure

Where:

shvnext The address of the next SHVBLOCK in the request list. *shvnext* is NULL for the last request block.

shvcode The shared variable block request code. The request codes are:

RXSHV_SET
RXSHV_SYSET
Assign a new value to a REXX procedure variable.

RXSHV_FETCH
RXSHV_SYFETCH
Retrieve the value of a REXX procedure variable.

RXSHV_DROPV
RXSHV_SYDRO
Drop (unassign) a REXX procedure variable.

RXSHV_PRIV
Fetch REXX procedure private information. The following information items can be retrieved by name:

PARM
The number of arguments supplied to the REXX procedure. The number will be formatted as a character string.

PARM.n
The Nth argument string to the REXX procedure. If the Nth argument was not supplied to the procedure (either omitted or fewer than N parameters were specified), a null string will be returned.

QUENAME
The current REXX data queue name.

SOURCE
The REXX procedure source string used for the PARSE SOURCE instruction.

VERSION
The REXX interpreter version string used for the PARSE SOURCE instruction.

RXSHV_NEXTV
Fetch next variable. RXSHV_NEXTV traverses the variables in the current generation of REXX variables, excluding variables hidden by PROCEDURE instructions. The variables will not be returned in any specified order.

The REXX interpreter maintains an internal pointer to its list of variables. The variable pointers is reset to the first REXX variable whenever:

1. An external program returns control to the interpreter
2. A set, fetch or drop RexxVariablePool function is used.

RXSHV_NEXTV returns both the name and the value of REXX variables until the end of the variable list is reached. If no REXX variables are left to return, RexxVariablePool will set the RXSHV_LVAR bit in *shvret*.

RXSHV_EXIT
Set a return value for an external function or system exit call. RXSHV_EXIT is only valid from external functions or system exit events which return a string value. A single call is allowed per external call.

shvret Individual shared variable request return code. *shvret* is a 1-byte field of status flags for the individual shared variable request. The *shvret* fields for all request blocks in the list are ORed together to form the RexxVariablePool return code. The individual status conditions are:

RXSHV_OK
The request was processed with out error (all flag bits are FALSE).

RXSHV_NEWV
The named variable was uninitialized at the time of the call.

RXSHV_LVAR
No more variables are available for an RXSHV_NEXTV operation.

RXSHV_TRUNC
A variable value or variable name was truncated because the supplied RXSTRING was too small for the copied value.

RXSHV_BADN
The variable name specified in *shvname* was invalid for the requested operation.

RXSHV_MEMFL
The REXX interpreter was unable to obtain the storage required to complete the request.

RXSHV_BADF
The shared variable request block contains an invalid function code.

shvname An RXSTRING containing a REXX variable name. *shvname* usage varies for the different SHVBLOCK request codes:

RXSHV_SET
RXSHV_SYSET
RXSHV_FETCH
RXSHV_SYFET
RXSHV_DROPV
RXSHV_SYDRO
RXSHV_PRIV
shvname is an RXSTRING pointing to the name of the REXX variable accessed by the shared variable request block.

RXSHV_NEXTV
shvname is an RXSTRING defining an area of storage to receive the name of the next variable. *shvnamelen* is the length of the RXSTRING area. If the variable name is longer than *shvnamelen* characters, the name will be truncated and the RXSHV_TRUNC bit of *shvret* will be set. On return, *shvname.strlength* will contain the length of the variable name; *shvnamelen* will be unchanged.

If *shvname* is an empty RXSTRING (*strptr* is NULL), the REXX interpreter will allocate and return an RXSTRING to hold the variable name. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur. However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory will not be allocated for that request block. The RexxVariablePool caller is responsible for releasing the storage with DosFreeMem.

Note: The RexxVariablePool does not add a terminating null character to the variable name.

RXSHV_EXIT
shvname is unused for the RXSHV_EXIT function.

shvvalue An RXSTRING containing a REXX variable value. *shvvalue* meaning varies for the different SHVBLOCK request codes:

RXSHV_SET
RXSHV_SYSET
shvvalue is the value assigned to the REXX variable in *shvname*. *shvvaluelen* contains the length of the variable value.

RXSHV_EXIT
shvvalue is the value assigned to the exit handler return value. *shvvaluelen* contains the length of the variable value.

RXSHV_FETCH
RXSHV_SYFET
RXSHV_PRIV
RXSHV_NEXT
shvvalue is a buffer the REXX interpreter uses to return a copy of REXX variable *shvname*. *shvvaluelen* contains the length of the value buffer. On return, *shvvalue.strlength* will be set to the length of the returned value and *shvvaluelen* will be unchanged. If the variable value is longer than *shvvaluelen* characters, the value will be truncated and the RXSHV_TRUNC bit of *shvret* will be set. On return, *shvvalue.strlength* will be set to the length of the returned value; *shvvaluelen* will be unchanged.

If *shvvalue* is an empty RXSTRING (*strptr* is NULL), the REXX interpreter will allocate and return an RXSTRING to hold the variable value. If the REXX interpreter allocates the RXSTRING, an RXSHV_TRUNC condition cannot occur. However, RXSHV_MEMFL errors are possible for these operations. If an RXSHV_MEMFL condition occurs, memory will not be allocated for that request block. The RexxVariablePool caller is responsible for releasing the storage with DosFreeMem.

Note: The RexxVariablePool does not add a terminating null character to the variable value.

RXSHV_DROPV
RXSHV_SYDRO
shvvalue is not used.

The REXX interpreter processes each request block in the order provided; RexxVariablePool returns to the caller after the last block is processed or after a severe error (such as an out-of-memory condition).

The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The return codes for all of the individual requests are ORed together to form the composite return code. Individual shared variable request return code are returned in the request shared variable blocks.

RexxVariablePool Return Codes

0 to 127
RexxVariablePool has processed the entire shared variable request block list.

The RexxVariablePool function return code is a composite return code for the entire set of shared variable requests. The low-order 6 bits of the the *shvret* fields for all request blocks are ORed together to form the composite return code. Individual shared variable request status flags are returned in the shared variable request block *shvret* field.

RXSHV_NOAVL
The variable pool interface was not enabled when call was issued.

Interface Types

Three of the Variable Pool Interface functions (set, fetch and drop) have dual interfaces.

Symbolic Interface

The symbolic interface uses normal REXX variable rules when interpreting variables. Variable names are valid REXX symbols (in mixed case if desired) including compound symbols. Compound symbols will be referenced with tail substitution. The functions that use the symbolic interface are RXSHV_SYSET, RXSHV_SYFET, and RXSHV_SYDRO.

Direct Interface

The direct interface uses no substitution or case translation. Simple symbols must be valid REXX variable names. A valid REXX variable name:

- Does not begin with a digit or period
- Contains only uppercase A to Z, the digits 0 - 9, or the characters _, ! or ? before the first period of the name.
- Can contain any characters after the first period of the name.

Compound variables are specified using the derived name of the variable. Any characters (including blanks) may appear after the first period of the name. No additional variable sustitution is used. The direct interface is used by RXSHV_SET, RXSHV_FETCH, and RXSHV_DROP.

RexxVariablePool Restrictions

Only the main thread of an application can access the REXX variable pool. Applications may create and use new threads, but only the original thread that called RexxStart may use RexxVariablePool.

OS/2 EXE modules invoked from a REXX procedure execute in a new process. Because the modules are not using the same process and thread as the REXX procedure, the modules cannot use RexxVariablePool to access REXX variables. RexxVariablePool can be used from subcommand handlers, external functions and exit handlers.

Example

```
/*********************************************************************/
/*                                                                   */
/* SetRexxVariable - Set the value of a Rexx variable                */
/*                                                                   */
/*********************************************************************/

INT SetRexxVariable
  PSZ        name,                       /* Rexx variable to set      */
  PSZ        value)                      /* value to assign           */
{
  SHVBLOCK   block;                      /* variable pool control block*/

  block.shvcode = RXSHV_SYSET;           /* do a symbolic set operation*/
  block.shvret=(UCHAR)0;                 /* clear return code field    */
  block.shvnext=(PSHVBLOCK)0;            /* no next block              */
                                         /* set variable name string   */
  MAKERXSTRING(block.shvname, name, strlen(name));
                                         /* set value string           */
  MAKERXSTRING(block.shvvalue, value, strlen(value));
  block.shvvaluelen=strlen(value);       /* set value length           */
  return RexxVariablePool(&block);       /* set the variable           */
}
```

Figure 9-17. Sample Call to RexxVariablePool

Halt and Trace Functions

The halt and trace functions raise a REXX HALT condition or change the REXX interactive debug mode while a REXX procedure is running. These interfaces may be preferred over the RXHLT and RXTRC system exits. The system exits require an additional call to an exit routine after each REXX instruction completes. This may cause a noticable performance degradation. The Halt and Trace functions are a single request to change the halt or trace state, and do not degrade the REXX procedure performance.

Halt and Trace Functions

RexxSetHalt

RexxSetHalt raises a HALT condition in a running REXX program.

```
RexxSetHalt(ProcessId, ThreadId)
```

Parameters

ProcessId *(PID) - input*
The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

Returns

0	RXARI_OK
1	RXARI_PID_TID_NOT_FOUND
2	RXARI_PROCESSING_ERROR

Note: This call will not be processed if the target REXX program is executing with the RXHLT exit enabled.

RexxSetTrace

RexxSetTrace turns on interactive debug mode for a REXX procedure.

RexxSetTrace(ProcessId, ThreadId)

Parameters

ProcessId *(PID) - input*
The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

A RexxSetTrace call will not be processed if the REXX procedure is using the RXTRC exit.

RexxResetTrace

RexxResetTrace turns off interactive debug mode for a REXX procedure.

RexxResetTrace(ProcessId,ThreadId)

Parameters

ProcessId *(PID) - input*
The process ID of the target REXX procedure. *ProcessId* is the application process that called the RexxStart function.

ThreadId *(TID) - input*
The thread ID of the target REXX procedure. The thread ID of the target REXX procedure. *ThreadId* is the application thread that called the RexxStart function.

A RexxResetTrace call will not be processed if the REXX procedure is using the RXTRC exit.

Returns

0	RXARIOK
1	RXARI_PID_TID_NOT_FOUND
2	RXARI_PROCESSING_ERROR

Notes:

1. A RexxResetTrace call will not be processed if the REXX procedure is using the RXTRC exit.
2. Interactive debug will not be turned off unless interactive debug mode was originally started with RexxSetTrace.

Macrospace Interface

The macrospace can improve the performance of REXX procedures by maintaining REXX procecure images in memory for immediate load and execution. This is useful for frequently used procedures and functions such as editor macros.

Programs registered in the REXX macrospace are available to all processes. They may be executed using the RexxStart function or called as functions or subroutines from other REXX procedures.

Procedures in the macrospace are called the same way other REXX external functions are called. However, the macrospace REXX procedures may be placed at the front or at the very end of the external function search order.

REXX procedures in the macrospace can be saved to a disk file. A saved macrospace file can be reloaded with a single call to RexxLoadMacroSpace. An application, such as an editor, can create its own library of frequently-used functions and load the entire library into memory for fast access. Multiple macrospace libraries may be created and loaded.

Search Order

When RexxAddMacro loads a REXX procecure into the macrospace, the position in the external function search order is specified. The REXX procedure may be placed before all other forms of external function or after all other external functions.

RXMACRO_SEARCH_BEFORE A function registered with RXMACRO_SEARCH_BEFORE will be located by the REXX interpreter before any registered functions or external REXX files.

SEARCH_AFTER Function Registration A function registered with RXMACRO_SEARCH_AFTER will be located by the REXX interpreter after any registered functions or external REXX files.

Storage of Macrospace Libraries

Note: The REXX macrospace is placed in OS/2 shared memory. The size of the macrospace is only limited by the amount of memory and swap space available to the system. However, as the macrospace grows, it limits the memory available to other processes in the system. Allowing the macrospace to grow too large may degrade overall system performance due to increased system swap file access. It is recommended that only the most frequently used functions be placed in the macrospace.

Macrospace Interface Functions

The functions to manpulate macrospaces are:

RexxAddMacro

RexxAddMacro loads a REXX procedure into the macrospace.

RexxAddMacro (FuncName, SourceFile, Position)

Parameters

FuncName *(PSZ) - input*
: Address of the ASCIIZ function name. REXX procedures in the macrospace are called using the assigned function name.

SourceFile *(PSZ) - input*
Address of the ASCIIZ file specification for the REXX procedure source file. When a file extension is not supplied, .CMD is used. When the full path is not specified, the current directory and OS/2 path is searched.

Position *(ULONG) - input*
Position in the REXX external function search order. Possible values are:

RXMACRO_SEARCH_BEFORE
The function will be located by the REXX interpreter before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
The function will be located by the REXX interpreter after any registered functions or external REXX files.

Returns

0	RXMACRO_OK
1	RXMACRO_NO_STORAGE
7	RXMACRO_SOURCE_NOT_FOUND
8	RXMACRO_INVALID_POSITION

RexxDropMacro

RexxDropMacro removes a REXX procedure from the macrospace.

RexxDropMacro (FuncName)

Parameters

FuncName *(PSZ) - input*
Address of the ASCIIZ function name.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND

RexxClearMacroSpace

RexxClearMacroSpace removes all loaded REXX procedures from the macrospace.

```
RexxClearMacroSpace()
```

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND

Remarks

RexxClearMacroSpace should be used with care. This function will remove all functions from the macrospace, including functions loaded by other processes.

RexxSaveMacroSpace

RexxSaveMacroSpace saves all or part of the macrospace REXX procedures to a disk file.

RexxSaveMacroSpace (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(ULONG) - input*
Number of REXX procedures to save.

FuncNames *(PSZ *) - input*
Address of a list of ASCIIZ function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
Address of the ASCIIZ macrospace file name. If *MacroLibFile* already exists, it is replaced with the new file.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND
3	RXMACRO_EXTENSION_REQUIRED
5	RXMACRO_FILE_ERROR

Remarks

When *FuncCount* is zero or *FuncNames* is NULL, RexxSaveMacroSpace saves all functions in the macrospace.: Saved macrospace files may only be used with the same interpreter version that created the images. If RexxLoadMacroSpace is called to load the procedures, and the release level or service level is incorrect, RexxLoadMacroSpace will fail. If RexxLoadMacroSpace fails, the REXX procedures must be reloaded individually from the original source programs.

RexxLoadMacroSpace

RexxLoadMacroSpace loads all or part of the REXX procedures from a saved macrospace file.

RexxLoadMacroSpace (FuncCount, FuncNames, MacroLibFile)

Parameters

FuncCount *(ULONG) - input*
Number of REXX procedures to load from the saved macrospace.

FuncNames *(PSZ *) - input*
Address of a list of ASCIIZ REXX function names. *FuncCount* gives the size of the function list.

MacroLibFile *(PSZ) - input*
Address of the ASCIIZ saved macrospace file name.

Returns

0	RXMACRO_OK
1	RXMACRO_NO_STORAGE
2	RXMACRO_NOT_FOUND
4	RXMACRO_ALREADY_EXISTS
5	RXMACRO_FILE_ERROR
6	RXMACRO_SIGNATURE_ERROR

Remarks

When *FuncCount* is zero or *FuncNames* is NULL, RexxLoadMacroSpace loads all REXX procedures from the saved file.: If a RexxLoadMacroSpace call would replace an existing macrospace REXX procedure, the entire load request is discarded and the macrospace remains unchanged.

RexxQueryMacro

RexxQueryMacro searches the macrospace for a specified function.

RexxQueryMacro (FuncName, Position)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ function name.

Position *(PUSHORT) - output*
Address of an unsigned short integer flag. If the function is loaded in the macrospace, *Position* is set to the current function search-order position.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND

RexxReorderMacro

RexxReorderMacro changes the search order position of a loaded macrospace function.

RexxReorderMacro (FuncName, Position)

Parameters

FuncName *(PSZ) - input*
Address of an ASCIIZ macrospace function name.

Position *(ULONG) - input*

New search-order position of the macrospace function. Possible values are:

RXMACRO_SEARCH_BEFORE
The function will be located by the REXX interpreter before any registered functions or external REXX files.

RXMACRO_SEARCH_AFTER
The function will be located by the REXX interpreter after any registered functions or external REXX files.

Returns

0	RXMACRO_OK
2	RXMACRO_NOT_FOUND
8	RXMACRO_INVALID_POSITION

Return Codes

The follow return codes may be returned from the macrospace functions. These values signify the causes for a failure, in these functions.

RXMACRO_OK	0	The call to the function completed successfully
RXMACRO_NO_STORAGE	1	There was not enough memory to complete the requested function
RXMACRO_NOT_FOUND	2	The requested function was not found in the macrospace
RXMACRO_EXTENSION_REQUIRED	3	An extension is required for the macrospace file name.
RXMACRO_ALREADY_EXISTS	4	Duplicate functions cannot be loaded from a macrospace file
RXMACRO_FILE_ERROR	5	An error occurred accessing a macrospace file
RXMACRO_SIGNATURE_ERROR	6	A macrospace save file does not contain valid function images
RXMACRO_SOURCE_NOT_FOUND	7	The requested file was not found
RXMACRO_INVALID_POSITION	8	An invalid search-order position request flag was used

Example

```
                                        /* first load entire package  */
RexxLoadMacroSpace(0, NULL, "EDITOR.MAC");

for (i = 0; i < MACRO_COUNT; i++) {  /* verify each macro           */
                                        /* if not there               */
  if (RexxQueryMacro(macroffli", &position))
                                        /* add to list                */
    RexxAddMacro(macroffli", macro_filesffli",
        RXMACRO_SEARCH_BEFORE);
}
                                        /* rebuild the macrospace     */
RexxSaveMacroSpace(0, NULL, "EDITOR.MAC");

:

                                        /* build the argument string  */
MAKERXSTRING(argv[0], macro_argument,
    strlen(macro_argument));
                                        /* set up default return      */
MAKERXSTRING(retstr, return_buffer, sizeof(return_buffer));
                                        /* set up for macrospace call */
MAKERXSTRING(macrospace[0],NULL, 0);
MAKERXSTRING(macrospace[1],NULL, 0);

return_code = RexxStart(1,              /* one argument               */
                        argv,           /* argument array             */
                        macro[pos],     /* REXX procedure name        */
                        macrospace,     /* use macrospace version     */
                        "Editor",       /* default address name       */
                        RXCOMMAND,      /* calling as a subcommand    */
                        NULL,           /* no exits used              */
                        &rc,            /* converted return code      */
                        &retstr);       /* returned result            */
```

Figure 9-18. Sample Macrospace Usage

Chapter 10. Debugging Aids

In addition to the TRACE instruction described on page 3-37, there are the following debugging aids.

Interactive Debugging of Programs

The debug facility permits interactively controlled execution of a program. Adding the prefix character ? to the TRACE instruction or the TRACE function (for example, `TRACE ?I` or `TRACE(?I)`) turns on interactive debug and indicates to the user that interactive debug is active. Further TRACE instructions in the program are ignored, and the language processor pauses after nearly all instructions that are traced at the console (see the following for the exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** makes the language processor continue execution until the next pause for debug input. Repeatedly entering a null line, therefore, steps from pause point to pause point. For `TRACE ?A`, for example, this is equivalent to single-stepping through the program.

2. **Entering an equal sign (=)** with no blanks makes the language processor re-execute the clause last traced. For example, if an IF clause is about to take the wrong branch, you can change the value of the variables on which it depends, and then re-execute it.

   Once the clause has been re-executed, the language processor pauses again.

3. **Anything else entered** is treated as a *line* of one or more clauses, and processed immediately (that is, as though DO; *line*; END; had been inserted in the program). The same rules apply as in the INTERPRET instruction (for example, DO-END constructs must be complete). If an instruction has a syntax error in it, a standard message is displayed and you are prompted for input again. Similarly all the other SIGNAL conditions are disabled while the string is processed to prevent unintentional transfer of control.

   During execution of the string, no tracing takes place, except that nonzero return codes from host commands are displayed. Host commands are always executed, but the variable RC is not set. Once the string has been processed, the language processor pauses again for further debug input.

Interactive debug is turned off:

- If a TRACE instruction uses the ? prefix while interactive debug is in effect

  or

- At any time, if `TRACE O` or `TRACE` with no options is entered.

The numeric form of the TRACE instruction may be used to allow sections of the program to be executed without pause for debug input. `TRACE n` (that is, positive result) allows execution to continue, skipping the next `n` pauses (when interactive debug is or becomes active). `TRACE -n` (that is, negative result) allows execution to continue without pause and with tracing inhibited for `n` clauses that would otherwise be traced. The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using `TRACE ?R` to trace Results) and then enter a subroutine in which you have

no interest, you can enter TRACE O to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Similarly, if you are interested only in a subroutine, you can put a TRACE ?R instruction at its start. Having traced the routine, the original status of tracing is restored and hence (if tracing was off on entry to the subroutine) tracing (and interactive debug) is turned off until the next entry to the subroutine.

Since any instructions may be executed in interactive debug you have considerable control over execution.

The following are some examples:

```
Say expr      /* displays the result of evaluating the        */
              /* expression.                                  */

name=expr     /* alters the value of a variable.              */

Trace O       /* (or Trace with no options) turns off         */
              /* interactive debug and all tracing.           */

Trace ?A      /* turns off interactive debug but continue     */
              /* tracing all clauses.                         */

exit          /* terminates execution of the program.         */

do i=1 to 10; say stem.i; end
              /* displays ten elements of the array stem.     */
```

Exceptions: Some clauses cannot safely be re-executed, and therefore the language processor does not pause after them, even if they are traced. These are:

- Any repetitive DO clause, on the second or subsequent time around the loop.
- All END clauses (not a useful place to pause in any case).
- All THEN, ELSE, OTHERWISE, or null clauses.
- All RETURN clauses, except when returning from an internal function or subroutine call.
- All EXIT clauses.
- All SIGNAL and CALL clauses (the language processor pauses after the target label has been traced).
- Any clause that causes a syntax error. (These may be trapped by SIGNAL ON SYNTAX, but cannot be re-executed.)

RXTRACE Variable

When the interpreter starts the interpretation of a REXX procedure it will check the setting of the special environment variable, *RXTRACE*. If *RXTRACE* has been set to ON (not case sensitive) then the interpreter will start in interactive debug mode (as if the REXX instruction TRACE '?R' had been the first interpretable instruction). All other settings of *RXTRACE* will be ignored. *RXTRACE* will only be checked when starting a new REXX procedure.

Use the OS/2 SET command to set or query OS/2 environment variables.

Chapter 11. Reserved Keywords and Special Variables

Keywords may be used as ordinary symbols in many situations where there is no ambiguity. The precise rules are given in this chapter.

There are three special variables: RC, RESULT, and SIGL.

Reserved Keywords

The free syntax of REXX implies that some symbols are reserved for use by the language processor in certain contexts.

Within particular instructions, some symbols may be reserved to separate the parts of the instruction. These symbols are referred to as keywords. Examples of REXX keywords are the WHILE keyword in a DO instruction, and the THEN keyword, which acts as a clause terminator in this case, following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords; the symbols may be freely used elsewhere in clauses without being understood as keywords.

Be careful of host commands or subcommands with the same name as REXX keywords (for example, the OS/2 command CALL). This can create problems for any programmer whose REXX programs might be used for some time and in circumstances outside his or her control, and who wishes to make the program absolutely *watertight.*

In this case, a REXX program may be written with at least the first words in command lines enclosed in quotes.

The following is an example:

```
'DELETE' Fn'.'Ext
```

This also has an advantage in that it is more efficient; and with this style, the SIGNAL ON NOVALUE condition may be used to check the integrity of an exec.

An alternative strategy is to precede such command strings with two adjacent quotes. This concatenates the null string onto the front.

The following is an example:

```
''Erase Fn'.'Ext
```

A third option is to enclose the entire expression, or the first symbol, in parentheses.

The following is an example:

```
(Erase Fn'.'Ext)
```

The choice of strategy, if it is to be done at all, is a personal one by the programmer. It is not imposed by the REXX language.

Special Variables

There are three special variables that may be set automatically by the language processor:

RC Is set to the return code from any executed host command (or subcommand). Following the SYNTAX, ERROR, and FAILURE SIGNAL events, RC is set to the code appropriate to the event; the syntax error number (see Appendix A for error messages) or the command return code. RC is unchanged following a HALT, NOTREADY, or NOVALUE event.

Note: Host commands that are executed manually from debug mode do not cause the value of RC to change.

RESULT Is set by a RETURN instruction in a subroutine that has been called if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL Contains the line number of the clause currently running when the last transfer of control to a label took place. (This could be caused by a SIGNAL event, a CALL instruction, an internal function invocation, or a trapped error condition.)

None of these variables has an initial value. They may be altered by the user, just like any other variable. They also may be accessed by way of the variable pool interface (see "Variable Pool Interface" on page 9-45). The PROCEDURE and DROP instructions also affect these variables in the usual way.

Certain other information is always available to a REXX program. This includes the name by which the program was invoked and the source of the program (which is available using the PARSE SOURCE instruction, see page 3-26). The latter consists of the string "OS/2," followed by the call type and then the full path specification of the file being executed.

In addition, PARSE VERSION (see page 3-26) makes available the version and date of the language processor code that is running. The built-in functions TRACE and ADDRESS return the current trace setting and environment name respectively.

Finally, the current settings of the NUMERIC function can be obtained using the DIGITS, FORM, and FUZZ built-in functions.

Chapter 12. Useful OS/2 Commands

CALL Command

The OS/2 CALL command should not be confused with the REXX CALL *instruction* (see page 3-6). The REXX CALL instruction calls REXX internal or external subroutines. The OS/2 CALL command:

- When used in an OS/2 batch file, invokes a REXX program or OS/2 batch file. OS/2 batch files use REXX programs and OS/2 batch fles as commands by using the CALL command. When the called command completes, the calling batch file continues execution.

  Using a REXX program or batch file by name only (without CALL) *transfers* control to the named routine. The calling batch file is terminated.

- When used in a REXX program, the OS/2 CALL command calls a REXX program or OS/2 batch file as an OS/2 command. The call creates a new invocation of the CMD.EXE command shell. When the command files is a REXX program, the REXX interpreter is also called again. When the called program completes, the original REXX program resumes execution. To call a REXX program as an OS/2 command, the CALL command must be a quoted string:

  ```
  "call progl"
  ```

  If the CALL is not a quoted string

  ```
  call progl
  ```

  the REXX program is called as a subroutine.

Notes:

1. A REXX program cannot be called as an OS/2 command with the CALL command (for example, "progl" alone). OS/2 will give error ********** give chaining message.
2. A CMD file cannot call itself with the OS/2 CALL command. However, a REXX program can call itself with the REXX CALL instruction.

Other OS/2 Commands

COPY Copies files.

DELETE Deletes files

DIR Displays disk directories.

ERASE Erases files.

MODE Controls input and output device characteristics.

PATH Defines or displays the search path for commands and REXX programs. See also "Search Order" on page 4-2.

SET Displays or changes OS/2 environment variables. See also "VALUE" on page 4-41.

Subcommand Handler Services

See "Subcommand Interfaces" on page 9-9 for a complete subcommand handler description.

The RXSUBCOM Command

The RXSUBCOM command registers, drops, and queries REXX subcommand handlers. A REXX procedure or OS/2 batch file can use RXSUBCOM register dynamic link library subcommand handlers. Once the subcommand handler is registered, a REXX program can send commands to the subcommand handler with the REXX ADDRESS instruction. For example, REXX Dialog Manager programs use RXSUBCOM to register the ISPCIR subcommand handler.

```
'RXSUBCOM REGISTER ISPCIR ISPCIR ISPCIR'
Address ispcir
```

See "ADDRESS" on page 3-2 for details of the ADDRESS instruction.

RXSUBCOM REGISTER

RXSUBCOM REGISTER registers a dynamic link library subcommand handler. This command makes a command environment available to REXX.

RXSUBCOM REGISTER *envname dllname procname*

Options

envname
The subcommand handler name. The REXX ADDRESS instruction uses *envname* to send commands to the subcommand handler.

dllname
Name of the dynamic link library file containing the subcommand handler routine.

procname
Name of the dynamic link library procedure within *dllname* that REXX will call as a subcommand handler.

Returns

0	The command environment has been registered.
10	A duplicate registration has occurred. An *envname* subcommand handler in a different dynamic link library has already been registered. Both the new subcommand handler and the existing subcommand handler can be used.
30	The registration has failed. Subcommand handler *envname* in library *dllname* is already registered.
1002	RXSUBCOM registers was unable to obtain the memory necessary to register the subcommand handler.
-1	A parameter is missing or incorrectly specified.

RXSUBCOM DROP

RXSUBCOM DROP deregisters a subcommand handler.

RXSUBCOM DROP *envname [dllname]*

Options

envname
The name of the subcommand handler.

[dllname]
Name of the dynamic link library file containing the subcommand handler routine.

Returns

0	The subcommand handler was successfully deregistered.
30	The subcommand handler does not exist.
40	The environment was registered by a different process as RXSUBCOM_NONDROP.
-1	A parameter is missing or specified incorrectly.

RXSUBCOM QUERY

This command checks the existance of a subcommand handler. The return is the query result.

RXSUBCOM QUERY *envname [dllname]*

Options

envname
The name of the subcommand handler.

[dllname]
Name of the dynamic link library file containing the subcommand handler routine.

Returns

0	The subcommand handler is registered.
30	The subcommand handler is not registered.
-1	A parameter is missing or specified incorrectly.

RXSUBCOM LOAD

RXSUBCOM LOAD loads a subcommand handler dynamic link library.

RXSUBCOM LOAD *envname [dllname]*

Options

envname
The name of the subcommand handler.

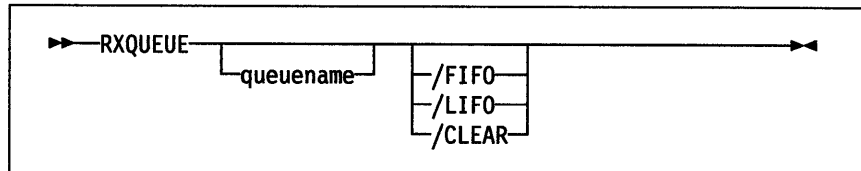
[dllname]
Name of the dynamic link library file containing the subcommand handler routine.

Returns

0	The dynamic link library was located and loaded successfully.
50	The dynamic link library was not located or the dynamic link library could not be loaded.
-1	A parameter is missing or incorrectly specified.

Queue Services (Filters)

RXQUEUE filter



The RXQUEUE filter normally operates on the default queue named SESSION. However, if an environment variable named ″RXQUEUE″ exists, the RXQUEUE value will be used for the queue name.

For a full description of REXX queue services for applications programming, see "Queue Interface" on page 8-4.

Parameters

[queuename]/LIFO
Stacks items from STDIN last in, first out (LIFO) on a REXX queue.

[queuename]/FIFO
Queues items from STDIN first in, first out (FIFO) on a REXX queue.

[queuename]/CLEAR
Removes all lines from a REXX queue.

Remarks
: RXQUEUE takes output lines from another programs and placess them on a REXX queue. A REXX procedure can use RXQUEUE to capture OS/2 command and program output for processing. RXQUEUE can direct output to any REXX queue, either either FIFO (first in, first out) or LIFO (last in, first out).

RXQUEUE uses the OS/2 environment variable RXQUEUE for the default queue name. When RXQUEUE does not have a value, RXQUEUE uses SESSION for the queue name.

This example obtains the OS/2 version number with RXQUEUE:

```
/* Sample program to show simple use of RXQUEUE */
/* Find out the OS/2 version number, using the  */
/* VER command.  VER produces two lines of      */
/* output; one blank line, and one line with the*/
/* format "The OS/2 Version is n.nn"            */

'VER |RXQUEUE'          /* Put the data on the Queue      */
pull .                  /* Get and discard the blank line */
Pull . . . . . number   /* Get the data, keeping only the number */

Say 'We are running on OS/2 Version' number
```

This example processes output from the OS/2 DIR command:

```
/* Sample program to show how to use the RXQUEUE filter */
/* This program filters the output from a DIR command,  */
/* ignoring small files.  It displays a list of the     */
/* large files, and the total of the sizes of the large */
/* files.                                                */

size_limit = 10000            /* The dividing line      */
                              /* between large and small*/
size_total = 0                /* Sum of large file sizes*/
NUMERIC DIGITS 12             /* Set up to handle very  */
                              /* numbers                */

/* Create a new queue so that this program cannot       */
/* interfere with data placed on the queue by another   */
/* program.                                              */

queue_name = rxqueue('Create')
Call rxqueue 'Set', queue_name

'DIR /N | RXQUEUE' queue_name

/* DIR output starts with five header lines              */
Do 5
   Pull .                     /* discard header line    */
End

/* Now all the lines are file or directory lines,        */
/* except for one at the end.                            */

Do queued() - 1               /* loop for lines we want */
   Parse Pull . . size . name ./* get one name and size */
   /* If the size field says "<DIR>", we ignore this    */
   /* line.                                              */
   If size <> '<DIR>' Then
      /* Now check size, and display                    */
      If size > size_limit Then Do
         Say format(size,12) name
         size_total = size_total + size
      End
End

Say 'The total size of those files is' size_total

/* Now we are done with the queue.  We delete it, which */
/* discards the line remaining in it.                   */

Call rxqueue 'DELETE', queue_name
```

Appendix A. Error Numbers and Messages

The error numbers produced by syntax errors during processing of REXX programs are all in the range 3 to 49 (and this is the value placed in the variable RC when SIGNAL ON SYNTAX event is trapped).

Two of the error messages can be generated by the external interfaces to the language processor either before the language processor gains control or after control has left the language processor. Therefore these errors cannot be trapped by SIGNAL ON SYNTAX. The error numbers involved are 3 and 5 (if the initial requirements for storage could not be met). Similarly, error 4 can be trapped only by SIGNAL ON HALT or CALL ON HALT.

Two of the error messages are generated when the program is being tokenized. Because the program has not yet started executing, these errors cannot be trapped by SIGNAL ON SYNTAX unless they occur during execution of an INTERPRET instruction. The errors involved are 6 and 30.

Error 3 Program is unreadable

Explanation: The REXX program could not be read from the disk.

Error 4 Program interrupted

Explanation: The system interrupted execution of your REXX program.

Error 5 Machine resources exhausted

Explanation: While attempting to process a program, the language processor was unable to get the space needed for its work areas and variables. This may have occurred because the program that invoked the language processor has already used up most of the available storage itself.

Error 6 Unmatched "/*" or quote

Explanation: The language processor reached the end of the file (or the end of data in an INTERPRET instruction) without finding the ending "*/" for a comment or quote for a literal string.

Error 7 WHEN or OTHERWISE expected

Explanation: The language processor expected a series of WHEN expressions and an OTHERWISE within a SELECT statement. This message is issued when any other instruction is found or if all WHEN expressions are found to be false and an OTHERWISE is not present. This error is often caused by forgetting the DO and END instructions around the list of instructions following a WHEN. For example:

```
WRONG                      RIGHT

Select                     Select
  When a=b then              When a=b then DO
    Say 'A equals B'           Say 'A equals B'
    exit                       exit
  Otherwise nop                end
  end                        Otherwise nop
                             end
```

Error 8 Unexpected THEN or ELSE

Explanation: The language processor has found a THEN or an ELSE clause that does not match a corresponding IF clause. This situation is often caused by forgetting to put an END or DO END instruction in the THEN part of a complex IF THEN ELSE construction. For example:

```
WRONG                      RIGHT

If a=b then do;            If a=b then do;
  Say EQUALS                 Say EQUALS
  exit                       exit
 else                        end
  Say NOT EQUALS            else
                             Say NOT EQUALS
```

Error 9 Unexpected WHEN or OTHERWISE

Explanation: The language processor has found a WHEN or OTHERWISE instruction outside of a SELECT construction. You may have accidentally enclosed the instruction in a DO END construction by leaving off an END instruction, or you may have tried to branch to it with a SIGNAL statement (which cannot work because the SELECT construction is then terminated).

Error 10 Unexpected or unmatched END

Explanation: The language processor has found more END instructions in your program than DO or SELECT instructions, or the END instructions were placed so that they did not match the DO or SELECT instructions.

This message can be caused if you try to SIGNAL into the middle of a loop. In this case, the END instruction will be unexpected because the previous DO instruction will not have been executed. Remember also, that SIGNAL terminates any current loops, so it can not be used to jump from one place inside a loop to another.

This message can also be caused if you place an END instruction immediately after a THEN or ELSE construction. It may be helpful to use TRACE Scan to show the structure of the program and make it more obvious where the error is. Putting the name of the control variable on END instructions that close repetitive loops can also help locate this kind of error.

Error 11 Control stack full

Explanation: This message is issued if you exceed the limit on levels of nesting of control structures (DO-END, IF-THEN-ELSE, and so on).

This message could be caused by a looping INTERPRET instruction. For example:

```
line='INTERPRET line'
INTERPRET line
```

These lines would loop until they exceeded the nesting level limit and this message would be issued. Similarly, a recursive subroutine that does not terminate correctly could loop until it causes this message.

Error 13 Invalid character in program

Explanation: The language processor found an invalid character outside of a literal (quoted) string. Valid characters are:

```
A through Z a through z 0 through 9
(Alphamerics)

! _ ? .  (Name Characters)

& * ( ) - + = ¬ ' " ; : < , > / \ ¦
(Special Characters)
```

Error 14 Incomplete DO/SELECT/IF

Explanation: The language processor has reached the end of the file (or end of data for an INTERPRET instruction) and has found that there is a DO or SELECT instruction without a matching END instruction, or an IF clause that is not followed by a THEN clause. Putting the name of the control variable on END instructions that close repetitive loops can help locate this kind of error.

Error 15 Invalid hexadecimal or binary string

Explanation: For the language processor, hexadecimal or binary constants cannot have leading or trailing blanks and can have imbedded blanks only at byte boundaries. The following are all valid hexadecimal constants:

```
'13'x
'A3C2 1c34'x
'1de8'x
```

These are all valid binary constants:

```
'1011'b
'110 1101'b
'101101 11010011'b
```

You may have mistyped one of the digits, for example, typing a letter o instead of a 0. This message can also be caused if you follow a string by the 1-character symbol X or B (as the name of a variable *X* or *B*) when the string is not intended to be taken as a hexadecimal or binary specification. In this case, use the explicit concatenation operator (||) to concatenate the string to the value of the symbol.

Error 16 Label not found

Explanation: The language processor could not find the label specified by a SIGNAL instruction or a label matching an enabled condition when the corresponding (trapped) event occurred. You may have mistyped the label or forgotten to include it.

Error 17 Unexpected PROCEDURE

Explanation: The language processor encountered a PROCEDURE instruction in an invalid position. This could occur because no internal routines are active or because the PROCEDURE instruction was not the first instruction executed after the CALL instruction or function invocation. This error can be caused by *dropping through* to an internal routine, rather than invoking it with a CALL instruction or a function call.

Error 18 **THEN expected**

Explanation: Each REXX IF and WHEN clause must be followed by a THEN clause. Another clause was found before a THEN clause was found.

Error 19 **String or symbol expected**

Explanation: The language processor expected a symbol or string following the CALL or SIGNAL keywords, but none was found. You may have omitted the string or symbol, or you may have inserted a special character (such as a parenthesis).

Error 20 **Symbol expected**

Explanation: The language processor expected a symbol following the CALL ON, END, ITERATE, LEAVE, NUMERIC, PARSE, PROCEDURE, or SIGNAL ON keywords or expected a list of symbols following the DROP or PROCEDURE (with EXPOSE option) keywords. Either there was no symbol when one was required or some other characters were found.

Error 21 **Invalid data on end of clause**

Explanation: You have followed a clause, such as SELECT or NOP, by some data other than a comment.

Error 23 **Invalid character string.**

Explanation: A data string (that is, the result of an expression) contains character codes that are not valid in the interpreter. This might be because some characters are *impossible* or because the character set is extended in some way and a given character combination is not allowed.

Error 24 **Invalid TRACE request**

Explanation: The language processor issues this message when the action specified on a TRACE instruction or the argument to the built-in function, did not start with an A, C, E, F, I, L, N, O, or R.

Error 25 **Invalid subkeyword found**

Explanation: The language processor expected a particular subkeyword at this position in an instruction but something else was found. For example, the NUMERIC instruction must be followed by the DIGITS, FUZZ, or FORM subkeyword. If NUMERIC is followed by anything else, this message will be issued.

Error 26 **Invalid whole number**

Explanation: The language processor found an expression in the NUMERIC instruction, a parsing positional pattern, or the right hand term of the exponentiation (**) operator that did not evaluate to a whole number or was greater than the limit, for these uses, of 999 999 999. This error condition is also raised when the DO repetitor is not a positive whole number or when an integer-divide or remainder operation does not result in a whole number.

Error 27 **Invalid DO syntax**

Explanation: The language processor found a syntax error in the DO instruction. You might have used BY, TO, or FOR phrases twice, or used BY, TO, or FOR instruction when you did not specify a control variable.

Error 28 **Invalid LEAVE or ITERATE**

Explanation: The language processor encountered an invalid LEAVE or ITERATE instruction. The instruction was invalid because:

- No loop was active

  or

- The name specified on the instruction did not match the control variable of any active loop.

Note that internal routine calls and the INTERPRET instruction protect DO loops by making them inactive. Therefore, for example, a LEAVE instruction in a subroutine cannot affect a DO loop in the calling routine.

You can cause this message to be issued if you use the SIGNAL instruction to transfer control within or into a loop. A SIGNAL instruction terminates all active loops, and any ITERATE or LEAVE instruction issued then would cause this message to be issued.

Error 29 **Environment name too long**

Explanation: The language processor encountered an environment name specified on an ADDRESS instruction that is longer than the allowed limit of 250 characters.

Error 30 **Name or string too long**

Explanation: The language processor found a variable name or a literal (quoted) string that is longer the allowed limit of 250 characters.

The limit for names includes any substitutions. A possible cause of this error is the use of a period (.) in a name, causing an unexpected substitution.

For a literal string, this error can be caused by leaving off an ending quote (or putting a single quote in a string) because several clauses may be included in the string. For example, the string 'don't' should be written as 'don''t' or "don't".

Error 31 Name starts with numeric or "."

Explanation: The language processor found a variable whose name began with a digit or a period. The REXX rules do not allow you to assign a value to a variable whose name begins with a digit or a period, because you could then redefine numeric constants.

Error 33 Invalid expression result

Explanation: The language processor encountered an expression result that is invalid in its particular context. The result may be invalid because an illegal FUZZ or DIGITS value was used in a NUMERIC instruction (FUZZ must be no larger than DIGITS). Another possibility is an invalid or missing expression following a VALUE keyword in an instruction.

Error 34 Logical value not 0 or 1

Explanation: The language processor found an expression in an IF, WHEN, DO WHILE, or DO UNTIL phrase that did not result in a 0 or 1. Any value operated on by a logical operator (¬, ¦, &, or &&) must result in a 0 or 1. For example, the phrase If result then exit rc will fail if result has a value other than 0 or 1. Thus, the phrase would be better written as If result¬=0 then exit rc.

Error 35 Invalid expression

Explanation: The language processor found a grammatical error in an expression. You might have ended an expression with an operator, had two adjacent operators with no data in between, or included special characters (such as operators) in an intended character expression without enclosing them in quotes. For example, in the OS/2 program, the command DIR C:\UTIL*.* should be written as DIR 'C:\UTIL*.*' (assuming DIR is not a variable) or even as 'DIR C:\UTIL*.*'.

Error 36 Unmatched "(" in expression

Explanation: The language processor found an unmatched parenthesis within an expression. You will get this message if you include a single parenthesis in a command without enclosing it in quotes.

Error 37 Unexpected "," or ")"

Explanation: The language processor found a comma (,) outside a routine invocation or too many right parentheses in an expression. You will get this message if you include a comma in a character expression without enclosing it in quotes. For example, the instruction:

```
Say Enter A, B, or C
```

should be written as:

```
Say 'Enter A, B, or C'
```

Error 38 Invalid template or pattern

Explanation: The language processor found an invalid special character, for example, % within a parsing template, or the syntax of a variable trigger was incorrect (no symbol was found after a left parenthesis). This message is also issued if the WITH subkeyword is omitted in a PARSE VALUE instruction.

Error 39 Evaluation stack overflow

Explanation: The language processor was not able to evaluate the expression because it is too complex (many nested parentheses, functions, and so on).

Error 40 Incorrect call to routine

Explanation: The language processor encountered an incorrect call to a built-in or external routine. Some possible causes are:

- You passed invalid data (arguments) to the routine.
- You passed too many arguments to the routine.
- The external routine invoked was not compatible with the language processor.

If you were not trying to invoke a routine, you may have a symbol or a string adjacent to a "(" when you meant it to be separated by a space or an operator. This causes it to be seen as a function call. For example, TIME(4+5) should probably be written as TIME*(4+5).

Error 41 Bad arithmetic conversion

Explanation: The language processor found a term in an arithmetic expression that was not a valid number or that had an exponent outside the allowed range of -999 999 999 to +999 999 999.

You may have mistyped a variable name, or included an arithmetic operator in a character expression without putting it in quotes. For example, the command DIR *prod.dat should be written as 'DIR *prod.dat' (in quotes);

otherwise, the language processor will try to multiply `DIR` by `prod.dat`.

Error 42 Arithmetic overflow/underflow

Explanation: The language processor encountered the result of an arithmetic operation that required an exponent greater than the limit of nine digits (more than 999 999 999 or less than -999 999 999).

This error can occur during evaluation of an expression (often as a result of trying to divide a number by 0) or during the stepping of a DO loop control variable.

Error 43 Routine not found

Explanation: The language processor was unable to find a routine called in your program. You invoked a function within an expression or invoked a subroutine by CALL, but:

- The specified label is not in the program.
- It is not the name of a built-in function.
- The language processor could not locate it externally.

The simplest, and probably most common, cause of this error is mistyping the name.

If you were not trying to invoke a routine, you may have put a symbol or string adjacent to a "(" when you meant it to be separated by a space or an operator. The language processor would see that as a function invocation. For example, the string `3(4+5)` should probably be written as `3*(4+5)`.

Error 44 Function did not return data

Explanation: The language processor invoked an external routine within an expression. The routine seemed to end without error, but it did not return data for use in the expression.

This may be due to specifying the name of a program that is not intended for use as a REXX function. It should be called as a command or subroutine.

Error 45 No data specified on function RETURN

Explanation: A REXX program has been called as a function, but an attempt is being made to return (by a RETURN instruction) without passing back any data. Similarly, an internal routine, called as a function, must end with a RETURN statement specifying an expression.

Error 46 Invalid variable reference

Explanation: The syntax of a variable reference is incorrect within a DROP, PARSE or PROCEDURE instruction. Check for a missing parenthesis or an incorrectly spelled variable name.

Error 47 Unexpected label

Explanation: A label was found in the instructions processed by an INTERPRET instruction. The label must be removed in order to work correctly.

Error 48 Failure in system service

Explanation: The language processor halts execution of the program because some system service, such as stream input or output or the manipulation of the external data queue, has failed to work correctly.

Error 49 Interpretation error

Explanation: The language processor has encountered a severe error while performing a self-consistency check.

These errors are issued by RXQUEUE and RXSUMCOM.

Error 115 The RXSUBCOM parameters are incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM accepts the following parameters:

- To register a subcommand environment:

```
RXSUBCOM REGISTER ENVIRONMENT_NAME DLL_NAME ENTRY_POINT
```

- To query a specific subcommand environment for existance:

```
RXSUBCOM QUERY ENVIRONMENT_NAME [dll_name]
```

- To drop a subcommand environment handler:

```
RXSUBCOM DROP ENVIRONMENT_NAME [dll_name]
```

- To load a subcommand environment from disk:

```
RXSUBCOM LOAD ENVIRONMENT_NAME [dll_name]
```

Error 116 The RXSUBCOM parameter REGISTER is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM REGISTER requires all of the following parameters:

```
RXSUBCOM REGISTER ENVIRONMENT_NAME DLL_NAME ENTRY_POINT
  ENVIRONMENT_NAME the name of the subcommand environment.
  DLL_NAME          the Dynalink Module name
  ENTRY_POINT       the name of the function to be executed when called
```

Error 117 The RXSUBCOM parameter DROP is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM DROP requires the environment name be specified.

```
RXSUBCOM DROP ENVIRONMENT_NAME [dll_name]
 ENVIRONMENT_NAME  the name of the subcommand environment
 DLL_NAME          the Dynalink Module name (optional)
```

Error 118 The RXSUBCOM parameter LOAD is incorrect.

Explanation: Check the RXSUBCOM parameters and retry the command. RXSUBCOM LOAD requires the environment name be specified.

```
RXSUBCOM LOAD ENVIRONMENT_NAME [DLL_NAME]
 ENVIRONMENT_NAME  the name of the subcommand environment
 DLL_NAME          the Dynalink Module name (optional)
```

Appendix B. Double-Byte Character Set (DBCS) Support

A Double-Byte Character Set supports languages that have more characters than can be represented by 8 bits (such as Korean Hangeul and Japanese Kanji). REXX has a full range of DBCS functions and handling techniques.

These include:

- String handling capabilities with DBCS characters
- OPTIONS modes that handle DBCS not only as literal strings, but also in data operations
- A number of functions that specifically support the processing of DBCS character strings
- Defined DBCS enhancements to current instructions and functions.

Note: The use of DBCS does not affect the meaning of the built-in functions as described in Chapter 4, "Functions" on page 4-1. There we described how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. The appendix describes how the resulting characters are represented as bytes. This internal representation is not usually seen if the results are printed. It may be seen if the results are displayed on certain terminals.

General Description

The following characteristics help define the rules used by DBCS to represent extended characters:

- Each DBCS character consists of 2 bytes.
- There are no DBCS control characters.
- The codes are within the ranges defined in the table, which shows the valid DBCS code for the DBCS blank.

Table B-1. DBCS Ranges

Byte	EBCDIC	ASCII
1st	X'41' to X'FE'	X'81' to X'FC'
2nd	X'41' to X'FE'	—
DBCS blank	X'4040'	X'8140'

Note: In ASCII, the 1st byte may vary from country to country, but is in the range defined in the table. (Japan, for example, uses only the ranges X'81' to X'9F' and X'E0' to X'FC' for the 1st byte.)

- DBCS alphanumeric and special symbols

  A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

  Here are some examples:

  X'42C1' is an EBCDIC double-byte A
  X'4281' is an EBCDIC double-byte a
  X'427D' is an EBCDIC double-byte quote

- No case translation

  In general, there is no concept of lowercase and uppercase in DBCS.

- Notational conventions

  This appendix uses the following notational conventions:

```
DBCS character              ->    .A .B .C .D
SBCS character              ->    a b c d e
DBCS blank                  ->    '. '
EBCDIC shift-out (X'0E')    ->    <
EBCDIC shift-in  (X'0F')    ->    >
```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

Enabling DBCS Data Operations

The OPTIONS instruction controls how REXX regards DBCS data. DBCS operations are enabled using the EXMODE option. (See "OPTIONS" on page 3-24 for more information.)

Pure DBCS Strings and Mixed SBCS/DBCS Strings

A *pure* DBCS string consists of only DBCS characters. A mixed SBCS/DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI bracket the DBCS data and distinguish it from the SBCS data. Since the SO and SI are only needed in the mixed strings, they are not associated with the pure DBCS strings.

In EBCDIC:

```
Pure DBCS string      ->      .A.B.C
Mixed string          ->      ab<.A.B>
Mixed string          ->      <.A.B>
```

In ASCII:

```
Pure DBCS string      ->      .A.B.C
Mixed string          ->      ab.A.B
```

Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If an invalid mixed string is used in one that does not allow invalid mixed strings under DBCS enabled mode, it causes a SYNTAX ERROR.

The following rules must be followed for mixed string validation:

- DBCS strings must be an even number of bytes in length.

EBCDIC only

- SO and SI must be paired in a string.
- Nesting of SO or SI is not permitted.

These examples show some possible misuses:

```
'ab<cd'       ->    INVALID - not paired
'<.A<.B>.C>   ->    INVALID - nested
'<.A.BC>'     ->    INVALID - odd byte length
```

When a variable is created, modified, or referred in a REXX program under OPTIONS EXMODE, it is validated whether it contains correct mixed string or not. When a referred variable contains invalid mixed string, it depends on the instruction, function, or operator whether it causes a syntax error.

Instruction Examples

Here are some examples that illustrate how instructions work with DBCS.

PARSE

In EBCDIC:

```
x1 = '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1
         w1   ->    '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 1 w1
         w1   ->    '<><.A.B><. . ><.E><.F><>'

PARSE VAR x1 w1 .
         w1   ->    '<.A.B>'
```

The leading and trailing SO and SI are unnecessary for word parsing and thus they are stripped off. However, one pair is still needed in order for a valid mixed DBCS string to be returned.

```
PARSE VAR x1 . w2
         w2   ->    '<. ><.E><.F><>'
```

Here the first blank delimited the word and the SO is added to the string to ensure the DBCS blank and the valid mixed string.

```
PARSE VAR x1 w1 w2
          w1    ->    '<.A.B>'
          w2    ->    '<. ><.E><.F><>'

PARSE VAR x1 w1 w2 .
          w1    ->    '<.A.B>'
          w2    ->    '<.E><.F>'
```

The word delimiting allows for unnecessary SO and SI to be dropped.

```
x2 = 'abc<>def <.A.B><><.C.D>'

PARSE VAR x2 w1 '' w2
          w1    ->    'abc<>def <.A.B><><.C.D>'
          w2    ->    ''

PARSE VAR x2 w1 '<>' w2
          w1    ->    'abc<>def <.A.B><><.C.D>'
          w2    ->    ''

PARSE VAR x2 w1 '<><>' w2
          w1    ->    'abc<>def <.A.B><><.C.D>'
          w2    ->    ''
```

Note that for the last three examples '', <>, and <><> are each a null string (a string of length 0). When parsing, the null string matches the end of string. For this reason, w1 is assigned the value of the entire string and w2 is assigned the null string.

And in ASCII:

```
x1 = '.A.B. . .E.F'

PARSE VAR x1 w1
          w1    ->    '.A.B. . .E.F'

PARSE VAR x1 1 w1
          w1    ->    '.A.B. . .E.F'

PARSE VAR x1 w1 .
          w1    ->    '.A.B'

PARSE VAR x1 . w2
          w2    ->    '. .E.F'

PARSE VAR x1 w1 w2
          w1    ->    '.A.B'
          w2    ->    '. .E.F'

PARSE VAR x1 w1 w2 .
          w1    ->    '.A.B'
          w2    ->    '.E.F'

x2 = 'abcdef .A.B.C.D'

PARSE VAR x2 w1 '' w2
          w1    ->    'abcdef .A.B.C.D'
          w2    ->    ''
```

SAY and TRACE

When the data is split up in shorter lengths, again the DBCS data integrity is kept under OPTIONS EXMODE. In EBCDIC, if the terminal line size is less than 4, the string is treated as SBCS data, because 4 is the minimum for mixed string data.

DBCS Function Handling

Some built-in functions can handle DBCS. The functions that deal with word delimiting and length determining conform with the following rules under OPTIONS EXMODE:

1. **Counting characters**—Logical character lengths are used when counting the length of a string (that is, 1 byte for one SBCS logical character, 2 bytes for one DBCS logical character). In EBCDIC, SO and SI are considered to be transparent, and are not counted, for every string operation.

2. **Character extraction from a string**—Characters are extracted from a string on a logical character basis. In EBCDIC, leading SO and trailing SI are not considered as part of one DBCS character. For instance, `.A` and `.B` are extracted from `<.A.B>`, and SO and SI are added to each DBCS character when they are finally preserved as completed DBCS characters. When multiple characters are consecutively extracted from a string, SO and SI that are between characters are also extracted. For example, `.A><.B` is extracted from `<.A><.B>`, and when the string is finally used as a completed string, the SO prefixes it and the SI suffixes it to give `<.A><.B>`.

   Here are some EBCDIC examples:

```
S1 = 'abc<>def'

SUBSTR(S1,3,1)      ->    'c'
SUBSTR(S1,4,1)      ->    'd'
SUBSTR(S1,3,2)      ->    'c<>d'

S2 = '<><.A.B><>'

SUBSTR(S2,1,1)      ->    '<.A>'
SUBSTR(S2,2,1)      ->    '<.B>'
SUBSTR(S2,1,2)      ->    '<.A.B>'
SUBSTR(S2,1,3,'x')  ->    '<.A.B><>x'

S3 = 'abc<><.A.B>'

SUBSTR(S3,3,1)      ->    'c'
SUBSTR(S3,4,1)      ->    '<.A>'
SUBSTR(S3,3,2)      ->    'c<><.A>'
DELSTR(S3,3,1)      ->    'ab<><.A.B>'
DELSTR(S3,4,1)      ->    'abc<><.B>'
DELSTR(S3,3,2)      ->    'ab<.B>'
```

   Here are some ASCII examples:

```
S2 = '.A.B'

SUBSTR(S2,1,1)       ->    '.A'
SUBSTR(S2,2,1)       ->    '.B'
SUBSTR(S2,1,2)       ->    '.A.B'
SUBSTR(S2,1,3,'x')   ->    '.A.Bx'

S3 = 'abc.A.B'

SUBSTR(S3,3,1)       ->    'c'
SUBSTR(S3,4,1)       ->    '.A'
SUBSTR(S3,3,2)       ->    'c.A'
DELSTR(S3,3,1)       ->    'ab.A.B'
DELSTR(S3,4,1)       ->    'abc.B'
DELSTR(S3,3,2)       ->    'ab.B'
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI and SO (or SO and SI) that are a result of string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.

4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than a SBCS if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and trailing contiguous SO and SI (or SI and SO) in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

   In EBCDIC, contiguous SO and SI (or SI and SO) between nonblank characters are also removed for comparison.

   Note: The strict comparison operators do not cause syntax errors even if invalid mixed strings are specified.

   In EBCDIC:

```
      '<.A>' = '<.A. >'     ->    1      /* true  */
  '<><><.A>' = '<.A><><>'   ->    1      /* true  */
  '<>  <.A>' = '<.A>'       ->    1      /* true  */
'<.A><><.B>' = '<.A.B>'     ->    1      /* true  */
       'abc' < 'ab<. >'     ->    0      /* false */
```

   In ASCII:

```
      '.A' = '.A. '         ->    1      /* true  */
    '. .A' = '.A'           ->    1      /* true  */
     'abc' < 'ab. '         ->    0      /* false */
```

5. **Word extraction from a string**—"Word" means that characters in a string are delimited by an SBCS or a DBCS blank.

   In EBCDIC, leading and trailing contiguous SO and SI (or SI and SO) are also removed when *words* are separated in a string, but contiguous SO and SI (or SI and SO) in a word are not removed or separated for word operations. Leading and trailing contiguous SO and SI (or SI and SO) of a word are not removed if they are among words that are extracted at the same time.

   In EBCDIC:

```
W1 = '<><. .A. . .B><.C. .D><>'

SUBWORD(W1,1,1)      ->    '<.A>'
SUBWORD(W1,1,2)      ->    '<.A. . .B><.C>'
SUBWORD(W1,3,1)      ->    '<.D>'
SUBWORD(W1,3)        ->    '<.D>'

W2 = '<.A. .B><.C><> <.D>'

SUBWORD(W2,2,1)      ->    '<.B><.C>'
SUBWORD(W2,2,2)      ->    '<.B><.C><> <.D>'
```

In ASCII:

```
W1 = '. .A. . .B.C. .D'

SUBWORD(W1,1,1)      ->    '.A'
SUBWORD(W1,1,2)      ->    '.A. . .B.C'
SUBWORD(W1,3,1)      ->    '.D'
SUBWORD(W1,3)        ->    '.D'

W2 = '.A. .B.C .D'

SUBWORD(W2,2,1)      ->    '.B.C'
SUBWORD(W2,2,2)      ->    '.B.C .D'
```

Built-in Function Examples

Examples for built-in functions, those that support DBCS and follow the rules defined, are given in this section. For full function descriptions and the syntax diagrams, refer to Chapter 4, "Functions" on page 4-1.

ABBREV

In EBCDIC:

```
ABBREV('<.A.B.C>','<.A.B>')     ->   1
ABBREV('<.A.B.C>','<.A.C>')     ->   0
ABBREV('<.A><.B.C>','<.A.B>')   ->   1
ABBREV('aa<>bbccdd','aabbcc')   ->   1
```

In ASCII:

```
ABBREV('.A.B.C','.A.B')         ->   1
ABBREV('.A.B.C','.A.C')         ->   0
```

Applying the character comparison and character extraction from a string rules.

COMPARE

In EBCDIC:

```
COMPARE('<.A.B.C>','<.A.B><.C>')     ->   0
COMPARE('<.A.B.C>','<.A.B.D>')       ->   3
COMPARE('ab<>cde','abcdx')           ->   5
COMPARE('<.A><>','<.A>','<. >')      ->   0
```

In ASCII:

```
COMPARE('.A.B.C','.A.B.C')           ->   0
COMPARE('.A.B.C','.A.B.D')           ->   3
COMPARE('abcde','abcdx')             ->   5
```

Applying the character concatenation for padding, character extraction from a string, and character comparison rules.

COPIES

In EBCDIC:

```
COPIES('<.A.B>',2)        ->    '<.A.B.A.B>'
COPIES('<.A><.B>',2)      ->    '<.A><.B.A><.B>'
COPIES('<.A.B><>',2)      ->    '<.A.B><.A.B><>'
```

In ASCII:

```
COPIES('.A.B',2)          ->    '.A.B.A.B'
```

Applying the character concatenation rule. Applying the character extraction from a string and character comparison rules.

INSERT and OVERLAY

In EBCDIC:

```
INSERT('a','b<><.A.B>',1)               -> 'ba<><.A.B>'
INSERT('<.A.B>','<.C.D><>',2)           -> '<.C.D.A.B><>'
INSERT('<.A.B>','<.C.D><><.E>',2)       -> '<.C.D.A.B><><.E>'
INSERT('<.A.B>','<.C.D><>',3,,'<.E>')   -> '<.C.D><.E.A.B>'

OVERLAY('<.A.B>','<.C.D><>',2)            -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',2)        -> '<.C.A.B>'
OVERLAY('<.A.B>','<.C.D><><.E>',3)        -> '<.C.D><><.A.B>'
OVERLAY('<.A.B>','<.C.D><>',4,,'<.E>')    -> '<.C.D><.E.A.B>'
OVERLAY('<.A>','<.C.D><.E>',2)            -> '<.C.A><.E>'
```

In ASCII:

```
INSERT('a','b.A.B',1)                   -> 'ba.A.B'
INSERT('.A.B','.C.D',2)                 -> '.C.D.A.B'
INSERT('.A.B','.C.D.E',2)               -> '.C.D.A.B.E'
INSERT('.A.B','.C.D',3,,'.E')           -> '.C.D.E.A.B'

OVERLAY('.A.B','.C.D',2)                -> '.C.A.B'
OVERLAY('.A.B','.C.D.E',2)              -> '.C.A.B'
OVERLAY('.A.B','.C.D.E',3)              -> '.C.D.A.B'
OVERLAY('.A.B','.C.D',4,,'.E')          -> '.C.D.E.A.B'
OVERLAY('.A','.C.D.E',2)                -> '.C.A.E'
```

Applying the character extraction from a string and character comparison rules.

LEFT, RIGHT, and CENTER

In EBCDIC:

```
LEFT('<.A.B.C.D.E>',4)        ->    '<.A.B.C.D>'
LEFT('a<>',2)                 ->    'a<> '
LEFT('<.A>',2,'*')            ->    '<.A>*'
RIGHT('<.A.B.C.D.E>',4)       ->    '<.B.C.D.E>'
RIGHT('a<>',2)                ->    ' a'
CENTER('<.A.B>',10,'<.E>')    ->    '<.E.E.E.E.A.B.E.E.E.E>'
CENTER('<.A.B>',11,'<.E>')    ->    '<.E.E.E.E.A.B.E.E.E.E.E>'
CENTER('<.A.B>',10,'e')       ->    'eeee<.A.B>eeee'
```

In ASCII:

```
LEFT('.A.B.C.D.E',4)       ->   '.A.B.C.D'
LEFT('a',2)                ->   'a '
LEFT('.A',2,'*')           ->   '.A*'
RIGHT('.A.B.C.D.E',4)      ->   '.B.C.D.E'
RIGHT('a',2)               ->   ' a'
CENTER('.A.B',10,'.E')     ->   '.E.E.E.E.A.B.E.E.E.E'
CENTER('.A.B',11,'.E')     ->   '.E.E.E.E.A.B.E.E.E.E.E'
CENTER('.A.B',10,'e')      ->   'eeee.A.Beeee'
```

Applying the character concatenation for padding and character extraction from a string rules.

LENGTH

In EBCDIC:

```
LENGTH('<.A.B><.C.D><>')       ->   4
```

In ASCII:

```
LENGTH('.A.B.C.D')             ->   4
```

Applying the counting characters rule.

LINEIN

When reading a line from a stream under OPTIONS EXMODE, the LINEIN function ignores any DBCS strings when searching for line-end characters. The DBCS string itself is read normally from the stream.

REVERSE

In EBCDIC:

```
REVERSE('<.A.B><.C.D><>')       ->   '<><.D.C><.B.A>'
```

In ASCII:

```
REVERSE('.A.B.C.D')             ->   '.D.C.B.A'
```

Applying the character extraction from a string and character concatenation rules.

SPACE

In EBCDIC:

```
SPACE('a<.A.B. .C.D>',1)          ->  'a<.A.B> <.C.D>'
SPACE('a<.A><><. .C.D>',1,'x')    ->  'a<.A>x<.C.D>'
SPACE('a<.A><. .C.D>',1,'<.E>')   ->  'a<.A.E.C.D>'
```

In ASCII:

```
SPACE('a.A.B. .C.D',1)            ->  'a.A.B .C.D'
SPACE('a.A. .C.D',1,'x')          ->  'a.Ax.C.D'
SPACE('a.A. .C.D',1,'.E')         ->  'a.A.E.C.D'
```

Applying the word extraction from a string and character concatenation rules.

STRIP

In EBCDIC:

```
STRIP('<><.A><.B><.A><>',,'<.A>')    ->  '<.B>'
```

In ASCII:

```
STRIP('.A.B.A',,'.A')                ->  '.B'
```

Applying the character extraction from a string and character concatenation rules.

SUBSTR and DELSTR

In EBCDIC:

```
SUBSTR('<><.A><><.B><.C.D>',1,2)  ->  '<.A><><.B>'
DELSTR('<><.A><><.B><.C.D>',1,2)  ->  '<><.C.D>'
SUBSTR('<.A><><.B><.C.D>',2,2)    ->  '<.B><.C>'
DELSTR('<.A><><.B><.C.D>',2,2)    ->  '<.A><><.D>'
SUBSTR('<.A.B><>',1,2)            ->  '<.A.B>'
SUBSTR('<.A.B><>',1)              ->  '<.A.B><>'
```

In ASCII:

```
SUBSTR('.A.B.C.D',1,2)            ->  '.A.B'
DELSTR('.A.B.C.D',1,2)            ->  '.C.D'
SUBSTR('.A.B.C.D',2,2)            ->  '.B.C'
DELSTR('.A.B.C.D',2,2)            ->  '.A.D'
SUBSTR('.A.B',1,2)                ->  '.A.B'
SUBSTR('.A.B',1)                  ->  '.A.B'
```

Applying the character extraction from a string and character concatenation rules.

SUBWORD and DELWORD

In EBCDIC:

```
SUBWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<.A. . .B><.C>'
DELWORD('<><. .A. . .B><.C. .D>',1,2) ->  '<><. .D>'
SUBWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<.A. . .B><.C>'
DELWORD('<><.A. . .B><.C. .D>',1,2)   ->  '<><.D>'
SUBWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.A. .B><.C>'
DELWORD('<.A. .B><.C><> <.D>',1,2)    ->  '<.D>'
```

In ASCII:

```
SUBWORD('. .A. . .B.C. .D',1,2)       ->  '.A. . .B.C'
DELWORD('. .A. . .B.C. .D',1,2)       ->  '. .D'
SUBWORD('.A. . .B.C. .D',1,2)         ->  '.A. . .B.C'
DELWORD('.A. . .B.C. .D',1,2)         ->  '.D'
SUBWORD('.A. .B.C .D',1,2)            ->  '.A. .B.C'
DELWORD('.A. .B.C .D',1,2)            ->  '.D'
```

Applying the word extraction from a string and character concatenation rules.

TRANSLATE

In EBCDIC:

```
TRANSLATE('abcd','<.A.B.C>','abc')          ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','abc')        ->  '<.A.B.C>d'
TRANSLATE('abcd','<><.A.B.C>','ab<>c')      ->  '<.A.B.C>d'
TRANSLATE('a<>bcd','<><.A.B.C>','ab<>c')    ->  '<.A.B.C>d'
TRANSLATE('a<>xcd','<><.A.B.C>','ab<>c')    ->  '<.A>x<.C>d'
```

In ASCII:

```
TRANSLATE('abcd','.A.B.C','abc')            ->  '.A.B.Cd'
TRANSLATE('axcd','.A.B.C','abc')            ->  '.Ax.Cd'
```

Applying the character extraction from a string, character comparison, and character concatenation rules.

VERIFY

In EBCDIC:

```
VERIFY('<><>.A.B><><.X>','<.B.A.C.D.E>')    -> 3
```

In ASCII:

```
VERIFY('.A.B.X','.B.A.C.D.E')               -> 3
```

Applying the character extraction from a string and character comparison rules.

WORD, WORDINDEX, and WORDLENGTH

In EBCDIC:

```
X = '<><. .A. . .B><.C. .D>'

WORD(X,1)          ->    '<.A>'
WORDINDEX(X,1)     ->    2
WORDLENGTH(X,1)    ->    1

Y = '<><.A. . .B><.C. .D>'

WORD(Y,1)          ->    '<.A>'
WORDINDEX(Y,1)     ->    1
WORDLENGTH(Y,1)    ->    1

Z = '<.A  .B><.C> <.D>'

WORD(Z,2)          ->    '<.B><.C>'
WORDINDEX(Z,2)     ->    3
WORDLENGTH(Z,2)    ->    2
```

In ASCII:

```
X = '. .A. . .B.C. .D'

WORD(X,1)          ->    '.A'
WORDINDEX(X,1)     ->    2
WORDLENGTH(X,1)    ->    1

Y = '.A. . .B.C. .D'

WORD(Y,1)          ->    '.A'
WORDINDEX(Y,1)     ->    1
WORDLENGTH(Y,1)    ->    1

Z = '.A. .B.C .D'
```

Where .A is followed by a DBCS blank
and .C is followed by an SBCS blank

```
WORD(Z,2)          ->    '.B.C'
WORDINDEX(Z,2)     ->    3
WORDLENGTH(Z,2)    ->    2
```

Applying the word extraction from a string and (for WORDINDEX and WORDLENGTH) counting characters rules.

WORDS

In EBCDIC:

```
X = '<><. .A. . .B><.C. .D>'

WORDS(X)          ->   3
```

In ASCII:

```
X = '. .A. . .B.C. .D'

WORDS(X)          ->   3
```

Applying the word extraction from a string rule.

WORDPOS

In EBCDIC:

```
WORDPOS('<.B.C> abc','<.A. .B.C> abc')              ->  2
WORDPOS('<.A.B>','<.A.B. .A.B><. .B.C. .A.B>',3)  ->  4
```

In ASCII:

```
WORDPOS('.B.C abc','.A. .B.C abc')               ->  2
WORDPOS('.A.B','.A.B. .A.B. .B.C. .A.B',3)       ->  4
```

Applying the word extraction from a string and character comparison rules.

DBCS Processing Functions

This section describes the functions that support DBCS mixed strings. These functions handle mixed strings regardless of the OPTIONS mode.

Note: When used with DBCS functions, *length* is always measured in bytes (as opposed to LENGTH(string) which is measured in characters).

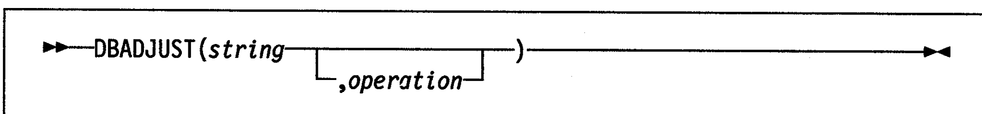
Counting Option

In EBCDIC, when specified in the functions, the counting option can control whether or not the SO and SI are considered present when determining the length. **Y** specifies counting SO and SI within mixed strings. **N** specifies *not* to count the SO and SI, and is the default.

In ASCII, the count options is ignored if **Y** is specified and the default **N** is always in effect. As such, EBCDIC and ASCII implementations yield different results.

Function Descriptions

DBADJUST



In ASCII, merely returns the input string. In EBCDIC, adjusts all contiguous SI and SO (or SO and SI) characters in *string* based on the *operation* specified. The following are valid *operations*. Only the capitalized and boldfaced letter is needed; all characters following it are ignored.

Blank changes contiguous characters to blanks (X'4040').

Remove removes contiguous characters, and is the default.

Here are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b','B')    ->    '<.A. .B>a  b'
DBADJUST('<.A><.B>a<>b','R')    ->    '<.A.B>ab'
DBADJUST('<><.A.B>','B')        ->    '<. .A.B>'
```

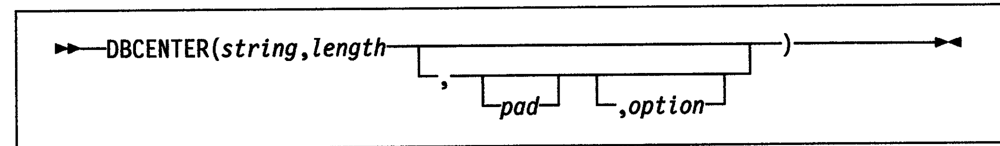
DBBRACKET

In ASCII, merely returns the input string. In EBCDIC, adds SO and SI brackets to a pure DBCS string. If *string* is not a pure DBCS string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some EBCDIC examples:

```
DBBRACKET('.A.B')       ->    '<.A.B>'
DBBRACKET('abc')        ->    SYNTAX error
DBBRACKET('<.A.B>')     ->    SYNTAX error
```

DBCENTER



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

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* "Y" is ignored and "N" is always in effect.

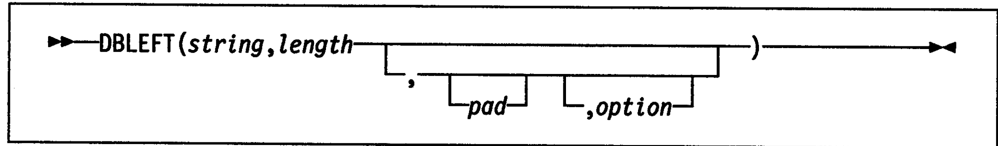
Here are some EBCDIC examples:

```
DBCENTER('<.A.B.C>',4)                ->    ' <.B> '
DBCENTER('<.A.B.C>',3)                ->    ' <.B>'
DBCENTER('<.A.B.C>',10,'x')           ->    'xx<.A.B.C>xx'
DBCENTER('<.A.B.C>',10,'x','Y')       ->    'x<.A.B.C>x'
DBCENTER('<.A.B.C>',4,'x','Y')        ->    '<.B>'
DBCENTER('<.A.B.C>',5,'x','Y')        ->    'x<.B>'
DBCENTER('<.A.B.C>',8,'<.P>')         ->    ' <.A.B.C> '
DBCENTER('<.A.B.C>',9,'<.P>')         ->    ' <.A.B.C.P>'
DBCENTER('<.A.B.C>',10,'<.P>')        ->    '<.P.A.B.C.P>'
DBCENTER('<.A.B.C>',12,'<.P>','Y')    ->    '<.P.A.B.C.P>'
```

Here are some ASCII examples:

```
DBCENTER('.A.B.C',9,'.P')      ->    ' .A.B.C.P'
DBCENTER('.A.B.C',10,'.P')     ->    '.P.A.B.C.P'
```

DBLEFT



returns a string of length *length* containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* **Y** is ignored and **N** is always in effect.

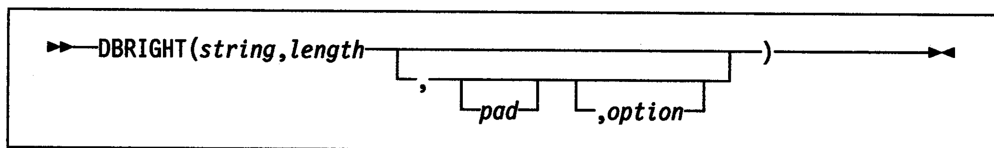
Here are some EBCDIC examples:

```
DBLEFT('ab<.A.B>',4)              ->    'ab<.A>'
DBLEFT('ab<.A.B>',3)              ->    'ab '
DBLEFT('ab<.A.B>',4,'x','Y')      ->    'abxx'
DBLEFT('ab<.A.B>',3,'x','Y')      ->    'abx'
DBLEFT('ab<.A.B>',8,'<.P>')       ->    'ab<.A.B.P>'
DBLEFT('ab<.A.B>',9,'<.P>')       ->    'ab<.A.B.P> '
DBLEFT('ab<.A.B>',8,'<.P>','Y')   ->    'ab<.A.B>'
DBLEFT('ab<.A.B>',9,'<.P>','Y')   ->    'ab<.A.B> '
```

Here are some ASCII examples:

```
DBLEFT('ab.A.B',3,,'Y')           ->    'ab '
DBLEFT('ab.A.B',9,'.P')           ->    'ab.A.B.P '
```

DBRIGHT



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

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* **Y** is ignored and **N** is always in effect.

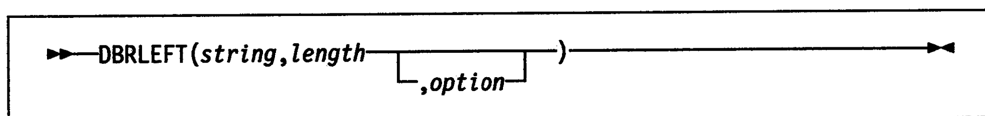
Here are some EBCDIC examples:

```
DBRIGHT('ab<.A.B>',4)                ->    '<.A.B>'
DBRIGHT('ab<.A.B>',3)                ->    ' <.B>'
DBRIGHT('ab<.A.B>',5,'x','Y')        ->    'x<.B>'
DBRIGHT('ab<.A.B>',10,'x','Y')       ->    'xxab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>')         ->    '<.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',9,'<.P>')         ->    ' <.P>ab<.A.B>'
DBRIGHT('ab<.A.B>',8,'<.P>','Y')     ->    'ab<.A.B>'
DBRIGHT('ab<.A.B>',11,'<.P>','Y')    ->    '   ab<.A.B>'
DBRIGHT('ab<.A.B>',12,'<.P>','Y')    ->    '<.P>ab<.A.B>'
```

Here are some ASCII examples:

```
DBRIGHT('ab.A.B',3)                  ->    ' .B'
DBRIGHT('ab.A.B',12,'.P','Y')        ->    '.P.P.Pab.A.B'
```

DBRLEFT



returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* **Y** is ignored and **N** is always in effect.

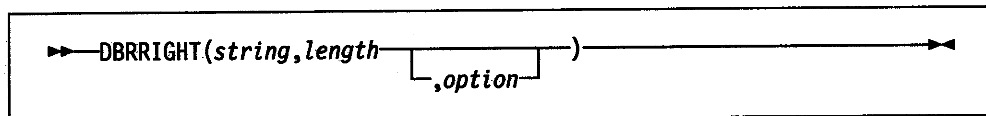
Here are some EBCDIC examples:

```
DBRLEFT('ab<.A.B>',4)                ->    '<.B>'
DBRLEFT('ab<.A.B>',3)                ->    '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')            ->    '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')            ->    '<.A.B>'
DBRLEFT('ab<.A.B>',8)                ->    ''
DBRLEFT('ab<.A.B>',9,'Y')            ->    ''
```

Here are some ASCII examples:

```
DBRLEFT('ab.A.B',3)                  ->    '.A.B'
DBRLEFT('ab.A.B',9,'Y')              ->    ''
```

DBRRIGHT



returns the remainder from the DBRIGHT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* **Y** is ignored and **N** is always in effect.

Here are some EBCDIC examples:

```
DBRRIGHT('ab<.A.B>',4)          ->    'ab'
DBRRIGHT('ab<.A.B>',3)          ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5)          ->    'a'
DBRRIGHT('ab<.A.B>',4,'Y')      ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')      ->    'ab<.A>'
DBRRIGHT('ab<.A.B>',8)          ->    ''
DBRRIGHT('ab<.A.B>',8,'Y')      ->    ''
```

Here are some ASCII examples:

```
DBRRIGHT('ab.A.B',3)            ->    'ab.A'
DBRRIGHT('ab.A.B',8)            ->    ''
```

DBTODBCS

```
>>──DBTODBCS(string)──────────────────────────────><
```

converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

Here are some EBCDIC examples:

```
DBTODBCS('Rexx 1988')      ->    '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')      ->    '<.A. .B>'
```

Here are some ASCII examples:

```
DBTODBCS('Rexx 1988')      ->    '.R.e.x.x. .1.9.8.8'
DBTODBCS('.A .B')          ->    '.A. .B'
```

Note: In these examples, the `.x` is the DBCS character corresponding to an SBCS `x`.

DBTOSBCS

```
>>──DBTOSBCS(string)──────────────────────────────><
```

converts all passed, valid DBCS characters (including the DBCS blank) within *string* to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

Here are some EBCDIC examples:

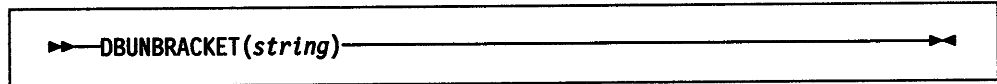
```
DBTOSBCS('<.S.d>/<.2.-.1>')     ->    'Sd/2-1'
DBTOSBCS('<.X. .Y>')            ->    '<.X> <.Y>'
```

Here are some ASCII examples:

```
DBTOSBCS('.S.d/.2.-.1')         ->    'Sd/2-1'
DBTOSBCS('.X. .Y')              ->    '.X .Y'
```

Note: In these examples, the `.d` is the DBCS character corresponding to an SBCS `d`. But the `.X` and `.Y` do not have a corresponding SBCS character and are not converted.

DBUNBRACKET

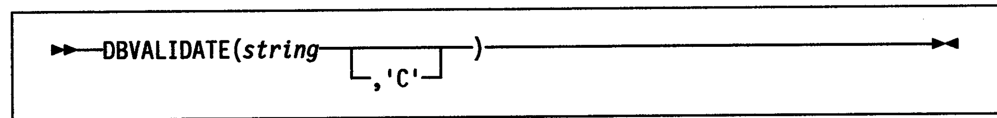


In ASCII, merely returns the input string. In EBCDIC, removes the SO and SI brackets from a pure DBCS *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')      ->    '.A.B'
DBUNBRACKET('ab<.A>')      ->    SYNTAX error
```

DBVALIDATE



returns 1 if the *string* is a valid mixed string or SBCS string. Otherwise, returns 0. Mixed string validation rules are:

1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only — Proper SO and SI pairing.

In ASCII, the option **C** has no effect. In EBCDIC, if **C** is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on (that is, in EBCDIC, the leftmost byte range is from X'41' to X'FE').

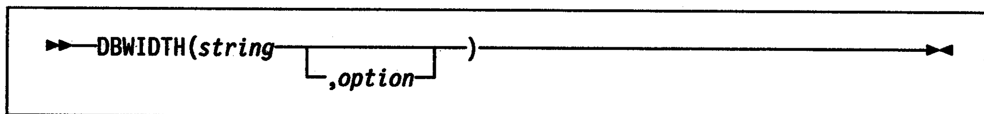
Here are some EBCDIC examples:

```
x='abc<de'

DBVALIDATE('ab<.A.B>')          ->    1
DBVALIDATE(x)                   ->    0

y='C1C20E111213140F'X

DBVALIDATE(y)                   ->    1
DBVALIDATE(y,'C')               ->    0
```

Here are some ASCII examples:

```
DBVALIDATE('ab.A.B')  ->    1
DBVALIDATE('ab.A.')   ->    0  /* widow left DBCS byte */

y='C1C2FCFFFCFF'X       /* 'FCFF'x will pass DBCS range check */

DBVALIDATE(y)                   ->    1
DBVALIDATE(y,'C')               ->    1
```

DBWIDTH



returns the length of *string* in bytes.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

In ASCII, *option* **Y** is ignored and **N** is always in effect.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>','Y')    ->    8
DBWIDTH('ab<.A.B>','N')    ->    6
```

Here are some ASCII examples:

```
DBWIDTH('ab.A.B','Y')      ->    6
DBWIDTH('ab.A.B','N')      ->    6
```

Index

A

ABBREV function
  description 4-6
  using to select a default 4-6
abbreviations
  testing with ABBREV function 4-6
abnormal change in flow of control 7-1
ABS function 4-7
absolute positional patterns 5-4
absolute value
  finding using ABS function 4-7
  used with power 6-5
abuttal 2-8
active loops 3-19
add external function 4-47
addition
  definition 6-4
  operator 2-9
ADDRESS
  function 4-7
  instruction 3-2
  settings saved during subroutine calls 3-8
algebraic precedence 2-11
alphabetics
  checking with DATATYPE 4-16
  used as symbols 2-4
alphanumeric checking with DATATYPE 4-16
altering
  flow within a repetitive DO loop 3-19
  special variables 2-17
AND operator 2-10
ANDing character strings together 4-9
AND, logical 2-10
API functions
API return codes
application programming interfaces 8-10
  application data structures 8-10
    RXSYSEXIT 9-7
    SHVBLOCK 9-45
  exit interfaces 8-10
    RexxDeregisterExit 9-42
    RexxQueryExit 9-43
    RexxRegisterExitDll 9-40
    RexxRegisterExitExe 9-41
  external function interfaces 8-10
    RexxDeregisterFunction 9-23
    RexxRegisterFunctionDll 9-20
    RexxRegisterFunctionExe 9-22
  halt and trace interfaces 8-10
    RexxResetTrace 9-55
    RexxSetHalt 9-53
    RexxSetTrace 9-54
  handler definitions 8-10
    exit handler 9-26
application programming interfaces *(continued)*
  handler definitions *(continued)*
    external function 9-18
    RexxSubcomHandler 9-9
    subcommand handler 9-9
  invoking REXX interpreter 8-10
    RexxStart 9-5
  macrospace interfaces 8-10
    RexxAddMacro 9-57
    RexxClearMacroSpace 9-59
    RexxDropMacro 9-58
    RexxLoadMacroSpace 9-61
    RexxQueryMacro 9-62
    RexxReorderMacro 9-63
    RexxSaveMacroSpace 9-60
  RXSTRING 9-3
  RXSYSEXIT 9-27
  subcommand interfaces 8-10
    RexxDeregisterSubcom 9-14
    RexxQuerySubcom 9-15
    RexxRegisterSubcomDll 9-12
    RexxRegisterSubcomExe 9-13
  variable pool interface 8-10
    RexxVariablePool 9-45
ARG function 4-7
ARG instruction 3-4
ARG option of PARSE instruction 3-25
arguments
  checking with ARG function 4-7
  of functions 3-4, 4-1
  of programs 3-4
  of subroutines 3-4, 3-6
  passing to functions 4-1
  retrieving with ARG function 4-7
  retrieving with ARG instruction 3-4
  retrieving with the PARSE ARG instruction 3-25
arithmetic
  combination rules 6-3
  comparisons 6-6
  errors 6-9
  NUMERIC settings 3-22
  operators 2-9, 6-1, 6-3
  overflow 6-9
  precision 6-2
  underflow 6-9
array
  initialization of 2-16
  setting up 2-14
assigning data to variables 3-25
assignment
  description of 2-13
  multiple assignments 5-6
  of compound variables 2-14, 2-16

assignment indicator (=) 2-13
associative storage 2-14

B

backslash, use of 2-6, 2-10
BASE option of DATE function 4-17
BEEP function 4-8
binary digits 2-4
binary strings 2-4
binary to hexadecimal conversion 4-10
BITAND function 4-9
BITOR function 4-9
bits checked using DATATYPE 4-16
BITXOR function 4-10
blank removal with STRIP function 4-36
blanks
  adjacent to special character 2-2
  as concatenation operator 2-8
  in parsing, treatment of 5-2
boolean operations 2-10
bottom of program reached during execution 3-15
bracketed DBCS strings
  DBBRACKET function B-13
  DBUNBRACKET function B-17
built-in function invoking 3-6
built-in functions
  ABBREV 4-6
  ABS 4-7
  ADDRESS 4-7
  ARG 4-7
  BEEP 4-8
  BITAND 4-9
  BITOR 4-9
  BITXOR 4-10
  B2X 4-10
  CENTER 4-11
  CENTRE 4-11
  CHARIN 4-11
  CHAROUT 4-12
  CHARS 4-13
  COMPARE 4-13
  CONDITION 4-14
  COPIES 4-15
  C2D 4-15
  C2X 4-16
  DATATYPE 4-16
  DATE 4-17
  DBCS functions B-12
  DELSTR 4-19
  DELWORD 4-19
  description of 4-5
  DIGITS 4-19
  DIRECTORY 4-20
  Double-Byte Character Set functions B-12
  D2C 4-20
  D2X 4-21
  ENDLOCAL 4-21
built-in functions *(continued)*
  ERRORTEXT 4-22
  FILESPEC 4-22
  FORM 4-23
  FORMAT 4-23
  FUZZ 4-24
  INSERT 4-24
  LASTPOS 4-24
  LEFT 4-25
  LENGTH 4-25
  LINEIN 4-25
  LINEOUT 4-27
  LINES 4-28
  MAX 4-29
  MIN 4-29
  OVERLAY 4-29
  POS 4-30
  QUEUED 4-30
  RANDOM 4-30
  REVERSE 4-31
  RIGHT 4-31
  SETLOCAL 4-32
  SIGN 4-32
  SOURCELINE 4-33
  SPACE 4-33
  STREAM 4-33
  STRIP 4-36
  SUBSTR 4-37
  SUBWORD 4-37
  SYMBOL 4-37
  TIME 4-38
  TRACE 4-40
  TRANSLATE 4-40
  TRUNC 4-41
  VALUE 4-41
  VERIFY 4-43
  WORD 4-43
  WORDINDEX 4-43
  WORDLENGTH 4-44
  WORDPOS 4-44
  WORDS 4-44
  XRANGE 4-45
  X2B 4-45
  X2C 4-45
  X2D 4-46
BY phrase of DO instruction 3-9
B2X function 4-10

C

CALL command (OS/2) 12-1
CALL instruction 3-6
CENTER function 4-11
centering a string using CENTER function 4-11
centering a string using CENTRE function 4-11
CENTRE function 4-11
changing destination of commands 3-2

Character input and output 8-1—8-10
  See also Default character streams
  See also External character streams
  See also Files
  See also Line input and output
  See also Serial input and output
  See also Stream
  See also Typewriter input and output
Character input streams 8-2
Character output streams 8-2
character position of a string 4-24
character removal with STRIP function 4-36
character to decimal conversion 4-15
character to hexadecimal conversion 4-16
CHARIN function 4-11
  role in input and output 8-1
CHAROUT function 4-12
  role in input and output 8-1
CHARS function 4-13
  role in input and output 8-1
clauses
  as labels 2-12
  assignment 2-13
  continuation of 2-7
  description of 2-2
  null 2-12
CMD command environment 3-2
codes, error A-1
collating sequence using XRANGE 4-45
Collections of variables 4-41
COLLECTOR example program 8-9
colon
  as a special character 2-6
  in a label 2-12
colon as label terminators 2-12
combination, arithmetic 6-3
comma
  as continuation character 2-7
  in CALL instruction 3-6
  in function calls 4-1
  separator of arguments 3-6, 4-1
  within a parsing template 5-10
  within a parsing template list 3-4
comma in parsing 5-10
command errors, trapping 7-1
command inhibition
  See TRACE instruction
command line editing, OS/2 2-19
commands
  alternative destinations 2-17
  destination of 3-2
  issuing to host 2-17
commands, editing 2-19
comments
  description of 2-2
COMPARE function 4-13
comparisons
  of numbers 2-9, 6-6
comparisons *(continued)*
  of strings 2-9
    using COMPARE 4-13
compound symbols 2-14
compound variable
  description of 2-14
  setting new value 2-16
concatenation of strings 2-8
concatenation operator
  abuttal 2-8
  blank 2-8
  || 2-8
concatenation operator, OS/2 2-6
CONDITION function 4-14
condition trap information using CONDITION 4-14
conditional loops 3-9
conditions
  ERROR 7-1
  FAILURE 7-1
  HALT 7-1
  NOTREADY 7-1
  NOVALUE 7-2
  saved during subroutine calls 3-8
  SYNTAX 7-2
conditions, trapping of 7-1
console
  reading from with PULL 3-29
  writing to with SAY 3-33
constant symbols 2-14
content addressable storage 2-14
continuation
  character 2-7
  of clauses 2-7
  of data for display 3-33
control variable 3-10
controlled loops 3-10
conversion
  binary to hexadecimal 4-10
  character to decimal 4-15
  character to hexadecimal 4-16
  decimal to character 4-20
  decimal to hexadecimal 4-21
  formatting numbers 4-23
  hexadecimal to binary 4-45
  hexadecimal to character 4-45
  hexadecimal to decimal 4-46
conversion functions 4-5—4-46
COPIES function 4-15
copying a string using COPIES 4-15
count from stream 4-12
counting words in a string 4-44
create external data queue 4-47
C2D function 4-15
C2X function 4-16

D

data length 2-8
data terms 2-7
DATATYPE function 4-16
date and version of the language processor 3-26
DATE function 4-17
DBADJUST function B-12
DBBRACKET function B-13
DBCENTER function B-13
DBCS functions
  DBADJUST B-12
  DBBRACKET B-13
  DBCENTER B-13
  DBLEFT B-14
  DBRIGHT B-14
  DBRLEFT B-15
  DBRRIGHT B-15
  DBTODBCS B-16
  DBTOSBCS B-16
  DBUNBRACKET B-17
  DBVALIDATE B-17
  DBWIDTH B-18
DBCS handling B-1
DBCS strings 3-24, B-1
DBCS (Double-Byte Character Set) characters B-1
DBLEFT function B-14
DBRIGHT function B-14
DBRLEFT function B-15
DBRRIGHT function B-15
DBTODBCS function B-16
DBTOSBCS function B-16
DBUNBRACKET function B-17
DBVALIDATE function B-17
DBWIDTH function B-18
debugging programs
  See interactive debug
  See TRACE instruction
debug, interactive 3-37
decimal arithmetic 6-1—6-9
decimal to character conversion 4-20
decimal to hexadecimal conversion 4-21
Default character streams 8-1
default environment 2-17
Delayed state
  of NOTREADY condition 8-8
deleting part of a string 4-19
deleting words from a string 4-19
delimiters in a clause
  See colon
  See semicolons
DELSTR function 4-19
DELWORD function 4-19
derived name 2-14
derived names of variables 2-14
DIGITS function 4-19
DIGITS option of NUMERIC instruction 3-22, 6-2
DIRECTORY function 4-20
displaying data
  See SAY instruction
division
  definition 6-4
  operator 2-9
dll functions 4-49
DO instruction 3-9—3-13
  See also loops
DosAllocMem 9-3, 9-7, 9-10, 9-18, 9-27
DosFreeMem 9-6, 9-7, 9-8, 9-19, 9-22, 9-47, 9-48
Double-Byte Character Set (DBCS) strings 3-24, B-1
drop external function 4-47
DROP instruction 3-14
dummy instruction
  See NOP instruction
dynamic link library (RexxUtil) 4-49
D2C function 4-20
D2X function 4-21

E

editing commands, OS/2 system 2-19
elapsed time saved during subroutine calls 3-8
elapsed-time clock 3-8, 4-38
ELSE keyword
  See IF instruction
END clause
  See also DO instruction
  See also SELECT instruction
  specifying control variable 3-10
ENDLOCAL function 4-21
engineering notation 6-8
environment variable, OS/2 4-41
environments
  addressing of 3-2
  default 3-3, 3-26
  determining current using ADDRESS function 4-7
  SAA supported 1-1
  temporary change of 3-2
equal operator 2-9
equal sign in parsing 5-5
equality, testing of 2-9
error codes A-1
ERROR condition of SIGNAL and CALL
 instructions 7-4
error messages
  retrieving with ERRORTEXT 4-22
error messages and codes A-1
errors
  during execution of functions 4-4
  during stream input and output 8-6
  from commands 2-17
  syntax A-1
  traceback after 3-40
errors, trapping 7-1
ERRORTEXT function 4-22

ETMODE 3-24
EUROPEAN option of DATE function 4-18
evaluation of expressions 2-8
Examples of programs 8-8
exception conditions saved during subroutine calls 3-8
exclusive OR operator 2-10
exclusive ORing character strings together 4-10
execution by language processor 2-1
execution of data 3-17
EXIT instruction 3-15
EXMODE 3-24, B-2
exponential notation
  definition 6-7
  description of 6-1
  usage 2-5
exponentiation
  definition 6-7
  operator 2-9
EXPOSE option of PROCEDURE instruction 3-27
expressions
  evaluation 2-8
  examples 2-11
  parsing of 3-26
  results of 2-8
  tracing results of 3-38
External character streams 8-1
external data queue
  counting lines in 4-30
  creating and deleting queues 4-47
  detached processes 8-6
  in REXX and OS/2 8-4
  naming and querying 4-47
  private 8-5
  reading from with PULL 3-29
  RXQUEUE function 4-47
  session 8-5
  writing to with PUSH 3-30
  writing to with QUEUE 3-31
external function interfaces 8-10
external functions
  description of 4-2
  search order 4-3
external routine invoking 3-6
external subroutines
  description of 4-2
External variables
  access with VALUE function 4-41
extracting a substring 4-37
extracting words from a string 4-37

F

FAILURE condition of SIGNAL and CALL instructions 7-1
FIFO (first-in/first-out) stacking 3-31
FILECOPY example program 8-8
filename, extension, path of program 3-26
Files 8-1
FILESPEC function 4-22
finding a mismatch using COMPARE 4-13
finding the length of a string 4-25
flow control
  abnormal, with CALL 7-1
  abnormal, with SIGNAL 7-1
  with CALL/RETURN 3-6
  with DO construct 3-9
  with IF construct 3-16
  with SELECT construct 3-34
FOR phrase of DO instruction 3-9
FOREVER repetitor on DO instruction 3-9
FORM function 4-23
FORM option of NUMERIC instruction 3-22, 6-8
FORMAT function 4-23
formatting
  DBCS blank adjustments B-12
  DBCS bracket adding B-13
  DBCS bracket stripping B-17
  DBCS EBCDIC to DBCS B-16
  DBCS string width B-18
  DBCS strings to SBCS B-16
  numbers for display 4-23
  numbers with TRUNC 4-41
  of output during tracing 3-39
  text centering 4-11
  text left justification 4-25, B-14
  text left remainder justification B-15
  text right justification 4-31, B-13, B-14
  text right remainder justification B-15
  text spacing 4-33
  text validation function B-17
functions
  built-in 4-5, 4-6
  description of 4-1
  external 4-2
  forcing built-in or external reference 4-2
  internal 4-2
  invocation of 4-1
  numeric arguments of 6-9
  return from 3-32
  variables in 3-27
function, built-in
  See built-in functions
FUZZ
  controlling numeric comparison 6-7
  option of NUMERIC instruction 3-22, 6-7
FUZZ function 4-24

G

Global variables
  access with VALUE function 4-41
GOTO, abnormal 7-1
greater than operator 2-9
greater than or equal operator 2-9

greater than or less than operator (> <) 2-9
grouping instructions to execute repetitively 3-9
group, DO 3-9

H

HALT condition of SIGNAL and CALL instructions 7-1
halt, trapping 7-1
hexadecimal
  See also conversion
  checking with DATATYPE 4-16
hexadecimal digits 2-3
hexadecimal strings 2-3
host commands 2-17
hours calculated from midnight 4-38

I

IF instruction 3-16
implied semicolons 2-7
imprecise numeric comparison 6-7
inclusive OR operator 2-10
indefinite loops 3-10
indentation during tracing 3-39
indirect evaluation of data 3-17
inequality, testing of 2-9
infinite loops 3-9
initialization
  of arrays 2-16
  of compound variables 2-16
Input and output streams 8-1—8-10
Input from the user 8-1
input to PULL from STDIN 3-29
input to PULL from the keyboard 3-29
Input/Output model 8-1
Input, errors during 8-6
INSERT function 4-24
inserting a string into another 4-24
instructions
  ADDRESS 3-2
  ARG 3-4
  CALL 3-6
  defined 2-13
  DO 3-9
  DROP 3-14
  EXIT 3-15
  IF 3-16
  INTERPRET 3-17
  ITERATE 3-19
  LEAVE 3-20
  NOP 3-21
  NUMERIC 3-22
  OPTIONS 3-24
  PARSE 3-25
  PROCEDURE 3-27
  PULL 3-29
  PUSH 3-30
instructions *(continued)*
  QUEUE 3-31
  RETURN 3-32
  SAY 3-33
  SELECT 3-34
  SIGNAL 3-35
  TRACE 3-37
integer arithmetic 6-1—6-9
integer division
  definition 6-5
  description of 6-1
  operator 2-9
interactive debug 3-37
  See also TRACE instruction
internal functions
  description of 4-2
  return from 3-32
  variables in 3-27
internal routine invoking 3-6
INTERPRET instruction 3-17
interpretive execution of data 3-17
invoking
  built-in functions 3-6
  routines 3-6
ITERATE instruction
  See also DO instruction
  description 3-19
  use of variable on 3-19

K

keyword instructions 3-1
  See also instructions
keywords
  conflict with commands 11-1
  mixed case 3-1
  reservation of 11-1

L

label
  as targets of CALL 3-6
  as targets of SIGNAL 3-35
  description of 2-12
  duplicate 3-35
  in INTERPRET instruction 3-17
  search algorithm 3-35
language processor date and version 3-26
language structure and syntax 2-2
LANGUAGE (local) option of DATE function 4-18
LASTPOS function 4-24
leading blank removal with STRIP function 4-36
leading zeros
  adding with the RIGHT function 4-31
  removal with STRIP function 4-36
LEAVE instruction
  See also DO instruction
  description of 3-20

LEAVE instruction *(continued)*
  use of variable on 3-20
leaving your program 3-15
LEFT function 4-25
LENGTH function 4-25
less than operator 2-9
less than or equal operator 2-9
less than or greater than operator (< >) 2-9
LIFO (last-in/first-out) stacking 3-30
Line input and output 8-1
LINEIN function 4-25
  role in input and output 8-1
LINEIN option of PARSE instruction 3-25
LINEOUT function 4-27
  role in input and output 8-1
lines from a program retrieved with SOURCELINE 4-33
lines from stream 3-25, 4-25
LINES function 4-28
  role in input and output 8-1
lines remaining in stream 4-28
list 2-14
literal string patterns 5-3
literal strings 2-3
logical bit operations
  BITAND 4-9
  BITOR 4-9
  BITXOR 4-10
logical NOT character, OS/2 2-6
logical operations 2-10
logical OR operator, OS2 2-6
looping program
loops
  See also DO instruction
  active 3-19
  execution model 3-12
  modification of 3-19
  repetitive 3-10
  termination of 3-20
lower case symbols 2-4

M

macrospace interfaces 8-10
manipulate external data queue 4-47
MAX function 4-29
messages, error A-1
MIN function 4-29
minutes calculated from midnight 4-39
mixed DBCS string 4-16
Model of input and output 8-1
MONTH option of DATE function 4-18
multi-way call 3-7, 3-36
multiple
multiple assignments 5-6
multiple strings, parsing 5-10
multiplication
  definition 6-4
multiplication *(continued)*
  operator 2-9

N

names
  of functions 4-1
  of programs 3-26
  of subroutines 3-6
  of variables 2-5
negation
  of logical values 2-10
  of numbers 2-9
nesting of control structures 3-8
nibbles 2-4
NOP instruction 3-21
Normal option of DATE function 4-18
not equal operator 2-9
not greater than operator 2-9
not less than operator 2-9
NOT operator 2-6, 2-10
notation
  engineering 6-8
  scientific 6-8
NOTREADY condition
  condition trapping 8-8
  raised by stream errors 8-8
NOVALUE condition
  not raised by VALUE function 4-42
  on SIGNAL instruction 7-2
  use of 11-1
null clauses 2-12
null instruction
  See NOP instruction
null strings 2-3, 2-8
number from stream 4-13
numbers
  arithmetic on 2-9, 6-1, 6-3
  checking with DATATYPE 4-16
  comparison of 2-9, 6-6
  definition 6-2
  description of 2-5, 6-1
  formatting for display 4-23
  in DO instruction 3-9
  truncating 4-41
  use in the language 6-9
NUMERIC
  DIGITS option 3-22
  FORM option 3-22
  FUZZ option 3-22
  instruction 3-22
  settings saved during subroutine calls 3-8
numeric patterns
  See positional patterns

O

operation tracing results 3-37
operator
  arithmetic 2-9, 6-1, 6-3
  as special characters 2-6
  comparison 2-9, 6-6
  concatenation 2-8
  logical 2-10
  precedence (priorities) of 2-11
operators, OS/2 2-6
OPTIONS instruction 3-24
ORDERED option of DATE function 4-18
ORing character strings together 4-9
OR, logical
  exclusive 2-10
  inclusive 2-10
OS2ENVIRONMENT 4-41
OS/2 environment variable 4-41
OS/2 (operating system)
  issuing commands to 3-2
  useful commands 12-1
OTHERWISE clause
  See SELECT instruction
Output to the user 8-1
Output, errors during 8-6
overflow, arithmetic 6-9
OVERLAY function 4-29
overlaying a string onto another 4-29

P

packing a string with X2C 4-45
parentheses
  adjacent to blanks 2-6
  in expressions 2-11
  in function calls 4-1
  in parsing templates 5-7
PARSE instruction 3-25
PARSE LINEIN
  role in input and output 8-1
PARSE PULL
  role in input and output 8-1
parsing 5-1—5-14
  conceptual overview 5-12
  definition 5-1
  equal sign 5-5
  into words 5-1
  multiple assignments 5-6
  multiple strings 5-10
  patterns
    conceptual view 5-13
    positional 5-4
    string 5-3
  period as placeholder 5-2
  positional patterns
    absolute 5-4
    relative 5-5
    variable 5-8
parsing *(continued)*
  selecting words 5-1
  source string 5-1
  string patterns
    literal string patterns 5-3
    variable string patterns 5-7
  summary of instructions 5-8
  template, defined 5-1
  treatment of blanks 5-2
  UPPER, use of 5-8
  variable patterns
    positional 5-8
    string 5-7
  with DBCS characters 5-11
  word parsing
    conceptual view 5-14
    description and examples 5-1
parsing templates
  in ARG instruction 3-4
  in PARSE instruction 3-25
  in PULL instruction 3-29
patterns in parsing
  conceptual view 5-13
  positional 5-4
  string 5-3
period
  causing substitution in variable names 2-14
  in numbers 6-2
period as placeholder in parsing 5-2
permanent command destination change 3-2
Persistent input and output 8-1
POS function 4-30
position
  last occurrence of a string 4-24
positional patterns
  absolute 5-4
  relative 5-5
  variable 5-8
powers of ten in numbers 2-5
precedence of operators 2-11
precision of arithmetic 6-2
prefix operators 2-9, 2-10
presumed command destinations 3-2
PROCEDURE instruction 3-27
programming restrictions 2-1
programs
  arguments to 3-4
  examples 8-8
  retrieving lines with SOURCELINE 4-33
  retrieving name of 3-26
protecting variables 3-27
pseudo random number function of RANDOM 4-30
PULL instruction 3-29
  role in input and output 8-1
PULL option of PARSE instruction 3-26
pure DBCS string 4-16
PUSH instruction 3-30
  role in input and output 8-1

Q

queue
  counting lines in 4-30
  creating and deleting queues 4-47
  detached processes 8-6
  in REXX and OS/2 8-4
  naming and querying 4-47
  private 8-5
  reading from with PULL 3-29
  RXQUEUE function 4-47
  session 8-5
  writing to with PUSH 3-30
  writing to with QUEUE 3-31
QUEUE instruction 3-31
  role in input and output 8-1
queue interface 4-47
QUEUED function 4-30
  role in input and output 8-1

R

RANDOM function 4-30
random number function of RANDOM 4-30
RC (return code)
  not set during interactive debug 10-1
  set by commands 2-17
  special variable 7-5, 11-2
Read position in a stream 8-2
register external function 4-47
relative positional patterns 5-5
remainder
  definition 6-6
  description of 6-1
  operator 2-9
reordering data with TRANSLATE function 4-40
repeating a string with COPIES 4-15
repetitive loops
  altering flow 3-20
  controlled repetitive loops 3-10
  exiting 3-20
  simple do group 3-10
  simple repetitive loops 3-10
reservation of keywords 11-1
restoring variables 3-14
restrictions
  embedded blanks in numbers 2-5
  first character of variable name 2-13
  maximum length of results 2-8
restrictions in programming 2-1
Restructured Extended Executor language (REXX)
RESULT
  return value from a routine 4-5
  set by RETURN instruction 3-7, 3-32
  special variable 11-2
results
  length of 2-8
retrieving argument strings with ARG 3-4
return codes
  as set by commands 2-17
  setting on exit 3-15
RETURN instruction 3-32
return string
  setting on exit 3-15
returning control from REXX program 3-32
REVERSE function 4-31
RexxAddMacro 9-57
RexxClearMacroSpace 9-59
RexxDeregisterExit 9-42
RexxDeregisterFunction 9-23
RexxDeregisterSubcom 9-14
RexxDropMacro 9-58
RexxLoadMacroSpace 9-61
RexxQueryExit 9-43
RexxQueryFunction 9-24
RexxQueryMacro 9-62
RexxQuerySubcom 9-15
RexxRegisterExitDll 9-40
RexxRegisterExitExe 9-41
RexxRegisterFunctionDll 9-20
RexxRegisterFunctionExe 9-22
RexxRegisterSubcomDll 9-12
RexxRegisterSubcomExe 9-13
RexxReorderMacro 9-63
RexxResetTrace 9-55
  description 9-56
  macrospace example 9-65
  macrospace functions 9-57
  RexxAddMacro 9-57
  RexxClearMacroSpace 9-59
  RexxDropMacro 9-58
  RexxLoadMacroSpace 9-61
  RexxQueryMacro 9-62
  RexxReorderMacro 9-63
  RexxSaveMacroSpace 9-60
RexxSaveMacroSpace 9-60
RexxSetHalt 9-53
RexxSetTrace 9-54
RexxStart 9-5
  example using 9-8
  exit example 9-28
  macrospace example 9-65
  RexxStart function 9-5
  using exits 9-7
  using in-storage programs 9-5
  using macrospace programs 9-5
rexxutil functions 4-49
  RxMessage Box 4-49
  SysCls 4-51
  SysCreateObject 4-51
  SysCurPos 4-51
  SysCurState 4-52
  SysDeregisterObjectClass 4-52
  SysDriveInfo 4-53
  SysDriveMap 4-53

rexxutil functions *(continued)*
SysDropFuncs 4-54
SysFileDelete 4-54
SysFileSearch 4-56
SysFileTree 4-55
SysGetEA 4-58
SysGetKey 4-58
SysGetMessage 4-58
SysIni 4-59
SysMkDir 4-61
SysOS2Ver 4-62
SysPutEA 4-62
SysQueryClassList 4-62
SysRegisterObjectClass 4-63
SysRmDir 4-63
SysSearchPath 4-64
SysSetIcon 4-64
SysSleep 4-65
SysTempFileName 4-65
SysTextScreenRead 4-66
SysTextScreenSize 4-66
SysWaitNamedPipe 4-66
RexxVariablePool 9-45
RIGHT function 4-31
rounding
definition 6-3
using a character string as a number 2-5
routines
See functions
See subroutines
running off the end of a program 3-15
RXFUNCADD 4-47
RXFUNCDROP 4-47
RXFUNCQUERY 4-47
RxMessageBox 4-49
RXQUEUE function 4-47
RXSTRING 9-3
definition 9-3
description 9-3
null terminated 9-3
returning 9-3
RXSYSEXIT data structure 9-7
RXTRACE environment variable 10-2

S

SAY instruction 3-33
role in input and output 8-1
scientific notation 6-8
search order
external functions 4-3
for functions 4-2
for subroutines 3-6
seconds calculated from midnight 4-39
SELECT instruction 3-34
semicolons
implied 2-7
omission of 3-1
semicolons *(continued)*
within a clause 2-2
Serial input and output 8-1
SETLOCAL function 4-32
Shift-in (SI) characters B-2, B-6
Shift-out (SO) characters B-2, B-6
SHVBLOCK structure 9-45
SIGL
set by CALL instruction 3-7
set by SIGNAL instruction 3-35
special variable 7-5, 11-2
SIGN function 4-32
SIGNAL
execution of in subroutines 3-8
in INTERPRET instruction 3-17
SIGNAL instruction 3-35
significant digits in arithmetic 6-2
simple symbols 2-14
single stepping
See interactive debug
source of the program and retrieval of information 3-26
SOURCE option of PARSE instruction 3-26
source string 5-1
SOURCELINE function 4-33
SPACE function 4-33
special characters 2-6
special variables
RC 7-5, 11-2
RESULT 3-7, 3-32, 4-5, 11-2
SIGL 3-7, 7-5, 11-2
Standard input and output 8-1
STANDARD option of DATE function 4-18
stem of a variable
assignment to 2-16
description of 2-14
used in DROP instruction 3-14
used in PROCEDURE instruction 3-27
stepping through programs
See interactive debug
storage
Stream
Stream command 4-34, 8-3
Stream errors 8-6
STREAM function 4-33
function overview 8-3
strictly equal operator 2-9
strictly greater than operator 2-9, 2-10
strictly greater than or equal operator 2-10
strictly less than operator 2-9, 2-10
strictly less than or equal operator 2-10
strictly not equal operator 2-9
strictly not greater than operator 2-10
strictly not less than operator 2-10
string
as literal constants 2-3
as names of functions 2-3
as names of subroutines 3-6

string *(continued)*
binary specification of 2-4
comparison of 2-9
concatenation of 2-8
description of 2-3
hexadecimal specification of 2-3
interpretation of 3-17
length of 2-8
null 2-3, 2-8
quotation marks in 2-3
verifying contents of 4-43
string from stream 4-11
string patterns
literal 5-3
variable 5-7
STRIP function 4-36
structure and syntax 2-2
subcommand destinations 3-2
subcommand interfaces 8-10
definition 9-9
description 9-9
registering 9-9
RexxDeregisterSubcom 9-14
RexxQuerySubcom 9-15
RexxRegisterSubcomDll 9-12
RexxRegisterSubcomExe 9-13
subcommand errors 9-9
subcommand failures 9-10
subcommand handler example 9-11
subcommand return code 9-10
subcommands
addressing of 3-2
programming interface
subkeyword 2-13
subroutines
calling of 3-6
forcing built-in or external reference 3-6
naming of 3-6
passing back values from 3-32
return from 3-32
use of labels 3-6
variables in 3-27
subsidiary list 3-14
substitution
in expressions 2-7
in variable names 2-14
SUBSTR function 4-37
subtraction
definition 6-4
operator 2-9
SUBWORD function 4-37
symbol
assigning values to 2-13
classifying 2-14
compound 2-14
constant 2-14
description of 2-4
simple 2-14
symbol *(continued)*
uppercase translation 2-4
use of 2-13
valid names 2-5
SYMBOL function 4-37
syntax checking
See TRACE instruction
SYNTAX condition of SIGNAL instruction 7-2
syntax diagrams 1-2
syntax error
traceback after 3-40
trapping with SIGNAL instruction 7-1
syntax, general 2-2
SysCls 4-51
SysCreateObject 4-51
SysCurPos 4-51
SysCurState 4-52
SysDeregisterObject Class 4-52
SysDriveInfo 4-53
SysDriveMap 4-53
SysDropFuncs 4-54
SysFileDelete 4-54
SysFileSearch 4-56
SysFileTree 4-55
SysGetEA 4-58
SysGetKey 4-58
SysGetMessage 4-58
SysIni 4-59
SysMkDir 4-61
SysOS2Ver 4-62
SysPutEA 4-62
SysQueryClassList 4-62
SysRegisterObjectClass 4-63
SysRmDir 4-63
SysSearchPath 4-64
SysSetIcon 4-64
SysSleep 4-65
system exits
definition 9-26
description 9-26
exit functions 9-40
External function exit 9-30
External HALT exit 9-37
External trace exit 9-38
Host command exit 9-32
Initialization exit 9-39
Queue exit 9-33
registration example 9-41
RexxDeregisterExit 9-42
RexxQueryExit 9-43
RexxRegisterExitDll 9-40
RexxRegisterExitExe 9-41
RXCMD exit 9-32
RXFNC exit
RXCMDHST subfunction 9-32
RXFNCCAL subfunction 9-30
RXINIEXT subfunction 9-39
RXTEREXT subfunction 9-39
RXTRCTST subfunction 9-38

system exits *(continued)*
- RXHLT exit
  - RXHLTCLR subfunction 9-37
  - RXHLTTST subfunction 9-37
- RXINI exit 9-39
- RXMSQ exit
  - RXMSQNAM subfunction 9-34
  - RXMSQPLL subfunction 9-33
  - RXMSQPSH subfunction 9-33
  - RXMSQSIZ subfunction 9-34
- RXSIO exit
  - RXSIODTR subfunction 9-36
  - RXSIOSAY subfunction 9-35
  - RXSIOTRC subfunction 9-35
  - RXSIOTRD subfunction 9-35
- RXSYSEXIT structure 9-27
- RXTER exit 9-39
- RXTRC exit 9-38
- sample exit 9-28
- Session I/O exit 9-35
- Termination exit 9-39

SysTempFileName 4-65
SysTextScreenRead 4-66
SysTextScreenSize 4-66
SysWaitNamedPipe 4-66

T

tail 2-14
templates, parsing
- general description 5-1
- in ARG instruction 3-4
- in PARSE instruction 3-25
- in PULL instruction 3-29

template, definition of 5-1
temporary command destination change 3-2
ten, powers of 6-8
terminals
- reading from with PULL 3-29
- writing to with SAY 3-33

terms and data 2-7
text formatting
- *See* formatting
- *See* word

THEN
- as free standing clause 3-1
- following IF clause 3-16
- following WHEN clause 3-34

TIME function 4-38
TO phrase of DO instruction 3-9
tokenization 2-19
tokens 2-2
TRACE function 4-40
TRACE instruction 3-37
- *See also* interactive debug

TRACE setting
- altering with TRACE function 4-40
- altering with TRACE instruction 3-37

TRACE setting *(continued)*
- querying 4-40

trace tags 3-39
traceback, on syntax error 3-40
tracing
- action saved during subroutine calls 3-8
- by interactive debug 10-1
- data identifiers 3-39
- execution of programs 3-37

tracing flags
- +++ 3-40
- *-* 3-40
- >C> 3-40
- >F> 3-40
- >L> 3-40
- >O> 3-40
- >P> 3-40
- >V> 3-40
- >.> 3-40
- >>> 3-40

trailing blank removed using STRIP function 4-36
trailing zeros 6-3
Transient input and output 8-1
TRANSLATE function 4-40
translation
- *See also* uppercase translation
- with TRANSLATE function 4-40

trap conditions 4-14
trapping of conditions 7-1
TRUNC function 4-41
truncating numbers 4-41
type of data checking with DATATYPE 4-16
Typewriter input and output 8-1
typing data
- *See* SAY instruction

U

unassigning variables 3-14
unconditionally leaving your program 3-15
underflow, arithmetic 6-9
unpacking a string with B2X 4-10
unpacking a string with C2X 4-16
UNTIL phrase of DO instruction 3-9
UPPER option of PARSE instruction 3-25
uppercase translation
- during ARG instruction 3-4
- during PULL instruction 3-29
- of symbols 2-4
- with PARSE UPPER 3-25
- with TRANSLATE function 4-40

USA option of DATE function 4-18
User input and output 8-1—8-10
Utterances 8-1

V

VALUE function 4-41
VALUE option of PARSE instruction 3-26
VAR option of PARSE instruction 3-26
variable names 2-5
variable patterns, parsing with
  positional 5-8
  string 5-7
variable pool interfaces 8-10
  description 9-45
  direct interface 9-50
  dropping variables 9-45
  fetch next variable 9-46
  fetching private information 9-45
  fetching variables 9-45
  restrictions 9-50
  return codes 9-46, 9-49
  returning variable names 9-47
  returning variable values 9-48
  RexxVariablePool example 9-51
  setting variables 9-45
  shared variable request block 9-45
  SHVBLOCK structure 9-45
  symbolic interface 9-50
variable positional patterns 5-8
variable reference 5-7
variable string patterns 5-7
variables
  compound 2-14
  controlling loops 3-10
  description of 2-13
  dropping of 3-14
  exposing to caller 3-27
  external collections 4-41
  getting value with VALUE 4-41
  global 4-41
  in internal functions 3-27
  in subroutines 3-27
  new level of 3-27
  parsing of 3-26
  resetting of 3-14
  setting new value 2-13
  simple 2-14
  special
    RC 7-5, 11-2
    RESULT 3-7, 3-32, 4-5, 11-2
    SIGL 3-7, 7-5, 11-2
  testing for initialization 4-37
  valid names 2-13
VERIFY function 4-43
VERSION option of PARSE instruction 3-26

W

WEEKDAY option of DATE function 4-18
WHEN clause
  See SELECT instruction
WHILE phrase of DO instruction 3-9
whole numbers
  checking with DATATYPE 4-16
  description of 2-5
word
  counting in a string 4-44
  deleting from a string 4-19
  extracting from a string 4-37, 4-43
  finding length of 4-44
  in parsing 5-1
  locating in a string 4-43
WORD function 4-43
word parsing
  conceptual view 5-14
  description and examples 5-1
word processing
  See formatting
  See word
WORDINDEX function 4-43
WORDLENGTH function 4-44
WORDPOS function 4-44
WORDS function 4-44
Write position in a stream 8-3
writing to the stack
  with PUSH 3-30
  with QUEUE 3-31

X

XORing character strings together 4-10
XOR, logical 2-10
XRANGE function 4-45
X2B function 4-45
X2C function 4-45
X2D function 4-46

Z

zeros added on the left 4-31
zeros removal with STRIP function 4-36

Special Characters

. (period)
  as placeholder in parsing 5-2
  causing substitution in variable names 2-14
  in numbers 6-2
< 2-19
< (less than operator) 2-9
< < (strictly less than operator) 2-9, 2-10
< < = (strictly less than or equal operator) 2-10
< > (less than or greater than operator) 2-9
< = (less than or equal operator) 2-9
+ (addition operator) 2-9, 6-3
+ + + tracing flag 3-40
| 2-19
| (inclusive OR operator) 2-10

|| (concatenation operator) 2-8
&& (exclusive OR operator) 2-10
& (AND operator) 2-10
* (multiplication operator) 2-9, 6-3
- tracing flag 3-40
** (power operator) 2-9, 6-5
¬ (NOT operator) 2-10
¬< (not less than operator) 2-9
¬< < (strictly not less than operator) 2-10
¬> (not greater than operator) 2-9
¬> > (strictly not greater than operator) 2-10
¬= (not equal operator) 2-9
¬= = 2-9
¬= = (strictly not equal operator) 2-10
/ (division operator) 2-9, 6-3
/q 2-19
// (remainder operator) 2-9, 6-6
, (comma)
  as continuation character 2-7
  in CALL instruction 3-6
  in function calls 4-1
  separator of arguments 3-6, 4-1
  within a parsing template 5-10
  within a parsing template list 3-4
% (integer division operator) 2-9, 6-5
> 2-19
> (greater than operator) 2-9
>C> tracing flag 3-40
>F> tracing flag 3-40
>L> tracing flag 3-40
>O> tracing flag 3-40
>P> tracing flag 3-40
>V> tracing flag 3-40
>.> tracing flag 3-40
> < (greater than or less than operator) 2-9
> > (strictly greater than operator) 2-9, 2-10
> > > tracing flag 3-40
> > = (strictly greater than or equal operator) 2-10
> = (greater than or equal operator) 2-9
? prefix on TRACE option 3-38
: (colon)
  as a special character 2-6
  in a label 2-12
= (equal sign)
  assignment indicator 2-13
  equal operator 2-9
  immediate debug command 10-1
  in DO instruction 3-9
= = (strictly equal operator) 2-9
— (subtraction operator) 2-9, 6-3
\ (NOT operator) 2-10
\< (not less than operator) 2-10
\< < (strictly not less than operator) 2-10
\> (not greater than operator) 2-10
\> > (strictly not greater than operator) 2-10
\= (not equal operator) 2-10
\= = (strictly not equal operator) 2-10

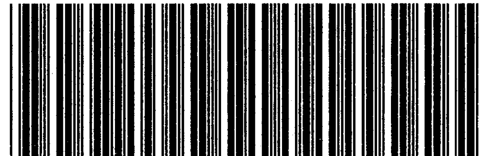
® IBM, OS/2 and Operating System/2 are
registered trademarks of
International Business Machines Corporation

© IBM Corp. 1992

International Business
Machines Corporation

Printed in the
United States of America
All Rights Reserved

10G6268



S10G-6268-00

P10G6268