

AS/400 Advanced Series



REXX/400 Reference

Version 4

AS/400 Advanced Series



REXX/400 Reference

Version 4

Take Note!

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

First Edition (August 1997)

This edition applies to the licensed program Operating System/400 (Program 5769-SS1), Version 4 Release 1 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions.

Make sure that you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch serving your locality. If you live in the United States, Puerto Rico, or Guam, you can order publications through the IBM Software Manufacturing Solutions at 800+879-2755. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication. You can also mail your comments to the following address:

IBM Corporation
Attention Department 542
IDCLERK
3605 Highway 52 N
Rochester, MN 55901-7829 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

If you have access to Internet, you can send your comments electronically to IDCLERK@RCHVMW2.VNET.IBM.COM; IBMMAIL, to [IBMMAIL\(USIB56RZ\)](mailto:IBMMAIL(USIB56RZ)).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

© **Copyright International Business Machines Corporation 1997. All rights reserved.**

Note to U.S. Government Users — Documentation related to restricted rights — Use, duplication or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Contents

Notices	ix
Programming Interface Information	x
Trademarks	x
About REXX/400 Reference	xi
Who Should Read This Book	xi
What You Should Know before Reading This Book	xi
What This Book Contains	xi
Where to Find More Information	xi
Prerequisite and Related Information	xi
Information Available on the World Wide Web	xii
Chapter 1. Introduction	1
Who Should Read This Book	1
What the SAA Solution Is	1
Supported Environments	2
Common Programming Interface	2
How to Use This Book	2
How to Read the Syntax Diagrams	3
For Further REXX Information	4
For Further AS/400 System Information	5
Chapter 2. REXX General Concepts	7
Structure and General Syntax	8
Characters	8
Comments	8
Tokens	9
Implied Semicolons	14
Continuations	14
Expressions and Operators	14
Expressions	15
Operators	15
Parentheses and Operator Precedence	18
Clauses and Instructions	20
Null Clauses	20
Labels	20
Instructions	20
Assignments	20
Keyword Instructions	20
Commands	21
Assignments and Symbols	21
Constant Symbols	22
Simple Symbols	22
Compound Symbols	22
Stems	24
Commands to External Environments	25
Environment	25
Commands	26
REXX and the AS/400 System	26

Chapter 3. Keyword Instructions	29
ADDRESS	30
ARG	33
CALL	35
DO	39
Simple DO Group	39
Repetitive DO Loops	40
Conditional Phrases (WHILE and UNTIL)	42
DROP	44
EXIT	45
IF	46
INTERPRET	47
ITERATE	49
LEAVE	50
NOP	51
NUMERIC	52
OPTIONS	54
PARSE	56
PROCEDURE	59
PULL	62
PUSH	63
QUEUE	64
RETURN	65
SAY	66
SELECT	67
SIGNAL	68
TRACE	70
Alphabetic Character (Word) Options	71
Numeric Options	72
A Typical Example	72
Format of TRACE Output	73
Chapter 4. Functions	75
Syntax	75
Functions and Subroutines	76
Search Order	77
Errors During Execution	77
Built-in Functions	78
ABBREV (Abbreviation)	79
ABS (Absolute Value)	79
ADDRESS	79
ARG (Argument)	80
BITAND (Bit by Bit AND)	81
BITOR (Bit by Bit OR)	81
BITXOR (Bit by Bit Exclusive OR)	82
B2X (Binary to Hexadecimal)	82
CENTER/CENTRE	83
COMPARE	83
CONDITION	84
COPIES	85
C2D (Character to Decimal)	85
C2X (Character to Hexadecimal)	86
DATATYPE	86
DATE	87

DBCS (Double-Byte Character Set Functions)	88
DELSTR (Delete String)	88
DELWORD (Delete Word)	89
DIGITS	89
D2C (Decimal to Character)	89
D2X (Decimal to Hexadecimal)	90
ERRORTXT	91
FORM	91
FORMAT	91
FUZZ	92
INSERT	92
LASTPOS (Last Position)	93
LEFT	93
LENGTH	94
MAX (Maximum)	94
MIN (Minimum)	94
OVERLAY	95
POS (Position)	95
QUEUED	95
RANDOM	96
REVERSE	96
RIGHT	97
SIGN	97
SOURCELINE	97
SPACE	98
STRIP	98
SUBSTR (Substring)	98
SUBWORD	99
SYMBOL	99
TIME	100
TRACE	101
TRANSLATE	102
TRUNC (Truncate)	102
VALUE	103
VERIFY	104
WORD	104
WORDINDEX	105
WORDLENGTH	105
WORDPOS (Word Position)	105
WORDS	106
XRANGE (Hexadecimal Range)	106
X2B (Hexadecimal to Binary)	106
X2C (Hexadecimal to Character)	107
X2D (Hexadecimal to Decimal)	107
AS/400 System-Specific Function	108
SETMSGRC	108
Chapter 5. Parsing	111
Simple Templates for Parsing into Words	111
Templates Containing String Patterns	113
Templates Containing Positional (Numeric) Patterns	114
Parsing with Variable Patterns	118
Using UPPER	118
Parsing Instructions Summary	119

Parsing Instructions Examples	119
Advanced Topics in Parsing	121
Parsing Multiple Strings	121
Combining String and Positional Patterns: A Special Case	121
Parsing with DBCS Characters	122
Details of Steps in Parsing	122
Chapter 6. Numbers and Arithmetic	127
Introduction	127
Definition	128
Numbers	128
Precision	128
Arithmetic Operators	129
Arithmetic Operation Rules—Basic Operators	129
Arithmetic Operation Rules—Additional Operators	131
Numeric Comparisons	133
Exponential Notation	133
Whole Numbers	135
Numbers Used Directly by REXX	135
Errors	136
Chapter 7. Conditions and Condition Traps	137
Action Taken When a Condition Is Not Trapped	138
Action Taken When a Condition Is Trapped	138
Condition Information	140
Descriptive Strings	141
Special Variables	141
The Special Variable RC	141
The Special Variable SIGL	141
Chapter 8. Input and Output Streams	143
The External Data Queue	144
Size Limits	144
Damage Handling	145
CL Queue Commands	145
Chapter 9. AS/400 System Interfaces	147
REXX on the AS/400 System	147
Entering REXX Source Code	147
Starting the REXX Language Processor	148
Terminal Input and Output	149
Pseudo-CL Variables in REXX programs	149
Security	150
Application Interfaces	151
Return Codes and Values	151
Starting the Language Processor from an Application	151
Command Interface	153
Data Types and Structures	155
External Functions and Subroutines	156
System Exit Interfaces	157
System Exits and the Variable Pool	158
System Exit Functions and Subfunctions	158
Entry Conditions	159
Exit Conditions	160

Exit Definitions	160
The QREXVAR Interface	166
Shared-Variable Request Block	167
Queuing Interfaces	170
Queue Application Programming Interface	170
Chapter 10. Debug Aids	173
Interactive Debugging of Programs	173
Chapter 11. Reserved Keywords and Special Variables	175
Reserved Keywords	175
Special Variables	176
Appendix A. Double-Byte Character Set (DBCS) Support	177
General Description	177
Enabling DBCS Data Operations	178
Symbols and Strings	178
Validation	179
Instruction Examples	180
DBCS Function Handling	181
Built-in Function Examples	183
DBCS Processing Functions	186
Counting Option	186
Function Descriptions	187
DBADJUST	187
DBBRACKET	187
DBCENTER	187
DBLEFT	188
DBRIGHT	188
DBRLEFT	189
DBRRIGHT	189
DBTODBCS	190
DBTOSBCS	190
DBUNBRACKET	190
DBVALIDATE	190
DBWIDTH	191
Appendix B. REXX/400 Implementation Limits	193
Index	195

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 that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

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 Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the software interoperability coordinator. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Address your questions to:

IBM Corporation
Software Interoperability Coordinator
3605 Highway 52 N
Rochester, MN 55901-7829 USA

This publication could contain technical inaccuracies or typographical errors.

This publication may refer to products that are announced but not currently available in your country. This publication may also refer to products that have not been announced in your country. IBM makes no commitment to make available any unannounced products referred to herein. The final decision to announce any product is based on IBM's business and technical judgment.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This publication contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

Programming Interface Information

This *REXX/400 Reference* is intended to help you write programs using the AS/400 REXX interpreter. This book documents General-Use Programming Interface and Associated Guidance Information provided by AS/400.

General-Use programming interfaces allow you to write programs that obtain the services of AS/400.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Application System/400	Operating System/2
AS/400	Operating System/400
BookManager	OS/2
CICS	OS/400
Common User Access	PrintManager
DB2/400	SAA
IBMLink	SQL/400
IMS	System/36
Integrated Language Environment	System/38
Library Reader	Systems Application Architecture

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About REXX/400 Reference

This is a reference book containing all of the IBM REXX for AS/400 (also known as REXX/400) instructions and functions. They are listed alphabetically in their own sections. Also included are details about general concepts you need to know in order to program in REXX. You will need a terminal with access to AS/400 and you should be reasonably familiar with AS/400, but you need not have had any previous programming experience.

The programming language described by this book is called the REstructured eXtended eXecutor language (abbreviated REXX.) The book also describes how the AS/400 REXX/400 language processor (shortened, hereafter, to the language processor) processes or *interprets* the REstructured eXtended eXecutor language.

Who Should Read This Book

This book is for users who need to refer to REXX/400 instructions and functions, and for those who need to learn more details about items such as parsing.

What You Should Know before Reading This Book

You should have read the *REXX/400 Programmer's Guide*, SC41-5728, to learn how to program in REXX.

What This Book Contains

This book contains details about all of the REXX/400 instructions and functions, as well as information about general concepts, parsing, math functions, condition trapping, system interfaces, and debug aids.

Where to Find More Information

You can find more information about AS/400 and REXX in the publications listed in "For Further REXX Information" on page 4.

Prerequisite and Related Information

For information about other AS/400 publications (except Advanced 36), see either of the following:

- The *Publications Reference* book, SC41-5003, in the AS/400 Softcopy Library.
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database that contains descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with the OS/400 operating system at no charge.

Information Available on the World Wide Web

More AS/400 information is available on the World Wide Web. You can access this information from the AS/400 home page, which is at the following uniform resource locator (URL) address:

<http://www.as400.ibm.com>

Select the Information Desk, and you will be able to access a variety of AS/400 information topics from that page.

Chapter 1. Introduction

This introductory section:

- Identifies the book's purpose and audience
- Gives a brief overview of the Systems Application Architecture (SAA) solution
- Explains how to use the book.

Who Should Read This Book

This book describes the IBM REXX for AS/400 Interpreter (hereafter referred to as the interpreter or language processor) and the REstructured eXtended eXecutor (called 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).

The REXX/400 language processor is an interpreter, that is, a set of programs which read REXX program source, interpret it, and take the necessary action. Descriptions include the use and syntax of the language and explain how the language processor “interprets” the language as a program is running.

The REXX language is specifically designed to manipulate character strings. It is most often used to build command strings and pass them to the current command environment. On the AS/400 system this enables both users and programs to expand the uses of the AS/400 system's control language (CL) by adding such structured programming language capabilities as loops, branches, subroutine and function calls, and condition trapping. In effect, a REXX program acts as a surrogate for the user of the command line interface, supplying commands and responding to the resulting output.

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.

If you are using REXX only on an AS/400 system, this has no effect on your programs. If you plan to run your programs in other computing environments, however, some restrictions may apply. We suggest that you consult the *SAA Common Programming Interface REXX 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 manages 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).

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
 - REXX (formerly called Procedures Language)
 - RPG
- Services
 - Communications
 - Database
 - Dialog
 - Language Environment
 - 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 Use This Book

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

1. Chapter 1, “Introduction”

2. Chapter 2, “REXX General Concepts”
3. Chapter 3, “Keyword Instructions” (in alphabetic order)
4. Chapter 4, “Functions” (in alphabetic order)
5. Chapter 5, “Parsing” (a method of dividing character strings, such as commands)
6. Chapter 6, “Numbers and Arithmetic”
7. Chapter 7, “Conditions and Condition Traps”
8. Chapter 8, “Input and Output Streams”
9. Chapter 9, “AS/400 System Interfaces”
10. Chapter 10, “Debug Aids”
11. Chapter 11, “Reserved Keywords and Special Variables.”

There are also appendixes covering:

- Appendix A, “Double-Byte Character Set (DBCS) Support”
- Appendix B, “REXX/400 Implementation Limits.”

How to Read the Syntax Diagrams

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

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

The ► symbol indicates the beginning of a statement.

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

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

The →◄ symbol indicates the end of a statement.

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

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

►—STATEMENT—*required_item*—→◄

- Optional items appear below the main path.

►—STATEMENT—→◄
└—*optional_item*—┘

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

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

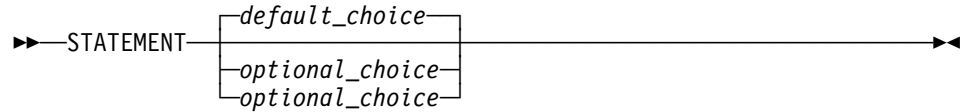
►—STATEMENT—*required_choice1*—→◄
└—*required_choice2*—┘

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

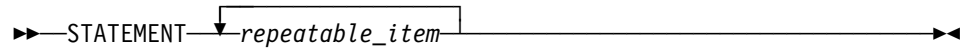
►—STATEMENT—→◄
└—*optional_choice1*—┘
└—*optional_choice2*—┘

- If one of the items is the default, it appears above the main path and the remaining choices are shown below.

Introduction

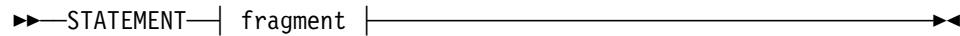


- An arrow returning to the left above the main line indicates an item that can be repeated.

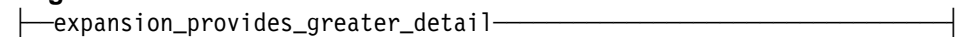


A repeat arrow above a stack indicates that you can repeat the items in the stack.

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

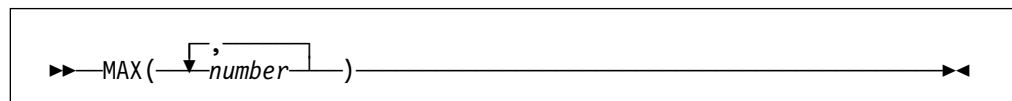


fragment:



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

The following example shows how the syntax is described:



For Further REXX Information

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

- The *Programming: REXX/400 Programmer's Guide*, SC24-5553, provides both a general introduction to REXX programming and a source of applications examples for experienced programmers.
- The *SAA Common Programming Interface REXX Level 2 Reference*, SC24-5549, may be useful to more experienced REXX users who may wish to code portable programs. This book defines SAA REXX. 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.

For Further AS/400 System Information

Here is a list of books that you may wish to include in your AS/400 system library:

- *Publications Reference*, SC41-5003, provides information on all manuals in the AS/400 Softcopy library.
- *AS/400 Information Directory*, provides a unique, multimedia interface to a searchable database containing descriptions of titles available from IBM or from selected other publishers. This is shipped with your system at no charge.
- *ADTS/400: Source Entry Utility*, SC09-1774, provides information about using the Application Development Tools source entry utility (SEU) to create and edit source members. The manual explains how to start and end an SEU session and how to use the many features of this full-screen text editor. The manual contains examples to help both new and experienced users accomplish various editing tasks, from the simplest line commands to using pre-defined prompts for high-level languages and data formats.
- *ILE Application Development Example*, SC41-5602, provides the information needed to understand the relationships of some of the various programming tools and utilities on the AS/400 system that can be used to produce an application. It presents a hypothetical mailing list application that illustrates how the user can design and run applications on the AS/400 system.
- *Backup and Recovery*, SC41-5304, provides information about the different media available to save and protect system data, as well as a description of how to record changes made to database files and how that information can be used for system recovery and activity report information. It also provides information about how to install the system.
- *CL Programming*, SC41-5721, provides a wide-ranging discussion of AS/400 programming topics, including the following:
 - A general discussion of objects and libraries
 - Control language (CL) programming, controlling flow and communicating between programs, working with objects in CL programs, and creating CL programs
 - Predefined and impromptu messages and message handling
 - How to define and create user-defined commands and menus
 - Application testing, including debug mode, breakpoints, traces, and display functions.
- *CL Reference*, SC41-5722, provides a reference for using CL commands to request functions of the operating system (OS/400) and functions provided by the various languages and utilities. Also included is information for creating new CL commands or for changing (or renaming) existing CL commands.
- *Data Management*, SC41-5710, provides information about using files in applications programs. A file is the OS/400 object type that provides storage of and access to data in the database, or devices such as display stations and printers, or on another system. This manual includes information on the following topics:
 - Fundamental structure and concepts of data management support on the system

Introduction

- Data management support for display stations, printers, tapes, and diskettes, as well as spooling support
 - Overrides and file redirection (temporarily making changes to files when an application program is run)
 - Copying files by using system commands to copy data from one place to another
 - Tailoring a system using double-byte data.
- *DB2 for AS/400 Database Programming*, SC41-5701, provides a detailed discussion of the AS/400 database organization, including information on how to create, describe, and manipulate database files on the system.
 - *Security – Reference*, SC41-5302, provides information about system security concepts, planning for security, and setting up security on the system.
 - *Work Management*, SC41-5306, provides information about how to create a work management environment and how to change it.
 - *AS/400 Advanced Series Handbook*, GA19-5486, acquaints the user with the features and capabilities of the AS/400 system. It familiarizes the user with characteristics of the system and the various licensed programs used on it.
 - *System Operation*, SC41-4203, provides information about how to use the system unit operator display, send and receive messages, respond to error messages, start and stop the system, use control devices, work with program temporary fixes (PTF) and process and manage jobs on the system.
 - *SAA Common Programming Interface Communications Reference*, SC26-4399, assists you in programming with the CPI-Communications interface. It contains general-use programming interfaces which let you write programs that use the services of CPI-Communications.
 - *National Language Support*, SC41-5101, assists you in evaluating, planning, and using the AS/400 National Language Support (NLS) and multilingual capabilities.
 - *DB2 for AS/400 SQL Programming*, SC41-5611, provides information about the EXECSQL environment, including how to pass data with REXX variables, and how to create a REXX program that will execute without the DB2/400 Query Management and SQL Development Kit Version 3 LPP, 5763-ST1, installed.
 - *DB2 for AS/400 SQL Reference*, SC41-5612, provides information about SQL/400 statements and their parameters. It also includes an appendix describing the SQL communications area (SQLCA) and SQL description area (SQLDA).

Chapter 2. REXX General Concepts

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

- Command procedures
- Application programs
- Prototyping
- Personal computing.

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

REXX is specifically designed for the manipulation of character strings. It is most often used to build command strings and pass them to the current command environment, which on the AS/400 system defaults to the OS/400 control language (CL). For more information, see the *CL Reference*. This enables both users and programs to expand CL by using structured programming features such as loops, branches, subroutine and function calls, and condition trapping.

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

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

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

The limit on the length of symbols (variable names) is 250 characters. You can use characters from a national language character set. See *National Language Support* for more information.

You can use compound symbols, such as

NAME.Y.Z

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

REXX programs are processed by a built-in language processor. The source code of a REXX program is processed directly, rather than being first compiled into a separate program. Because of this, you do not have to wait for the program to compile and if the program fails because of a syntax error, the point of error is clearly indicated. Usually, it will not take long to understand the difficulty and make a correction.

Structure and General Syntax

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” on page 9)
- Zero or more blanks (again ignored)
- A semicolon (;) delimiter that may be implied by line-end, certain keywords, or the colon (:).

Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to operator characters and special characters (see page 13) are also removed.

Characters

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

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

- Some code pages do not contain all characters that REXX defines as valid (for example, ¬, the logical NOT character).
- Some characters that REXX defines as valid have different encodings in different code pages (for example, !, the exclamation point).

For information about Double-Byte Character Set characters, see Appendix A, “Double-Byte Character Set (DBCS) Support” on page 177.

Comments

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

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

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

```

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

```

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

```

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

```

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

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

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

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

```

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

```

For information about Double-Byte Character Set characters, see Appendix A, "Double-Byte Character Set (DBCS) Support" on page 177 and the `OPTIONS` instruction on page 54.

Tokens

A token is the unit of low-level syntax from which clauses are built. Programs written in REXX are composed of tokens. Tokens are character strings of any length, up to a maximum according to individual class. They are separated by blanks or comments or by the nature of the tokens themselves. The classes of tokens are:

Literal Strings:

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

Note that a string followed immediately by a (is considered to be the name of a function. If followed immediately by the symbol X or x, it is considered to be a hexadecimal string. If followed immediately by the symbol B or b, it is considered to be a binary string. Descriptions of these forms follow.

Hexadecimal Strings:

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

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

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

Note: A hexadecimal string is *not* a representation of a number. Rather, it is an escape mechanism that lets a user describe a character in terms of its encoding (and, therefore, is machine-dependent). In EBCDIC, '40'X is the encoding for a blank. In every case, a string of the form '.....'x is simply an alternative to a straightforward string. In EBCDIC 'C1'x and 'A' are identical, as are '40'x and a blank, and must be treated identically.

Implementation maximum: The packed length of a hexadecimal string may not exceed 250 bytes, or 500 hexadecimal digits.

Binary Strings:

A binary string is a literal string, expressed using a binary representation of its encoding. It is any sequence of zero or more binary digits (0 or 1) in groups of 8 (bytes) or 4 (nibbles). The first group may have fewer than four digits; in this case, up to three 0 digits are assumed to the left of the first digit, making a total of four digits. The groups of digits are optionally separated by one or more blanks, and the whole sequence is delimited by matching single or double quotation marks and immediately

followed by the symbol b or B. (Neither b nor B can be part of a longer symbol.) The blanks, which may be present only at byte or nibble boundaries (and not at the beginning or end of the string), are to aid readability. The language processor ignores them.

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

These are valid binary strings:

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

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

Symbols:

Symbols are groups of characters, selected from the:

- English alphabetic characters (A–Z and a–z¹)
- Numeric characters (0–9)
- Characters . !² ? and underscore (_)
- National Language Character Set (NLCS) characters
- Double-Byte Character Set (DBCS) characters, where each DBCS character consists of 2 bytes. All double-byte characters represented by codes within the following ranges are considered to be DBCS characters:

Table 1. DBCS Ranges

Byte	EBCDIC
1st	X'41' to X'FE'
2nd	X'41' to X'FE'
DBCS blank	X'4040'

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

¹ Note that some code pages do not include lowercase English characters a–z.

² The encoding of the exclamation point character depends on the code page in use.

These are valid symbols:

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

For information about National Language Character Set (NLCS) characters, see *National Language Support*.

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.

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

These are valid numbers in exponential notation:

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

Implementation maximum: A symbol can consist of up to 250 characters. (But note that its value, if it is a variable, is limited only by the amount of storage available.)

Numbers:

These are character strings consisting of one or more decimal digits, with an optional prefix of a plus or minus sign, and optionally including a single period (.) that represents a decimal point. A number can also have a power of 10 suffixed in conventional exponential notation: an E (uppercase or lowercase), followed optionally by a plus or minus sign, then followed by one or more decimal digits defining the power of 10. Whenever a character string is used as a number, rounding may occur to a precision specified by the NUMERIC DIGITS instruction (default nine digits). See pages 127-136 for a full definition of numbers.

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

These are valid numbers:

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

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

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

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

Operator Characters:

The characters: `+ - \ / % * | & = ~ > <` and the sequences `>= <= \> \< \= >< <> == \== // && || ** ~> ~< ~= ~== >> << >>= \<< ~<< \>> ~>> <=<` indicate operations (see page 15). A few of these are also used in parsing templates, and the equal sign is also used to indicate assignment. Blanks adjacent to operator characters are removed. Therefore, the following are identical in meaning:

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

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

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

Special Characters:

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

```
, ; : ) (
```

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

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

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

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

between the 'REPEAT' and the A remains as an operator. Thus, this clause is treated as though written:

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

Implied Semicolons

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

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

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

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

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

Continuations

One way to continue a clause onto the next line is to use the comma, which is referred to as the **continuation character**. The comma is functionally replaced by a blank, and, thus, no semicolon is implied. One or more comments can follow the continuation character before the end of the line.

The 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

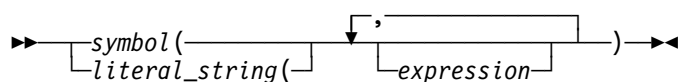
Expressions in REXX are a general mechanism for combining one or more pieces of data in various ways to produce a result, usually different from the original data.

Expressions

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

Terms include:

- **Literal Strings** (delimited by quotation marks), which are constants
- **Symbols** (no quotation marks), which are translated to uppercase. A symbol that does not begin with a digit or a period may be the name of a variable; in this case the value of that variable is used. Otherwise a symbol is treated as a constant string. A symbol can also be **compound**.
- **Function calls** (see page 75), 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 18). Expressions are wholly evaluated, unless an error occurs during evaluation.

All data is in the form of “typeless” character strings (typeless because it is not—as in some other languages—of a particular declared type, such as Binary, Hexadecimal, Array, and so forth, or of a declared length). Consequently, the result of evaluating any expression is itself a character string. Terms and results (except arithmetic and logical expressions) may be the **null string** (a string of length 0). Note that REXX imposes no restriction on the maximum length of results. However, there is usually some practical limitation dependent upon the amount of storage available to the language processor.

Implementation maximum: The length of the evaluated result of an expression is limited by the available storage.

Operators

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

- Concatenation
- Arithmetic
- Comparison
- Logical.

String Concatenation

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

(blank) Concatenate terms with one blank in between

|| Concatenate without an intervening blank

(abuttal) Concatenate without an intervening blank

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

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

Examples:

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

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

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

```
'c1 c2'x'CDE'
```

evaluate to ABCDE.

In the case of:

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

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

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

results in an abuttal, and evaluates to 37.40.

Arithmetic

You can combine character strings that are valid numbers (see page 12) using the arithmetic operators:

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

See Chapter 6, “Numbers and Arithmetic” on page 127 for details about precision, the format of valid numbers, and the operation rules for arithmetic. Note that if an

arithmetic result is shown in exponential notation, it is likely that rounding has occurred.

Comparison

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

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

For all the other comparison operators, if *both* terms involved are numeric, a numeric comparison (in which leading zeros are ignored, and so forth—see “Numeric Comparisons” on page 133) 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, which the AS/400 system is, lowercase alphabets precede uppercase, and the digits 0–9 are higher than all alphabets.

The comparison operators and operations are:

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

REXX General Concepts

<code>\<<, -<<</code>	Strictly NOT less than
<code><<=</code>	Strictly less than or equal to
<code>\>>, ->></code>	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 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:

<code>&</code>	AND Returns 1 if both terms are true.
<code> </code>	Inclusive OR Returns 1 if either term is true.
<code>&&</code>	Exclusive OR Returns 1 if either (but not both) is true.
Prefix <code>\,~</code>	Logical NOT Negates; 1 becomes 0, and 0 becomes 1.

Parentheses and Operator Precedence

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

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

`term1 operator1 term2 operator2 term3`

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

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

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

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

<code>+ - ~ \</code>	(prefix operators)
<code>**</code>	(power)
<code>* / % //</code>	(multiply and divide)

+ - (add and subtract)
 (blank) || (abuttal) (concatenation with or without blank)
 = > < (comparison operators)
 == >> <<
 != -=
 >< <>
 > <>
 < <>
 == -=
 >> ->>
 << -<<
 >= >>=
 <= <<=

 & (and)
 | && (or, exclusive or)

Examples:

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

```

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

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 PL/I). 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. More than one label may precede any instruction. Labels are treated as null clauses and can be traced selectively to aid debugging.

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

You can use characters from a national language character set. See *National Language Support*, SC41-5101, for more information.

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

A single clause of the form *symbol=expression* is an instruction known as an **assignment**. An assignment gives a variable a (new) value. See "Assignments and Symbols" on page 21.

Keyword Instructions

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

```
DO
  instruction
  instruction
  instruction
END
```

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

Commands

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

Assignments and Symbols

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

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

symbol=expression;

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

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” on page 22), its value is the derived name of the symbol.

Example:

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

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

Constant Symbols

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

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

These are constant symbols:

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

Simple Symbols

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

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

These are simple symbols:

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

Compound Symbols

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

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

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

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

These are compound symbols:

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

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

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

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

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

is given by

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

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

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

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

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

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

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

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.  
Ö.  
<.A.B>.
```

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

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

For example:

```
hole. = "empty"  
hole.9 = "full"  
  
say hole.1 hole.mouse hole.9  
  
/* says "empty empty full" */
```

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

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

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

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

You can manipulate collections of variables, referred to by their stem, with the DROP and PROCEDURE instructions. DROP FRED. drops all variables with that stem (see page 44), and PROCEDURE EXPOSE FRED. exposes *all possible* variables with that stem (see page 59).

Notes:

1. When the ARG, PARSE, or PULL instruction or the VALUE built-in function or the variable pool interface changes a variable, the effect is identical with an assignment. Anywhere a value can be assigned, using a stem sets an entire collection of variables.
2. Because an expression can include the operator =, and an instruction may consist purely of an expression (see “Commands to External Environments” on page 25), a possible ambiguity is resolved by the following rule: any clause that starts with a symbol and whose second token is (or starts with) an equal sign (=) is an **assignment**, rather than an expression (or a keyword instruction). This is not a restriction, because you can ensure the clause is processed as a command in several ways, such as by putting a null string before the first name, or by enclosing the first part of the expression in parentheses.

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

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

is an assignment, not an ADDRESS instruction.

3. You can use the SYMBOL function (see page 99) to test whether a symbol has been assigned a value. In addition, you can set SIGNAL ON NOVALUE to trap the use of any uninitialized variables (except when they are tails in compound variables—see page 138).

Commands to External Environments

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

Environment

The SAA environments listed on page 2 are *programming* environments. Three *operating* environments exist on the AS/400 system:

- System/36 environment
- System/38 environment
- Non-System/36 or System/38 environment.

REXX interacts with **command environments** available in the operating environment. The command environment can be system-defined or user-defined.

The command environment is another language, such as CL, to which REXX passes what it interprets as a command and from which REXX waits for a response to that command. The environment is selected when REXX is run. See “Starting the REXX Language Processor” on page 148. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function. The underlying operating system defines environments external to the REXX program.

Commands

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

```
expression;
```

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

The environment then processes the command, which may have side-effects. It eventually returns control to the language processor, after setting a return code. A **return code** is a string, typically a number, that returns some information about the command that has been processed. A return code usually indicates if a command was successful or not but can also represent other information. The language processor places this return code in the REXX special variable RC. See “Special Variables” on page 141.

In addition to setting a return code, the underlying system may also indicate to the language processor if an error or failure occurred. An **error** is a condition raised by a command for which a program that uses that command would usually be expected to be prepared. (For example, a Delete Program (DLTPGM) command to the operating system might report Object PGM in MYLIB type *PGM not found, as an error.) A **failure** is a condition raised by a command for which a program that uses that command would *not* usually be expected to recover (for example, a command that is not executable or cannot be found).

Errors and failures in commands can affect REXX processing if a condition trap for ERROR or FAILURE is ON (see Chapter 7, “Conditions and Condition Traps” on page 137). 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 70.

Here is an example of submitting a command. The default command environment is being changed to a new environment.

```
SAY ADDRESS          /* Shows the default command environment "COMMAND" */  
ADDRESS 'MYLIB/APP1'  
SAY ADDRESS()       /* Shows the new environment "MYLIB/APP1"          */
```

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.

REXX and the AS/400 System

REXX/400, REXX on the AS/400 system, is part of Operating System/400 (OS/400). In most cases, you can run a REXX/400 program from any place you can enter a CL command.

A REXX/400 program can be run by using:

- The Start REXX Procedure (STRREXPRC) command
- A user-defined command with a REXX command processing program.

A REXX/400 program itself can contain CL commands as well as REXX instructions. See the *CL Reference*, SC41-5722, and “Starting the REXX Language Processor” on page 148 for more information.

REXX/400 conforms to the security characteristics of the AS/400 system. In general, this means that REXX/400 uses the security of the AS/400 system and does not add any of its own. See “Security” on page 150 for more information.

Experienced AS/400 system programmers will notice some of the ways REXX/400 differs from other programming languages on the AS/400 system. For example:

- REXX/400 programs are source members. REXX/400 programs are not compiled, but interpreted from the source.
- Variables are not declared in a REXX/400 program. Rather, they are assigned as needed. See “Assignments” on page 20.
- REXX/400 automatically monitors exception messages from commands and indicates their occurrence to the REXX program through the special variable RC or by trapping ERROR and FAILURE conditions. See “Special Variables” on page 176 and Chapter 7, “Conditions and Condition Traps” on page 137 for further information.

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

As explained in “Keyword Instructions” on page 20, a keyword instruction is recognized *only* if its keyword is the first token in a clause, and if the second token does not start with an `=` character (implying an assignment) or a colon (implying a label). The keywords ELSE, END, OTHERWISE, THEN, and WHEN are recognized in the same situation. Note that any clause that starts with a keyword defined by REXX cannot be a command. Therefore,

```
arg(fred) rest
```

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

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

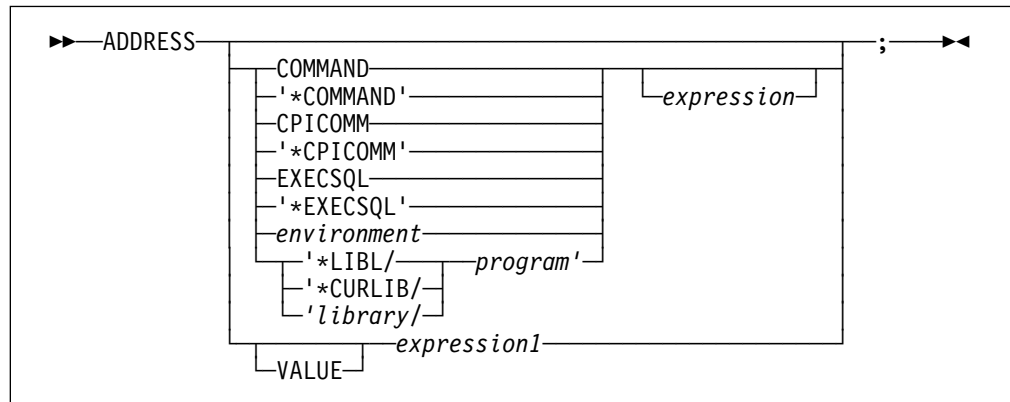
Blanks adjacent to keywords have no effect other than to separate the keyword from the subsequent token. One or more blanks following VALUE are required to separate the *expression* from the subkeyword in the example following:

```
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. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction.

Working with command environments is described on page 153.

To send a single command to a specified environment, code an *environment*, a literal string or a single symbol, which is taken to be a constant, followed by an *expression*. (The environment name is the name of a program object, for example *LIBL/MYPROG, that can process commands.) The *expression* is evaluated, and the resulting string is routed to the *environment* to be processed as a command. (Enclose in quotation marks any part of the expression you do not want to be evaluated.) After execution of the command, *environment* is set back to whatever it was before, thus temporarily changing the destination for a single command. The special variable RC is set, just as it would be for other commands. (See page 26.) Errors and failures in commands processed in this way are trapped or traced as usual.

Examples:

```

ADDRESS COMMAND DSPCURLIB
ADDRESS CPICOMM 'CINIT parm0 parm1 parm2 ... parmN'
ADDRESS EXECSQL 'INSERT INTO DB/TABLE1 VALUES(789)'
ADDRESS '*CURLIB/MYPROG' COMMAND_EXPRESSION
ADDRESS "MYLIB/MYNEWPROG" COMMAND_EXPRESSION

```

In the second example, this command is routed back to the CPICOMM environment, which is the common programming interface communications environment, the communications element of the SAA Common Programming Interface. For more information on CPICOMM, see the *SAA Common Programming Interface Communications Reference*.

In the third example, this command is routed back to the EXECSQL environment, which is the Structured Query Language (SQL) environment, the standard database interface language used by DB2/400. For more information on SQL statements, see the *DB2 for AS/400 SQL Reference*. or for more information on the EXECSQL environment, see the see the *DB2 for AS/400 SQL Programming*.

Note: To use the EXEC SQL environment on a system that does not have the DB2/400 Query Management and SQL Development Kit Version 2 LPP, 5763-ST1, installed, see the *DB2 for AS/400 SQL Programming* for special instructions for handling the REXX program.

If you specify only *environment*, a lasting change of destination occurs: all commands that follow are routed to the specified command environment, until the next ADDRESS instruction is processed. The previously selected environment is saved.

Examples:

```
ADDRESS COMMAND
ADDRESS FRED
ADDRESS '*CURLIB/ETHEL'
```

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

Example:

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

With no arguments, commands are routed back to the environment that was selected before the previous lasting change of environment was made, and the current environment name is saved. After changing the environment, repeated execution of ADDRESS alone, therefore, switches the command destination between two environments alternately. **A null string for the environment name ("") is the same as the default environment COMMAND.**

The two environment names are automatically saved across internal and external subroutine and function calls. See the CALL instruction (page 35) for more details.

The address setting is the currently selected environment name. You can retrieve the current address setting by using the ADDRESS built-in function (see page 79).

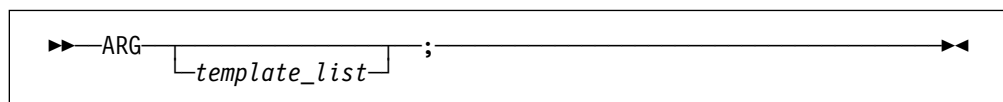
Notes:

1. *LIBL is the current library list for your job.
2. *CURLIB is the current library for your job.
3. The ADDRESS instruction can be used with either system-defined or user-defined environments. **The default system environment is COMMAND, the CL command environment.** A user environment is identified by the name of the program that is to be called. A program name, qualified or unqualified, can be specified. If an unqualified name is specified, the library list (*LIBL) will be searched to resolve to the program.
4. Whenever you specify a library, you must use the slash (/) separator between the library name and the environment name. Whenever you use the slash, you must enclose the full environment name in quotation marks.
5. Executing ADDRESS '*LIBL/MYPROG' will locate and hold the information associated with that specific library and program. The command environment

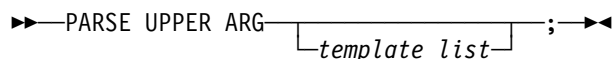
ADDRESS

will not change if you subsequently change the library list. If you wish to have MYPROG found with the new library list, you must enter ADDRESS '*LIBL/MYPROG' again.

ARG



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



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

Unless a subroutine or internal function is being processed, the strings passed as parameters to the program are parsed into variables according to the rules described in the section on parsing (page 111).

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

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

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

Example:

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

```
Arg adjective noun .
```

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

```
/* NOUN contains 'RIDER' */
```

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

Example:

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

```
Fred: Arg string, num1, num2
```

```
/* Now: STRING contains 'DATA X' */
```

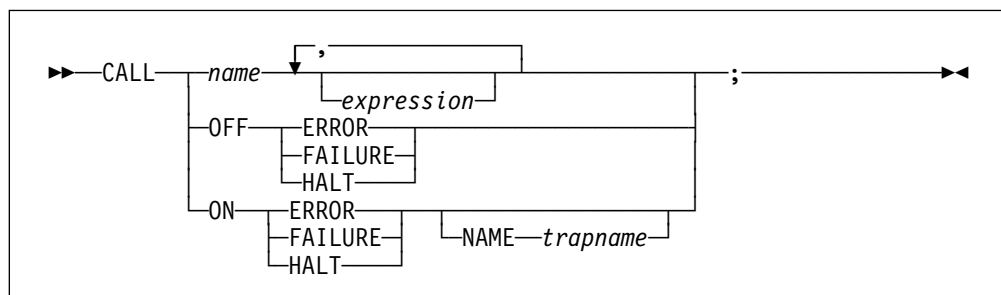
```
/* NUM1 contains '1' */
```

```
/* NUM2 contains '5' */
```

Notes:

1. The ARG built-in function can also retrieve or check the argument strings to a REXX program or internal routine. See page 80.
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 57 for details.

CALL



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

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, “Conditions and Condition Traps” on page 137.

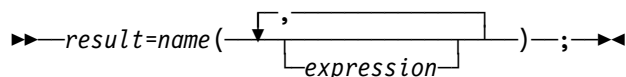
Note: The following discussion of the CALL instruction provides information on the CALL routine form of the CALL instruction.

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

- An internal routine** A function or subroutine that is in the same program as the CALL instruction or function call that calls it.
- A built-in routine** A function (which may be called as a subroutine) that is defined as part of the REXX language.
- An external routine** A function or subroutine that is neither built-in nor in the same program as the CALL instruction or function call that calls it.

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

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



If the called routine does not return a result, then you will get an error if you call it as a function (as previously shown).

REXX/400 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 instruction or ARG built-in function in the called routine accesses these strings rather than any

previously active in the calling program, until control returns to the CALL instruction. You can omit expressions, if appropriate, by including extra commas.

The CALL then causes a branch to the routine called *name*, using exactly the same mechanism as function calls. (See Chapter 4, “Functions” on page 75.) The search order is in the section on functions (see “Search Order” on page 77) but briefly is as follows:

Internal routines:

These are sequences of instructions inside the same program, starting at the label that matches *name* in the CALL instruction. If you specify the routine name in quotation marks, then an internal routine is not considered for that search order. You can use SIGNAL and CALL together to call an internal routine whose name is determined at the time of execution; this is known as a multi-way call (see page 69). The RETURN instruction completes the execution of an internal routine.

Built-in routines:

These are routines built into the language processor for providing various functions. They always return a string that is the result of the routine. (See page 78.)

External routines:

Users can write or use routines that are external to the language processor and the calling program. You can code an external routine in REXX or in any language that supports the system-dependent interfaces. For more information on these interfaces, see “Application Interfaces” on page 151. If the CALL instruction calls an external routine written in REXX as a subroutine, you can retrieve any argument strings with the ARG or PARSE ARG instructions or the ARG built-in function.

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

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

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

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

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

Example:

```

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

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

```

During internal subroutine (and function) execution, all important pieces of information are automatically saved and 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 ended. (But those currently active within the subroutine are ended.)
- **Trace action:** After a subroutine is debugged, you can insert a TRACE Off at the beginning of it, and this does not affect the tracing of the caller. Conversely, if you simply wish to debug a subroutine, you can insert a TRACE Results at the start and tracing is automatically restored to the conditions at entry (for example, Off) upon return. Similarly, ? (interactive debug) is saved across routines.
- **NUMERIC settings:** The DIGITS, FUZZ, and FORM of arithmetic operations (in “NUMERIC” on page 52) 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 30) 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 information describes the state and origin of the current trapped condition. The CONDITION built-in function returns this information. See “CONDITION” on page 84.
- **Elapsed-time clocks:** A subroutine inherits the elapsed-time clock from its caller (see “TIME” on page 100), 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 54.

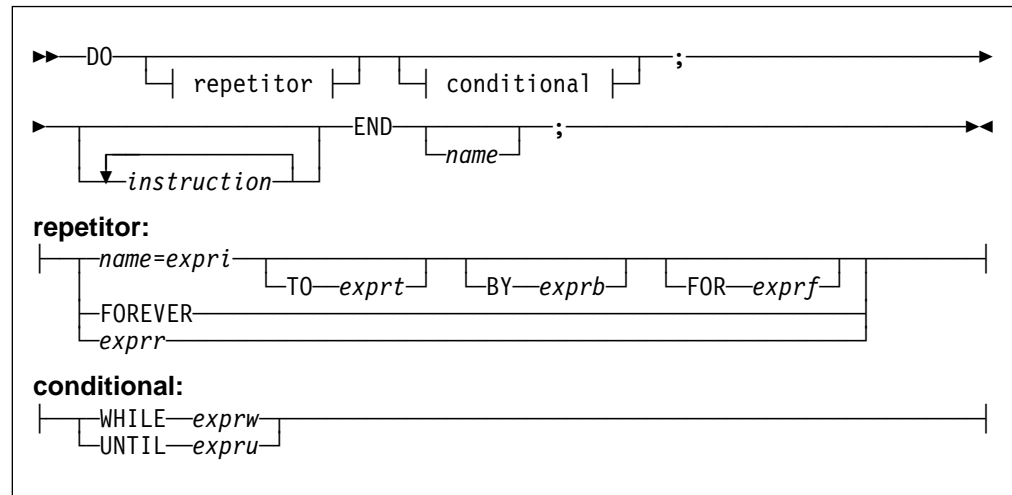
Note: When name is an external REXX program, the called program will run under the same language processor invocation as the calling program. This may affect the action of invocation-sensitive CL commands, such as Send Program Message (SNDPGMMSG), that may be issued from within the called program.

CALL

Where a new invocation of the REXX language processor is needed, use the Start REXX Procedure (STRREXPRC) command instead of the CALL instruction to run the program. For further information on invocation concerns, see the *REXX/400 Programmer's Guide*, SC41-5728.

Implementation maximum: The total nesting of control structures, such as DO END instructions, subroutines, and internal routines, may not exceed 100.

DO



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

Syntax Notes:

- The *exprr*, *expri*, *exprb*, *expri*, and *exprf* options (if present) are any expressions that evaluate to a number. The *exprr* and *exprf* options are further restricted to result in a positive whole number or zero. If necessary, the numbers are rounded according to the setting of NUMERIC DIGITS.
- The *exprw* or *expru* options (if present) can be any expression that evaluates to 1 or 0.
- The TO, BY, and FOR phrases can be in any order, if used, and are evaluated in the order in which they are written.
- The *instruction* can be any instruction, including assignments, commands, and keyword instructions (including any of the more complex constructs such as IF, SELECT, and the DO instruction itself).
- The subkeywords WHILE and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. Similarly, TO, BY, and FOR cannot be used in *expri*, *expri*, *exprb*, or *exprf*. FOREVER is also reserved, but only if it immediately follows the keyword DO and an equal sign does not follow it.
- The *exprb* option defaults to 1, if relevant.

Simple DO Group

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

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

Example:

```

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

```

Repetitive DO Loops

If a DO instruction has a repetitor phrase or a conditional phrase or both, the group of instructions forms a **repetitive DO loop**. The instructions are processed according to the repetitor phrase, optionally modified by the conditional phrase. (See “Conditional Phrases (WHILE and UNTIL)” on page 42).

Simple Repetitive Loops

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

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

Note: For a discussion on conditional phrases, see “Conditional Phrases (WHILE and UNTIL)” on page 42.

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

Example:

```

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

```

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

Controlled Repetitive Loops

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

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

The *expri*, *exprt*, and *exprb* options must result in numbers. They are evaluated only one time, before the loop begins and before the control variable is set to its

initial value. The default value for *exprb* is 1. If *exprt* is omitted, the loop runs indefinitely unless some other condition stops it.

Example:

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

The numbers do not have to be whole numbers:

Example:

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

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

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

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

Example:

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

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

DO

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

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

Example:

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

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

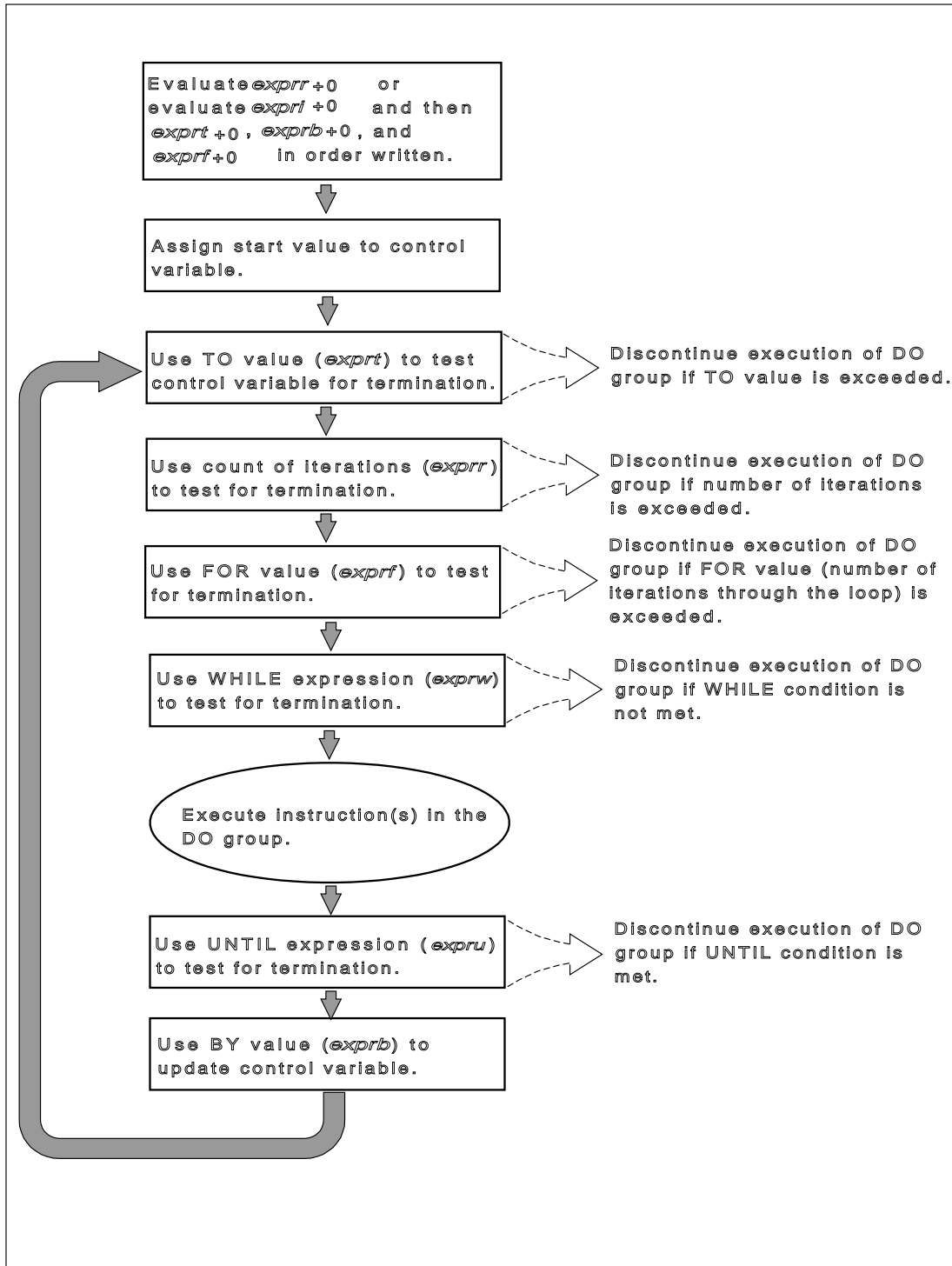
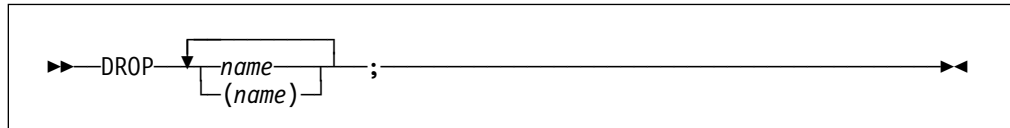


Figure 1. Concept of a DO Loop

DROP

DROP



DROP “unassigns” variables, that is, restores them to their original uninitialized state. If *name* is not enclosed in parentheses, it identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments.

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

Variables are dropped in sequence from left to right. It is not an error to specify a name more than one time or to DROP a variable that is not known. If an exposed variable is named

(see (see “PROCEDURE” on page 59), the variable in the older generation is dropped.

Example:

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

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

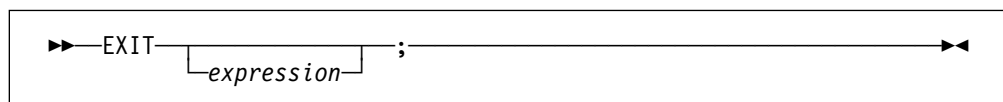
Example:

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

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

Example:

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

EXIT


EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller. The program is stopped immediately, even if an internal routine is currently being run. If no internal routine is active, RETURN (see page 65) and EXIT are identical in their effect on the program that is being run.

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

Example:

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

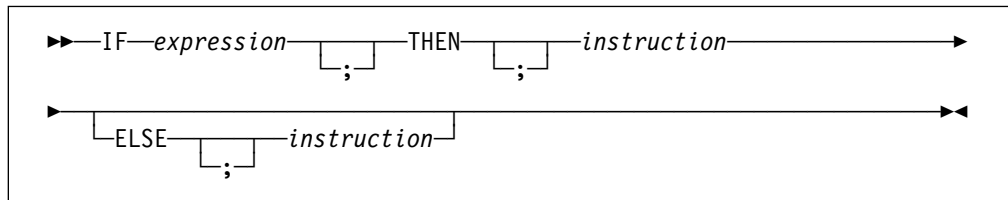
If you do not specify *expression*, no data is passed back to the caller. If the program was called as an external function, this is detected as an error—either immediately (if RETURN was used), or on return to the caller (if EXIT was used).

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

Notes:

1. The language processor distinguishes between invocation as a command and invocation as a subroutine or function only in that, if it was called through a command interface, an attempt is made to convert the returned value to a return code acceptable by the underlying operating system. The underlying operating system is the current command environment. If the conversion fails, it is deemed to be unsuccessful due to the underlying operating system and thus is not subject to trapping with SIGNAL ON SYNTAX.
2. If the caller is another REXX program that used the CALL instruction, the value is returned to the special variable RESULT.
3. If the caller is another REXX program that used a function invocation, the value replaces the function invocation in the calling program.
4. If REXX is called using the Start REXX Procedure (STREXPRC) command or a user-defined command and the value of the expression is not 0, then the returned string will be incorporated into the escape message CPF7CFF. The invoking program can then receive the message to obtain the returned value. In this case the return code must be a whole number within the range -2^{15} to $2^{15}-1$. If it is not, instead of escape message CPF7CFF, the REXX program will stop with REXX Error 26, “Invalid whole number”. Diagnostic message CPD7C9A is issued followed by escape message CPF7CFD.
5. REXX programs that call other REXX programs using the STREXPRC command will have to obtain nonzero return values by receiving the escape message and parsing the message data. The special variable RC will contain the escape message CPF7CFF.
6. See Chapter 9, “AS/400 System Interfaces” on page 147 for more information.

IF



IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* is evaluated and must result in 0 or 1.

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

Example:

```

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

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

Example:

```

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

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

Example:

```

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

Notes:

1. The *instruction* can be any assignment, command, or keyword instruction, including any of the more complex constructs such as DO, SELECT, or the IF instruction itself. A null clause is not an instruction, so putting an extra semicolon (or label) after the THEN or ELSE is not equivalent to putting a dummy instruction (as it would be in PL/I). The NOP instruction is provided for this purpose.
2. The symbol THEN cannot be used within *expression*, because the keyword THEN is treated differently, in that it need not start a clause. This allows the expression on the IF clause to be ended by the THEN, without a ; being required. If this were not so, people who are accustomed to other computer languages would experience considerable difficulties.

INTERPRET

►► INTERPRET *expression* ; ◀◀

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

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

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

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

Example:

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

Example:

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

Notes:

1. Label clauses are not permitted in an interpreted character string. If a SIGNAL or CALL instruction is issued or a trapped event occurs as a result of an INTERPRET instruction, this causes an immediate exit from the interpreted instruction or expression before the label search begins.
2. If you are new to the concept of the INTERPRET instruction and are getting results that you do not understand, you may find that executing it with TRACE R or TRACE I in effect is helpful.

Example:

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

When this is run, it gives the trace:

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

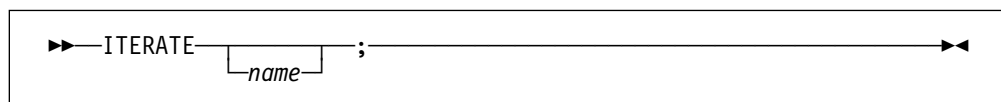
Hello Kitty!

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

3. For many purposes, you can use the VALUE function (see page 103) instead of the INTERPRET instruction. The following line could, therefore, have replaced line 5 in the last example:

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

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

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 processed again, unless the DO instruction ends the loop.

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

Example:

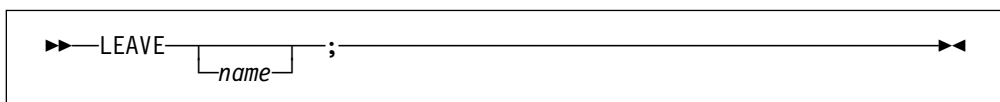
```

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

Notes:

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

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

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


Example:

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

Notes:

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

NOP



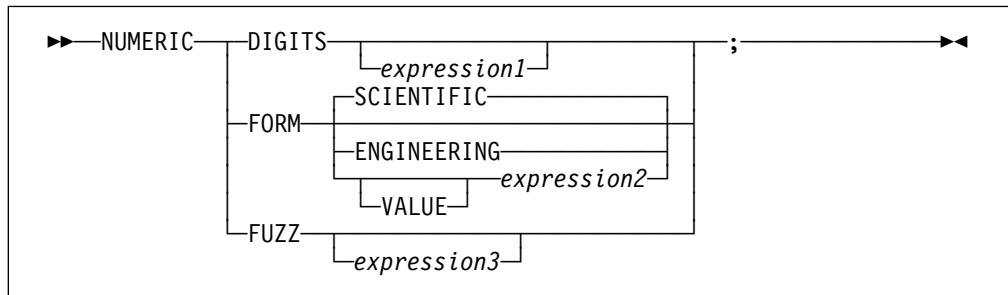
NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause:

Example:

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

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

NUMERIC



NUMERIC changes the way in which a program carries out arithmetic operations. The options of this instruction are described in detail on pages 127-136, but in summary:

NUMERIC DIGITS

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

There is no limit to the value for DIGITS (except the 16 megabyte limit on the total storage used for all REXX variables) 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 89.

NUMERIC FORM

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

You can retrieve the current NUMERIC FORM setting with the FORM built-in function. See "FORM" on page 91.

NUMERIC FUZZ

controls how many digits, at full precision, are ignored during a numeric comparison operation. (See page 133.) If you omit *expression3*, the default is 0 digits. Otherwise, *expression3* must evaluate to 0 or a positive whole number, rounded if necessary according to the current NUMERIC DIGITS setting, and must be smaller than the current NUMERIC DIGITS setting.

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

You can retrieve the current NUMERIC FUZZ setting with the FUZZ built-in function. See “FUZZ” on page 92.

Note: The three numeric settings are automatically saved across internal and external subroutine and function calls. See the CALL instruction (page 35) for more details.

OPTIONS

►►—OPTIONS—*expression*—;——————►►

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

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

The language processor recognizes the following words:

- 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.

The language processor recognizes the following words when they are the only word in *expression* and are specified as literal strings:

- 'ETMODE'** specifies that literal strings and comments containing DBCS characters are checked for being valid DBCS strings. ETMODE is the default when the REXX source file is tagged with a mixed Coded Character Set ID (CCSID).
- 'NOETMODE'** specifies that literal strings and comments containing DBCS characters are not checked for being valid DBCS strings. NOETMODE is the default. NOETMODE is also the default when the REXX source file is not tagged with a mixed CCSID.

Notes:

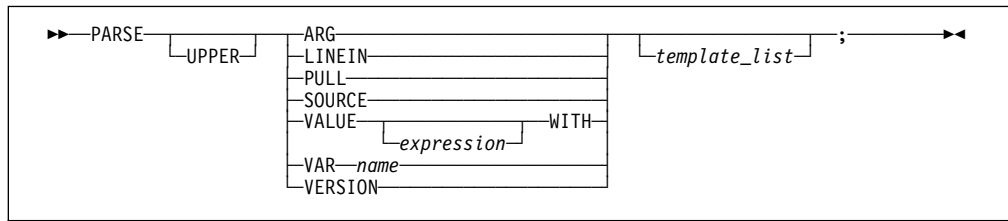
1. Because of the language processor's scanning procedures, you are advised to place an OPTIONS 'ETMODE' instruction as the first instruction in a program containing DBCS characters in literal strings and DBCS comments. Otherwise, errors may occur in processing the DBCS literal strings and DBCS comments that follow.
2. All REXX/400 source files tagged with a mixed CCSID (CCSID with a double-byte character set component) are treated as if OPTIONS 'ETMODE' was coded in the REXX source file. All other REXX source files are treated as if OPTIONS 'NOETMODE' was coded. The source file's IGCDATA attribute is ignored.

A REXX source file written in CCSID 65535 (*HEX) must use OPTIONS 'ETMODE' when literal strings and comments containing DBCS characters are to be checked for being valid DBCS strings. Only one OPTIONS 'ETMODE' or OPTIONS 'NOETMODE' may be specified per REXX source file and they are ignored in source files not written in CCSID 65535. OPTIONS 'NOETMODE'

cannot be used to override either the implicit or the explicit occurrence of `OPTIONS 'ETMODE'`.

3. The `EXMODE` setting is saved and restored across subroutine and function calls.
4. The words `EXMODE` and `NOEXMODE` can appear several times within the result. The one that takes effect is determined by the last valid one specified.
5. See Appendix A, “Double-Byte Character Set (DBCS) Support” on page 177 for more information on DBCS, SBCS, and associated functions.

PARSE



PARSE assigns data (from various sources) to one or more variables according to the rules of parsing. (See Chapter 5, “Parsing” on page 111.)

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 121 for information on parsing multiple source strings.

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

PARSE ARG

parses the string or strings passed to a program or internal routine as input arguments. (See the ARG instruction on page 33 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 80).

PARSE LINEIN

parses the next line from the default input stream. (See Chapter 8, “Input and Output Streams” on page 143 for a discussion of REXX input and output.) If no line is available, program execution will usually pause until a line is complete. Note that PARSE LINEIN should be used only when direct access to the character input stream is necessary. Usual line-by-line dialogue with the user should be carried out with the PULL or PARSE PULL instructions, to maintain generality.

PARSE LINEIN reads a line from STDIN, by default the Integrated Language Environment (ILE) Session Manager. In doing so, it works like a pause, by forcing you to press ENTER to continue. For more information on the ILE Session Manager, see “Terminal Input and Output” on page 149.

PARSE PULL

parses the next string from the external data queue. If the external data queue is empty, PARSE PULL reads a line from the default input stream (the user's terminal), and the program pauses, if necessary, until a line is complete. You can add data to the head or tail of the queue by using the PUSH and QUEUE instructions, respectively. You can find the number of lines currently in the queue with the QUEUED built-in function. (See page 95.) The queue remains active for the life of the job. Other programs which run in the job can alter the queue and use it as a means of communication with programs written in REXX. See also the PULL instruction on page 62.

Note: PULL and PARSE PULL read first from the external data queue. If the external data queue is empty, they read from STDIN. STDIN defaults to the ILE Session Manager display. (See the PULL instruction, on page 62, for further details.)

PARSE SOURCE

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

The source string contains the characters OS/400, followed by either COMMAND, FUNCTION, or SUBROUTINE, depending on whether the program was run using a command, such as Start REXX Procedure (STRREXPRC), a function call in an expression, or the CALL instruction. These two tokens are followed by the member, file and library name of the source member.

For example, if the REXTRY procedure stored in the QREXSRC source file in library USER1 were to enter the PARSE SOURCE instruction, the string parsed would look like this:

```
OS/400 COMMAND REXTRY QREXSRC USER1
```

PARSE VALUE

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

Thus, for example:

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

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

PARSE VAR *name*

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

```
PARSE VAR string word1 string
```

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

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

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

PARSE VERSION

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

PARSE

1. A word describing the language. The first four letters are the characters REXX, and the remainder may be used to identify a particular implementation or language processor. REXX/400 returns REXXSAA for this word.
2. The language level description, for example, 3.48. Numbers smaller than this may be assumed to indicate a subset of the language defined here.
3. Three tokens describing the language processor release date in the same format as the default for the DATE function (see "DATE" on page 87), for example 13 June 1989.

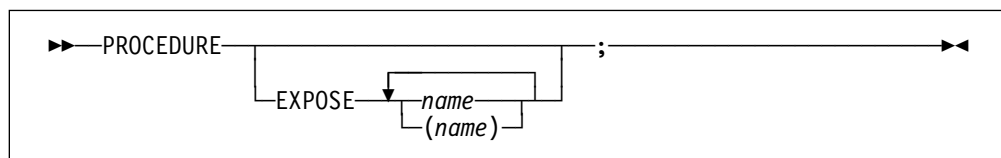
The returned string would look like the following:

```
REXSAA 3.48 13 June 1989
```

Notes:

1. Input retrieved by PARSE PULL or PULL is obtained from STDIN, which defaults to the ILE Session Manager display (if the external data queue is empty). Lines pulled from STDIN will be echoed to the job log, as a command (CMD) message, if REXX tracing is active.
2. In batch mode, STDIN, by default, is directed to the file QINLINE. If there is no data to read, the result of the PULL instruction is a null string. Lines read by these instructions are echoed to the job log, as a command (CMD) message.

PROCEDURE



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

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

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

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

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

PROCEDURE

Example:

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

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

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

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

Example:

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

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

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

Example:

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

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

Specifying a **stem** as *name* exposes this stem and *all possible* compound variables whose names begin with that stem. (See page 24 for information about stems.)

Example:

```

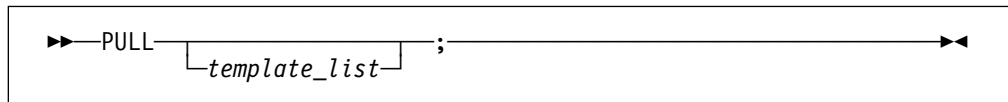
/* This is the main REXX program */
a.=11; i=13; j=15
i = i + 1
C.5 = 'FRED'
call lucky7
say a. a.1 i j c. c.5
say 'You should see 11 7 14 15 C. FRED'
exit
lucky7:Procedure Expose i j a. c.
/* This exposes I, J, and all variables whose      */
/* names start with A. or C.                      */
A.1='7'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

```

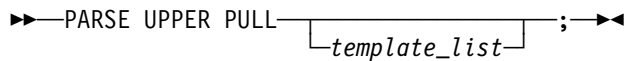
Variables may be exposed through several generations of routines, if desired, by ensuring that they are included on all intermediate PROCEDURE instructions.

See the CALL instruction and function descriptions on pages 35 and 75 for details and examples of how routines are called.

PULL



PULL reads a string from the head of the external data queue. (See Chapter 8, “Input and Output Streams” on page 143 for a discussion of REXX input and output.) It is just a short form of the instruction:



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 111). Use the PARSE PULL instruction if you do not desire uppercase translation.

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.

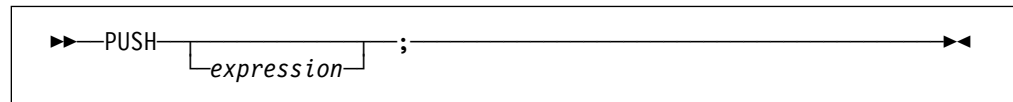
If the external data queue is empty, a line is read from the default input stream and the program pauses, if necessary, until a line is complete. (This is as though PARSE UPPER LINEIN had been processed. See page 56.)

The QUEUED built-in function (see page 95) returns the number of lines currently in the external data queue.

Notes:

1. PULL and PARSE PULL read first from the external data queue. If the queue is empty, they read from STDIN, by default the ILE Session Manager display.
2. See Chapter 8, “Input and Output Streams” on page 143 for more information on STDIN.

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 143 for a discussion of REXX input and output.)

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

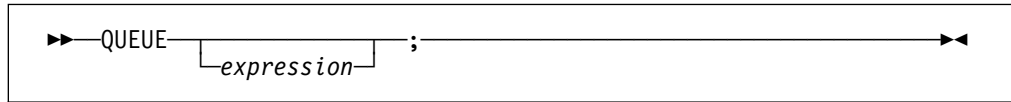
Example:

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

The QUEUED built-in function (described on page 95) returns the number of lines currently in the external data queue.

Implementation maximum: The length of a single string on the external data queue is limited to 32,767. The string will be truncated at this point, with no error indication.

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 143 for a discussion of REXX input and output.)

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

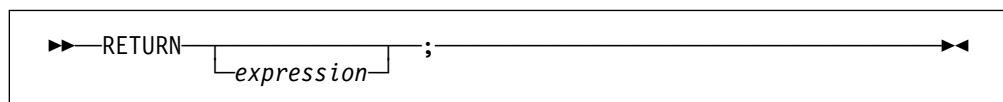
Example:

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

The QUEUED built-in function (described on page 95) returns the number of lines currently in the external data queue.

Implementation maximum: The length of a single string on the external data queue is limited to 32,767 bytes. The string will be truncated at this point, with no error indication.

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 run. (See page 45.)

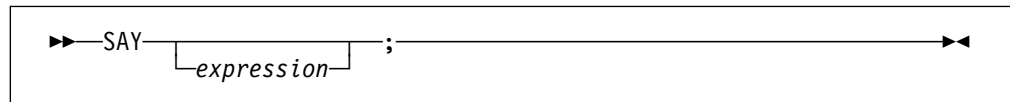
If a *subroutine* is being run (see the CALL instruction), *expression* (if any) is evaluated, control passes back to the caller, and the REXX special variable RESULT is set to the value of *expression*. If *expression* is omitted, the special variable RESULT is dropped (becomes uninitialized). The various settings saved at the time of the CALL (tracing, addresses, and so forth) are also restored. (See page 35.)

If a *function* is being processed, the action taken is identical, except that *expression* **must** be specified on the RETURN instruction. The result of *expression* is then used in the original expression at the point where the function was called. See the description of functions on page 75 for more details.

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

Notes:

1. If the procedure is called using the REXX CALL instruction, the value is returned to the special variable RESULT.
2. If the procedure is called using a REXX function invocation, the value replaces the function invocation in the calling program.
3. If neither Note 1 nor Note 2 apply and the value of the expression is not 0, the returned string will be incorporated into the escape message CPF7CFF. The invoking program can then receive the message to obtain the returned value. In this case the return code must be a whole number within the range $-2^{**}15$ to $2^{**}15-1$. If it is not, instead of escape message CPF7CFF, the REXX program will stop with REXX Error 26 "Invalid whole number". Diagnostic message CPD7C9A is issued followed by escape message CPF7CFD. REXX programs that call other REXX programs using the Start REXX Procedure (STRREXPRC) command will have to obtain nonzero return values by receiving the escape message. The special variable RC will contain the escape message number CPF7CFF.

SAY

SAY writes a line to the default output stream. The result of *expression* may be of any length. If you omit *expression*, the null string is written. Data output by the SAY instruction is written to the file STDOUT, which defaults to the Integrated Language Environment (ILE) Session Manager display. (See Chapter 8, “Input and Output Streams” on page 143 for a discussion of REXX input and output.)

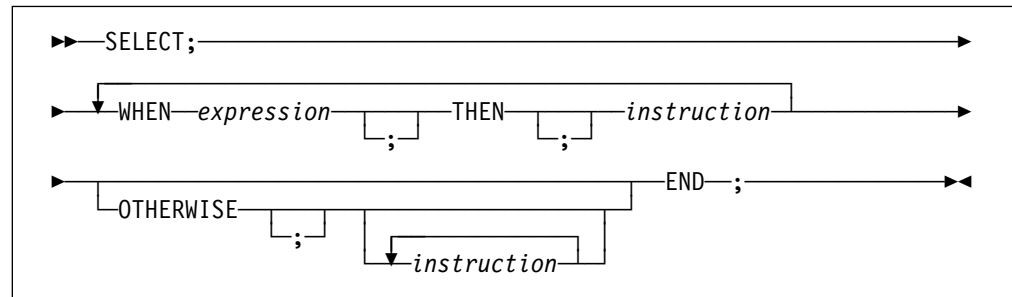
Example:

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

Notes:

1. Output from SAY is directed to STDOUT, which, in interactive mode, defaults to the ILE session manager. In batch mode, STDOUT defaults to the file QPRINT.
2. If REXX tracing is in effect, output is placed in the job log as a command (CMD) message.
3. In batch mode, all output from SAY instructions is logged to the job log, regardless of the current REXX trace setting.

SELECT



SELECT conditionally calls one of several alternative instructions.

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

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

Example:

```

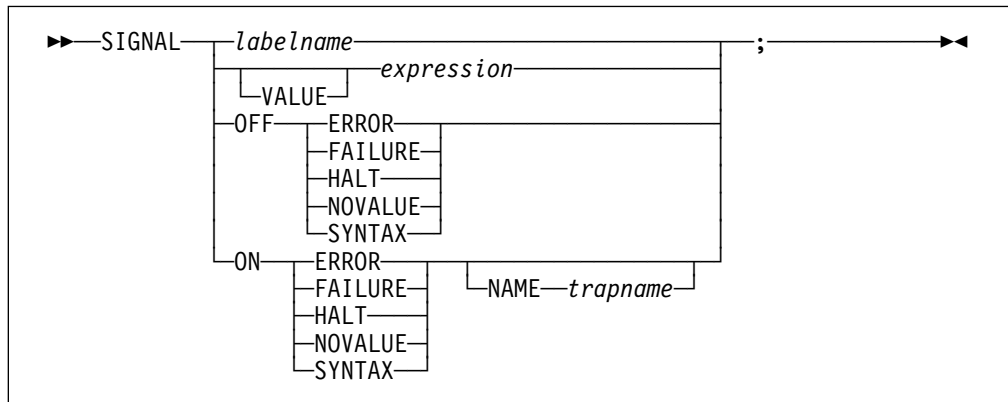
balance=100
check=50
balance = balance - check
Select
  when balance > 0 then
    say 'Congratulations! You still have' balance 'dollars left.'
  when balance = 0 then do
    say 'Warning, Balance is now zero! STOP all spending.'
    say "You cut it close this month! Hope you do not have any"
    say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

```

Notes:

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

SIGNAL



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

To control trapping, you specify OFF or ON and the condition you want to trap. OFF turns off the specified condition trap. ON turns on the specified condition trap. All information on condition traps is contained in Chapter 7, "Conditions and Condition Traps" on page 137.

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

Example:

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

Because the search effectively starts at the top of the program, 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 transfer of control to a label.

Using SIGNAL VALUE

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

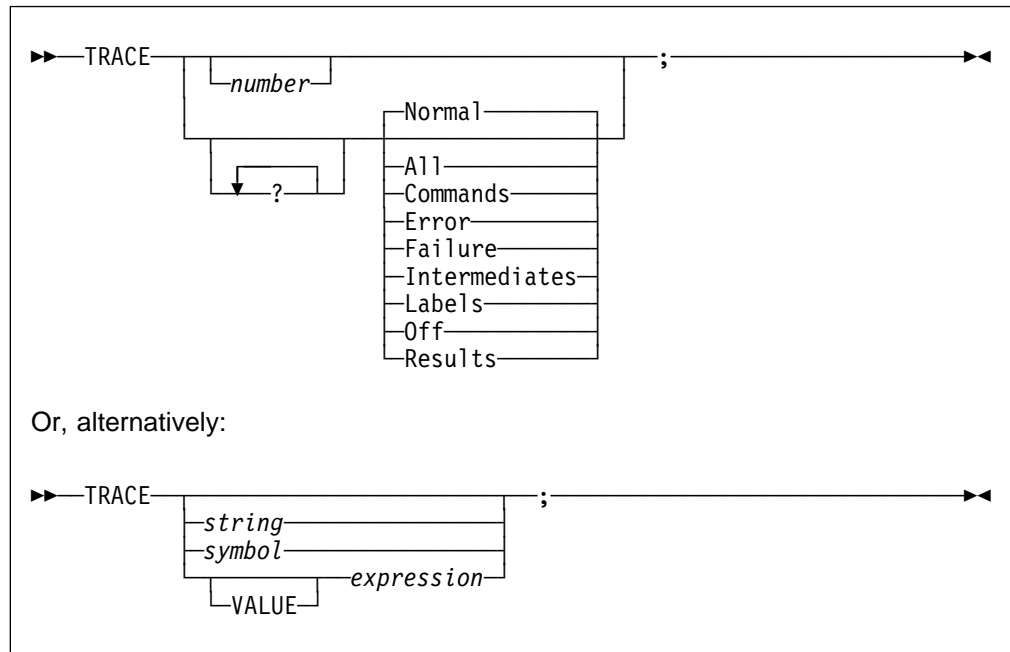
Example:

```

fred='PETE'
call multiway fred, 7
    ....
    ....
exit
Multiway: procedure
    arg label .                /* One word, uppercase */
                                /* Can add checks for valid labels here */
    signal value label        /* Transfer control to wherever */
    ....
Pete: say arg(1) '!' arg(2) /* Displays: "PETE ! 7" */
    return

```

TRACE



TRACE controls the tracing action (that is, how much is displayed to the user) during processing of a REXX program. (Tracing describes some or all of the clauses in a program, producing descriptions of clauses as they are processed.) TRACE is mainly used for debugging. Its syntax is more concise than that of other REXX instructions because TRACE is usually entered manually during interactive debugging. (This is a form of tracing in which the user can interact with the language processor while the program is running.) For this use, economy of key strokes is especially convenient.

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

The *string* or *expression* evaluates to:

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

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

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

The option that follows TRACE or the result of evaluating *expression* determines the tracing action. You can omit the subkeyword VALUE if *expression* does not begin with a symbol or a literal string (that is, if it starts with a special character, such as an operator or parenthesis).

Alphabetic Character (Word) Options

Although you can enter the word in full, only the capitalized and highlighted letter is needed; all characters following it are ignored. That is why these are referred to as alphabetic character options.

TRACE actions correspond to the alphabetic character options as follows:

All	Traces (that is, displays) all clauses before execution.
Commands	Traces all commands before execution. If the command results in an error or failure, ³ then tracing also displays the return code from the command.
Error	Traces any command resulting in an error or failure ³ after execution, together with the return code from the command. On the AS/400 system, this means that any command invocation that causes an escape message to be issued will be indicated as an ERROR condition (but see Failure and Normal, below).
Failure	Traces any command resulting in a failure ³ after execution, together with the return code from the command. This is the same as the Normal option.
Intermediates	Traces all clauses before execution. Also traces intermediate results during evaluation of expressions and substituted names.
Labels	Traces only labels passed during execution. This is especially useful with debug mode, when the language processor pauses after each label. It also helps the user to note all internal subroutine calls and transfers of control because of the SIGNAL instruction.
Normal	Traces any failing command after execution, together with the return code from the command. This is the default setting. Any attempt to enter an unknown command will raise a FAILURE condition. An attempt to enter a command to an unknown command environment will also raise a FAILURE condition; in such a case, the variable RC will contain the escape message data for "object not found."
Off	Traces nothing and resets the special prefix option (described later) to OFF.
Results	Traces all clauses before execution. Displays final results (contrast with Intermediates, preceding) of evaluating an expression. Also displays values assigned during PULL, ARG, and PARSE instructions. This setting is recommended for general debugging.
?	Controls interactive debug. During usual execution, a TRACE option with a prefix of ? causes interactive debug to be switched on. 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).

³ See page 26 for definitions of error and failure.

TRACE

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

You can switch off interactive debug in several ways:

- Entering TRACE 0 turns off all tracing.
- Entering TRACE with no options restores the defaults—it turns off interactive debug but continues tracing with TRACE Normal (which traces any failing command after execution) in effect.
- Entering TRACE ? turns off interactive debug and continues tracing with the current option.
- Entering a TRACE instruction with a ? prefix before the option turns off interactive debug and continues tracing with the new option.

Using the ? prefix, therefore, switches you alternately in or out of interactive debug. (Because the language processor ignores any further TRACE statements in your program after you are in interactive debug, use CALL TRACE '?' to turn off interactive debug.)

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, “Debug Aids” on page 173, for further information.) However, if the option is a negative whole number (or an expression that evaluates to a negative whole number), all tracing, including debug pauses, is temporarily inhibited for the specified number of clauses. For example, TRACE -100 means that the next 100 clauses that would usually be traced are not, in fact, displayed. After that, tracing resumes as before.

Tracing Tips

1. When a loop is being traced, the DO clause itself is traced on every iteration of the loop.
2. You can retrieve the trace actions currently in effect by using the TRACE built-in function (see “TRACE” on page 101).
3. Comments in the source REXX program are not included in the trace output.
4. Commands traced before execution always have the final value of the command (that is, the string passed to the environment), and the clause generating it produced in the traced output.
5. Trace actions are automatically saved across subroutine and function calls. See the CALL instruction (page 35) 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 appears 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 quotation marks so that leading and trailing blanks are apparent. The language processor may replace terminal control codes, such as EBCDIC values less than '40'X or the value 'FF'X, with a question mark to avoid terminal interference.

A line number precedes the first clause traced on any line. If the line number is greater than 99999, the language processor truncates it on the left, and the ? prefix indicates the truncation. For example, the line number 100354 appears as ?00354. All lines displayed during tracing have a three-character prefix to identify the type of data being traced. These can be:

- *-* Identifies the source of a single clause, that is, the data actually in the program.
- +++ Identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program (see below).
- >>> Identifies the result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.
- >.> Identifies the value “assigned” to a placeholder during parsing (see page 113).

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

- >C> The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.
- >F> The data traced is the result of a function call.
- >L> The data traced is a literal (string, uninitialized variable, or constant symbol).
- >0> The data traced is the result of an operation on two terms.
- >P> The data traced is the result of a prefix operation.
- >V> The data traced is the contents of a variable.

If no option is specified on a TRACE instruction, or if the result of evaluating the expression is null, the default tracing actions are restored. The defaults are TRACE N and interactive debug (?) off.

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. The special trace prefix +++ identifies these traceback lines.

TRACE

Notes:

1. TRACE output is sent to the job log as command (CMD) messages. These may be distinguished from other messages by the three character prefix.
2. In interactive mode, TRACE input and output is sent to the job log.
3. In batch mode, TRACE output is sent to the job log. All interactive trace reading is ignored. For example, if a REXX program issues the instruction TRACE ?A while in batch mode, it is treated as if the instruction was TRACE A.
4. The Trace REXX (TRCREX) command can be used to modify REXX's trace behavior without using the TRACE instruction or modifying the REXX program. See the *CL Reference* for more information.
5. See Chapter 10, "Debug Aids" on page 173 for some additional debugging tools.

Chapter 4. Functions

A **function** is an internal, built-in, or external routine that returns a single result string. (A **subroutine** is a function that is an internal, built-in, or external routine that may or may not return a result and that is called with the CALL instruction.)

Syntax

A **function call** is a term in an expression that calls a routine that carries out some procedures and returns a string. This string replaces the function call in the continuing evaluation of the expression. You can include function calls to internal and external routines in an expression anywhere that a data term (such as a string) would be valid, using the notation:

The diagram shows the syntax for a function call: `function_name (' ' expression)`. The `function_name` is followed by a left parenthesis `(`. Inside the parentheses, there is a single quote `'`, a space, another single quote `'`, and the word `expression`. The parentheses are closed with a right parenthesis `)`. Arrows point to the left and right of the entire expression, and a bracket underlines the `expression` part.

The *function_name* is a literal string or a single symbol, which is taken to be a constant.

There can be up to an implementation-defined maximum number of expressions, separated by commas, between the parentheses. On the AS/400 system, this implementation maximum is 20 expressions.

These expressions are called the **arguments** to the function. Each argument expression may include further function calls.

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

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

For example, the function SUBSTR is built-in to the language processor (see page 99) and could be used as:

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

A function may have a variable number of arguments. You need to specify only those that are required. For example, SUBSTR('ABCDEF',4) would return DEF.

Functions and Subroutines

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

The following types of routines can be called as functions:

Internal If the routine name exists as a label in the program, the current processing status is saved, so that it is later possible to return to the point of invocation to resume execution. Control is then passed to the first label in the program that matches the name. As with a routine called by the CALL instruction, various other status information (TRACE and NUMERIC settings and so forth) is saved too. See the CALL instruction (page 35) 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 69).

If you are calling an internal routine as a function, you *must* specify an expression in any RETURN instruction to return from it. This is not necessary if it is called as a subroutine.

Example:

```
/* Recursive internal function execution... */
arg x
say x! = ' factorial(x)
exit

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

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

Built-in These functions are always available and are defined in the next section of this manual. (See pages 78—108.)

External You can write or use functions that are external to your program and to the language processor. An external routine can be written in any language (including REXX) that supports the system-dependent interfaces the language processor uses to call it. You can call a REXX program as a function and, in this case, pass more than one argument string. The ARG or PARSE ARG instructions or the ARG built-in function can retrieve these argument strings. For more information on external functions and subroutines, see “External Functions and Subroutines” on page 156. When called as a function, a program 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 called REXX program, and in either case you must specify an expression.
3. With care, you can use the INTERPRET instruction to process a function with a variable function name. However, you should avoid this if possible because it reduces the clarity of the program.

Search Order

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

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

Note: The words STREAM, LINES, LINEIN, LINEOUT, CHARS, CHARIN, and CHAROUT are reserved. You can use them as the name of an internal routine, but you should not use them as the name of an external routine.

Example:

```
/* This internal DATE function modifies the      */
/* default for the DATE function to standard date. */
date: procedure
    arg in
    if in='' then in='Standard'
    return 'DATE'(in)
```

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

External functions and **subroutines** have a specific search order. If a function is not found in the search for internal routines and built-in functions, then REXX will search for it as follows:

1. A member in the same source file that the call was made from.
2. Next, the library list is searched for the first occurrence of a file named QREXSRC. If found, QREXSRC is searched for a member with the desired name.
3. Finally, the library list is searched (from the top) for a program object with the desired name.

Errors During Execution

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

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

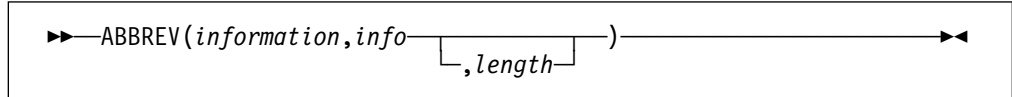
Built-in Functions

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

The following are general notes on the built-in functions:

- The parentheses in a function are always needed, even if no arguments are required. The first parenthesis must follow the name of the function with no space in between.
- The built-in functions work internally with NUMERIC DIGITS 9 and NUMERIC FUZZ 0 and are unaffected by changes to the NUMERIC settings, except where stated. Any argument named as a *number* is rounded, if necessary, according to the current setting of NUMERIC DIGITS (just as though the number had been added to 0) and checked for validity before use. This occurs in the following functions: ABS, FORMAT, MAX, MIN, SIGN, and TRUNC, and for certain options of DATATYPE.
- Any argument named as a *string* may be a null string.
- If an argument specifies a *length*, it must be a positive whole number or zero. If it specifies a start character or word in a string, it must be a positive whole number, unless otherwise stated.
- Where the last argument is optional, you can always include a comma to indicate you have omitted it; for example, DATATYPE(1,), like DATATYPE(1), would return NUM.
- If you specify a *pad* character, it must be exactly one character long. (A pad character extends a string, usually on the right. For an example, see the LEFT built-in function on page 93.)
- If a function has an *option* you can select by specifying the first character of a string, that character can be in upper- or lowercase.
- A number of the functions described in this chapter support DBCS. A complete list and descriptions of these functions are in Appendix A, “Double-Byte Character Set (DBCS) Support” on page 177.

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 positive whole number or zero. The default for *length* is the number of characters in *info*.

Here are some examples:

```

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

```

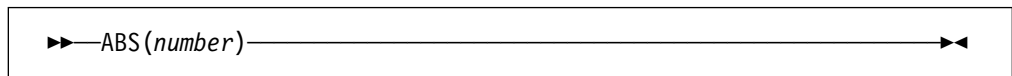
Note: A null string always matches if a length of 0 (or the default) is used. This allows a default keyword to be selected automatically if desired; for example:

```

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

```

ABS (Absolute Value)



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

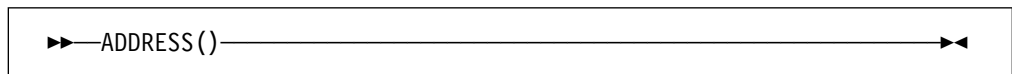
Here are some examples:

```

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

```

ADDRESS



returns the name of the environment to which commands are currently being submitted. See the ADDRESS instruction (page 30) for more information. Trailing blanks are removed from the result. For the default system environment, ADDRESS returns the string COMMAND.

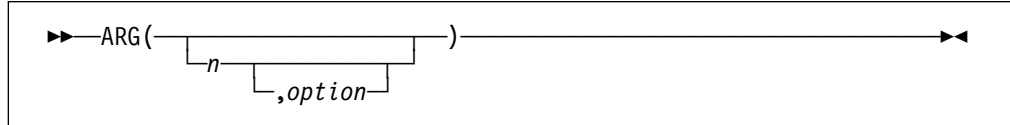
Functions

Here are some examples:

```
ADDRESS()  ->  'COMMAND'  /* the CL environment  */
ADDRESS()  ->  'EDIT'      /* possible editor    */
ADDRESS()  ->  '*LIBL/ABC' /* your ABC program  */
```

Note: If the command environment has been specified using *LIBL or *CURLIB, then ADDRESS() will return a string including *LIBL or *CURLIB. The actual library used for the command environment is determined by the state of *CURLIB or *LIBL when the first command was entered to the user-defined environment.

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. The *n* must be a positive whole number.

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

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

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

Here are some examples:

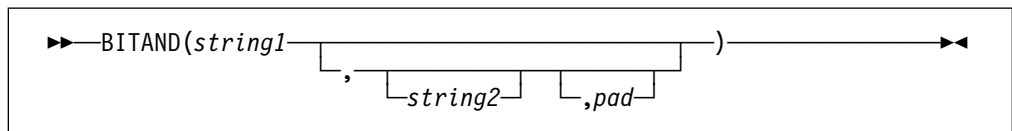
```
/* following "Call name;" (no arguments) */
ARG()      ->  0
ARG(1)     ->  ''
ARG(2)     ->  ''
ARG(1,'e') ->  0
ARG(1,'O') ->  1
```

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

Notes:

1. The number of argument strings is the largest number *n* for which ARG(*n*, 'e') would return 1 or 0 if there are no explicit argument strings. That is, it is the position of the last explicitly specified argument string.
2. Programs called as commands can have only 0 or 1 argument strings. The program has 0 argument strings if it is called with the name only and has 1 argument string if anything else (including blanks) is included with the command.
3. You can retrieve and directly parse the argument strings to a program or internal routine with the ARG or PARSE ARG instructions. (See pages 33, 56, and 111.)

BITAND (Bit by Bit AND)



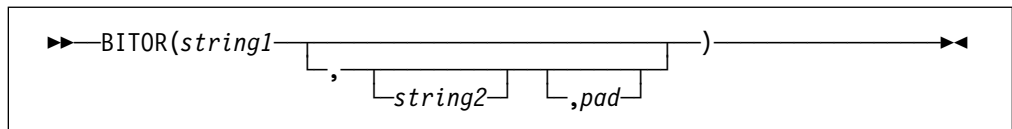
returns a string composed of the two input strings logically ANDed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the AND operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```

BITAND('12'x)           -> '12'x
BITAND('73'x,'27'x)     -> '23'x
BITAND('13'x,'5555'x)   -> '1155'x
BITAND('13'x,'5555'x,'74'x) -> '1154'x
BITAND('pQrS',,'BF'x)   -> 'pqrs' /* EBCDIC */
    
```

BITOR (Bit by Bit OR)



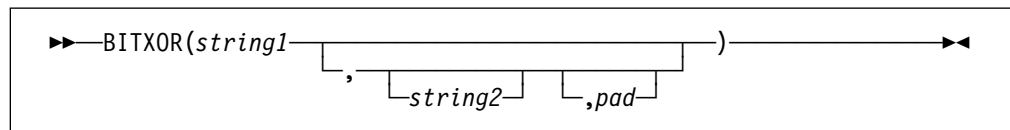
returns a string composed of the two input strings logically inclusive-ORed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the OR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Functions

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('Fred',', '40'x) -> 'FRED' /* EBCDIC */
```

BITXOR (Bit by Bit Exclusive OR)

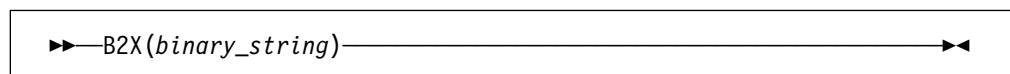


returns a string composed of the two input strings logically eXclusive-ORed together, bit by bit. (The encodings of the strings are used in the logical operation.) The length of the result is the length of the longer of the two strings. If no *pad* character is provided, the XOR operation stops when the shorter of the two strings is exhausted, and the unprocessed portion of the longer string is appended to the partial result. If *pad* is provided, it extends the shorter of the two strings on the right before carrying out the logical operation. The default for *string2* is the zero length (null) string.

Here are some examples:

```
BITXOR('12'x)          -> '12'x
BITXOR('12'x,'22'x)    -> '30'x
BITXOR('1211'x,'22'x)  -> '3011'x
BITXOR('1111'x,'444444'x) -> '555544'x
BITXOR('1111'x,'444444'x,'40'x) -> '555504'x
BITXOR('1111'x,', '4D'x) -> '5C5C'x
BITXOR('C711'x,'222222'x,' ') -> 'E53362'x /* EBCDIC */
```

B2X (Binary to Hexadecimal)



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

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

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

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

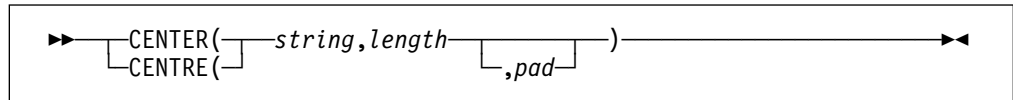
Here are some examples:

```
B2X('11000011') -> 'C3'
B2X('10111')    -> '17'
B2X('101')      -> '5'
B2X('1 111 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



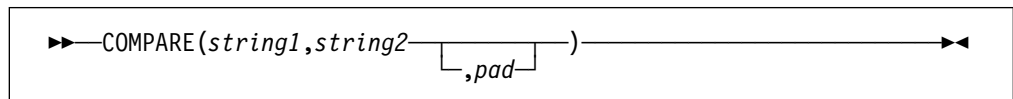
returns a string of length *length* with *string* centered in it, with *pad* characters added as necessary to make up length. The *length* must be a positive whole number or zero. The default *pad* character is blank. If the string is longer than *length*, it is truncated at both ends to fit. If an odd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

Here are some examples:

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

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

COMPARE

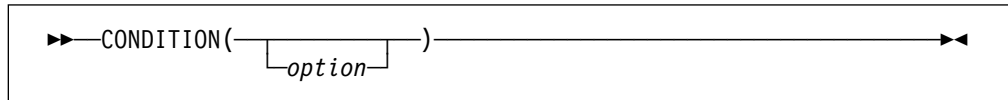


returns 0 if the strings, *string1* and *string2*, are identical. Otherwise, returns the position of the first character that does not match. The shorter string is padded on the right with *pad* if necessary. The default *pad* character is a blank.

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 137 for a description of condition traps.) You can request the following pieces of information:

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

To select the information to return, use the following *options*. (Only the capitalized and highlighted letter is needed; all characters following it are ignored.)

Condition name	returns the name of the current trapped condition.
Description	returns any descriptive string associated with the current trapped condition. See page 141 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 <i>option</i> .
Status	returns the status of the current trapped condition. This can change during processing, and is either: <ul style="list-style-type: none"> ON - the condition is enabled OFF - the condition is disabled DELAY - any new occurrence of the condition is delayed or ignored.

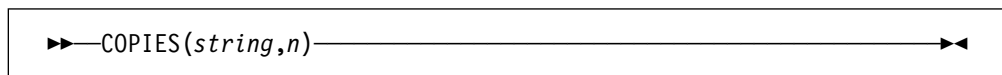
If no condition has been trapped, then the `CONDITION` function returns a null string in all four cases.

Here are some examples:

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

Note: The `CONDITION` function returns condition information that is saved and restored across subroutine calls (including those a CALL ON condition trap causes). Therefore, after a subroutine called with CALL ON *trapname* has returned, the current trapped condition reverts to the condition that was current before the CALL took place (which may be none). `CONDITION` returns the values it returned before the condition was trapped.

COPIES

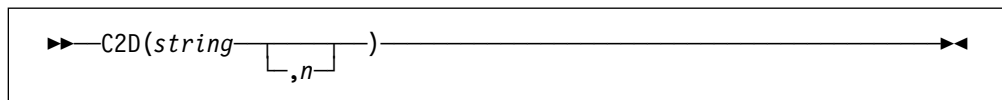


returns *n* concatenated copies of *string*. The *n* must be a positive whole number or zero.

Here are some examples:

```
COPIES('abc',3)    ->  'abcbcabcb'
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('')           ->      0
C2D('a')         ->     129   /* EBCDIC */
```

If you specify *n*, the string is taken as a signed number expressed in *n* characters. The number is positive if the leftmost bit is off, and negative, in two's complement notation, if the leftmost bit is on. In both cases, it is converted to a whole number, which may, therefore, be negative. The *string* is padded on the left with '00'x characters (note, not "sign-extended"), or truncated on the left to *n* characters. This padding or truncation is as though RIGHT(*string*,*n*, '00'x) had been processed. If *n* is 0, C2D always returns 0.

Here are some examples:

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

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

C2X (Character to Hexadecimal)

▶▶ C2X(*string*) ◀◀

returns a string, in character format, that represents *string* converted to hexadecimal. The returned string contains twice as many bytes as the input string. For example, on an EBCDIC system, C2X(1) returns F1 because the EBCDIC representation of the character 1 is 'F1'X.

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

Here are some examples:

```
C2X('72s')      ->  'F7F2A2' /* 'C6F7C6F2C1F2'X in EBCDIC */
C2X('0123'X)   ->  '0123' /* 'F0F1F2F3'X   in EBCDIC */
```

DATATYPE

▶▶ DATATYPE(*string* [,*type*]) ◀◀

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

If you specify *type*, returns 1 if *string* matches the type; otherwise returns 0. If *string* is null, the function returns 0 (except when *type* is X, which returns 1 for a null string). The following are valid *types*. (Only the capitalized and highlighted letter is needed; all characters following it are ignored. Note that for the hexadecimal option, you must start your string specifying the name of the option with x rather than h.)

- Alphanumeric** returns 1 if *string* contains only characters from the ranges a–z, A–Z, and 0–9.
- Binary** returns 1 if *string* contains only the characters 0 or 1 or both.
- C** returns 1 if *string* is a mixed SBCS/DBCS string.
- Dcbc** returns 1 if *string* is a DBCS-only 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* is a valid symbol, that is if SYMBOL(*string*) would not return BAD. (See page 11.) Note that both uppercase and lowercase alphabets 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.

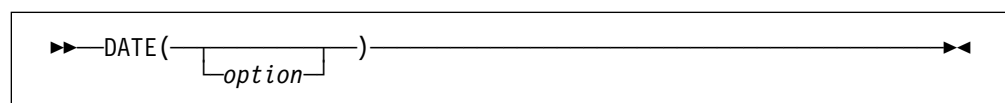
hexadecimal returns 1 if *string* contains only characters from the ranges a–f, A–F, 0–9, and blank (as long as blanks appear only between pairs of hexadecimal characters). Also returns 1 if *string* is a null string, which is a valid hexadecimal string.

Here are some examples:

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

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

DATE



returns, by default, the local date in the format: *dd mon yyyy* (day month year—for example, 13 Mar 1992), with no leading zero or blank on the day. 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 highlighted letter is needed; all characters following it are ignored.)

Base returns the number of complete days (that is, not including the current day) since and including the base date, 1 January 0001, in the format: *dddddd* (no leading zeros or blanks). The expression `DATE('B')//7` returns a number in the range 0–6 that corresponds to the current day of the week, where 0 is Monday and 6 is Sunday.

Note: The base date of 1 January 0001 is determined by extending the current Gregorian calendar backward (365 days each year, with an extra day every year that is divisible by 4 except century years that are not divisible by 400). It does not take into account any errors in the calendar system that created the Gregorian calendar originally.

Days returns the number of days, including the current day, so far in this year in the format: *ddd* (no leading zeros or blanks).

European returns date in the format: *dd/mm/yy*.

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).

Functions

Standard	returns date in the format: <i>yyyymmdd</i> (suitable for sorting, and so forth).
Usa	returns date in the format: <i>mm/dd/yy</i> .
Weekday	returns the English name for the day of the week, in mixed case, for example, Tuesday.

Here are some examples, assuming today is 13 March 1992:

```
DATE()      ->  '13 Mar 1992'
DATE('B')   ->  727269
DATE('D')   ->  73
DATE('E')   ->  '13/03/92'
DATE('M')   ->  'March'
DATE('N')   ->  '13 Mar 1992'
DATE('O')   ->  '92/03/13'
DATE('S')   ->  '19920313'
DATE('U')   ->  '03/13/92'
DATE('W')   ->  'Friday'
```

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

DBCS (Double-Byte Character Set Functions)

The following are all DBCS processing functions. See page 177.

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

DELSTR (Delete String)

▶▶ DELSTR(*string*,*n* [,*length*]) ▶▶

returns *string* after deleting the substring that begins at the *n*th character and is of *length* characters. If you omit *length*, or if *length* is greater than the number of characters from *n* to the end of *string*, the function deletes the rest of *string* (including the *n*th character). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the length of *string*, the function returns *string* unchanged.

Here are some examples:

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

DELWORD (Delete Word)

►► DELWORD(*string*,*n* [,*length*]) ◄◄

returns *string* after deleting the substring that starts at the *n*th word and is of *length* blank-delimited words. If you omit *length*, or if *length* is greater than the number of words from *n* to the end of *string*, the function deletes the remaining words in *string* (including the *n*th word). The *length* must be a positive whole number or zero. The *n* must be a positive whole number. If *n* is greater than the number of words in *string*, the function returns *string* unchanged. The string deleted includes any blanks following the final word involved but none of the blanks preceding the first word involved.

Here are some examples:

```
DELWORD('Now is the time',2,2) -> 'Now time'
DELWORD('Now is the time ',3)   -> 'Now is '
DELWORD('Now is the time',5)    -> 'Now is the time'
DELWORD('Now is the time',3,1)  -> 'Now is time'
```

DIGITS

►► DIGITS() ◄◄

returns the current setting of NUMERIC DIGITS. See the NUMERIC instruction on page 52 for more information.

Here is an example:

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

D2C (Decimal to Character)

►► D2C(*wholenumber* [,*n*]) ◄◄

returns a string, in character format, that represents *wholenumber*, a decimal number, converted to binary. If you specify *n*, it is the length of the final result in characters; after conversion, the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, then the result is truncated on the left. The *n* must be a positive whole number or zero.

If you omit *n*, *wholenumber* must be a positive whole number or zero, and the result length is as needed. Therefore, the returned result has no leading '00'x characters.

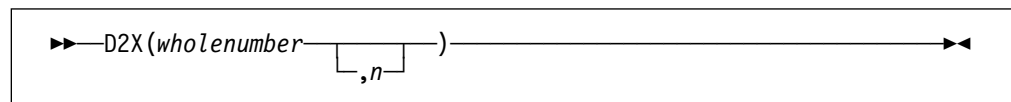
Functions

Here are some examples:

```
D2C(9)          -> ' ' /* '09'x is unprintable in EBCDIC */
D2C(129)       -> 'a' /* '81'x is an EBCDIC 'a' */
D2C(129,1)    -> 'a' /* '81'x is an EBCDIC 'a' */
D2C(129,2)    -> ' a' /* '0081'x is EBCDIC ' a' */
D2C(257,1)    -> ' ' /* '01'x is unprintable in EBCDIC */
D2C(-127,1)   -> 'a' /* '81'x is EBCDIC 'a' */
D2C(-127,2)   -> ' a' /* 'FF'x is unprintable EBCDIC;
                    /* '81'x is EBCDIC 'a' */
D2C(-1,4)     -> ' ' /* 'FFFFFFF'x is unprintable in EBCDIC */
D2C(12,0)     -> '' /* '' is a null string */
```

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

D2X (Decimal to Hexadecimal)



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

If you specify *n*, it is the length of the final result in characters; after conversion the input string is sign-extended to the required length. If the number is too big to fit into *n* characters, it is truncated on the left. The *n* must be a positive whole number or zero.

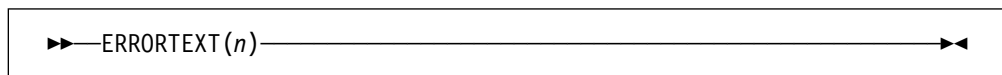
If you omit *n*, *wholenumber* must be a positive whole number or zero, and the returned result has no leading zeros.

Here are some examples:

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

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

ERRORTEXT

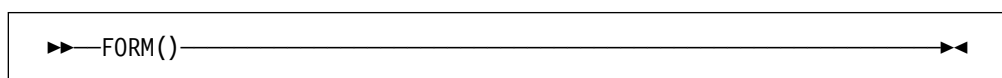


returns the REXX error message associated with error number *n*. The *n* must be in the range 0–99, and any other value is an error. Returns the null string if *n* is in the allowed range but is not a defined REXX error number.

Here are some examples:

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

FORM

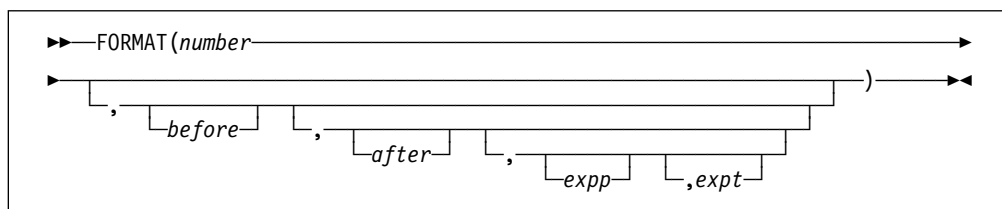


returns the current setting of NUMERIC FORM. See the NUMERIC instruction on page 52 for more information.

Here is an example:

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

FORMAT



returns *number*, rounded and formatted.

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

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

If *before* is not large enough to contain the integer part of the number (plus the sign for a negative number), an error results. If *before* is larger than needed for that part, the number is padded on the left with blanks. If *after* is not the same size as the decimal part of the number, the number is rounded (or extended with zeros) to fit. Specifying 0 causes the number to be rounded to an integer.

Functions

Here are some examples:

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

The first three arguments are as described previously. In addition, *expp* and *expt* control the exponent part of the result, which, by default, is formatted according to the current NUMERIC settings of DIGITS and FORM. The *expp* sets the number of places for the exponent part; the default is to use as many as needed (which may be zero). The *expt* sets the trigger point for use of exponential notation. The default is the current setting of NUMERIC DIGITS.

If *expp* is 0, no exponent is supplied, and the number is expressed in *simple* form with added zeros as necessary. If *expp* is not large enough to contain the exponent, an error results.

If the number of places needed for the integer or decimal part exceeds *expt* or twice *expt*, respectively, exponential notation is used. If *expt* is 0, exponential notation is always used unless the exponent would be 0. (If *expp* is 0, this overrides a 0 value of *expt*.) If the exponent would be 0 when a nonzero *expp* is specified, then *expp*+2 blanks are supplied for the exponent part of the result. If the exponent would be 0 and *expp* is not specified, simple form is used.

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

►►FUZZ()◄◄

returns the current setting of NUMERIC FUZZ. See the NUMERIC instruction on page 52 for more information.

Here is an example:

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

INSERT

►►INSERT(*new*,*target*
 ,
 n
 ,
 length
 ,
 pad
)
◄◄

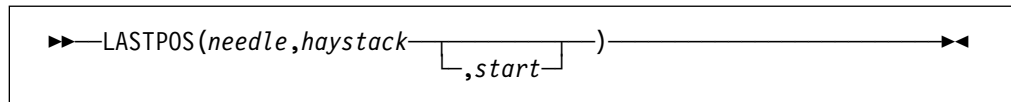
inserts the string *new*, padded or truncated to length *length*, into the string *target* after the *n*th character. The default value for *n* is 0, which means insert before the beginning of the string. If specified, *n* and *length* must be positive whole numbers or zero. If *n* is greater than the length of the target string, padding is added before the string *new* also. The default value for *length* is the length of *new*. If *length* is less than the length of the string *new*, then INSERT truncates *new* to length *length*. The default *pad* character is a blank.

Here are some examples:

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

```
INSERT('123', 'abc')         -> '123abc'
INSERT('123', 'abc', , 5, '-') -> '123--abc'
```

LASTPOS (Last Position)

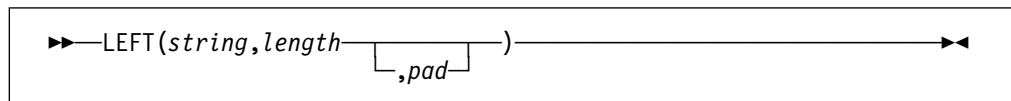


returns the position of the last occurrence of one string, *needle*, in another, *haystack*. (See also the POS function.) Returns 0 if *needle* is the null string or is not found. By default the search starts at the last character of *haystack* and scans backward. You can override this by specifying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



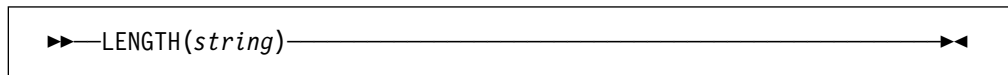
returns a string of length *length*, containing the leftmost *length* characters of *string*. The string returned is padded with *pad* characters (or truncated) on the right as needed. The default *pad* character is a blank. *length* must be a positive whole number or zero. The LEFT function is exactly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

```
↳ SUBSTR(string, 1, length, pad) ↳
```

Here are some examples:

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

LENGTH

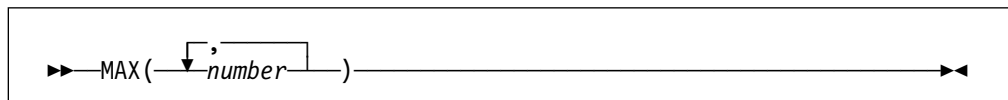


returns the length of *string*.

Here are some examples:

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

MAX (Maximum)



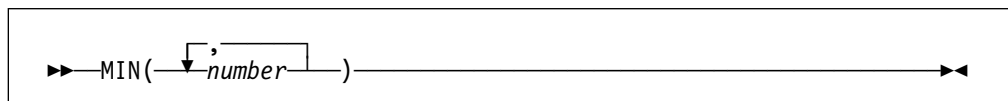
returns the largest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

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

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

MIN (Minimum)



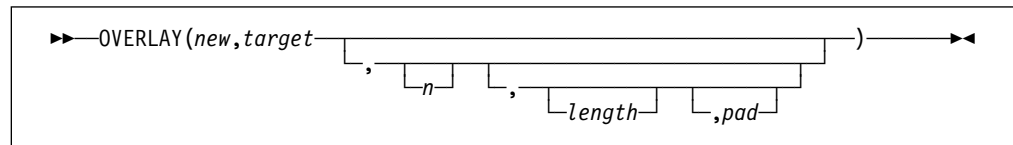
returns the smallest number from the list specified, formatted according to the current NUMERIC settings.

Here are some examples:

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

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

OVERLAY

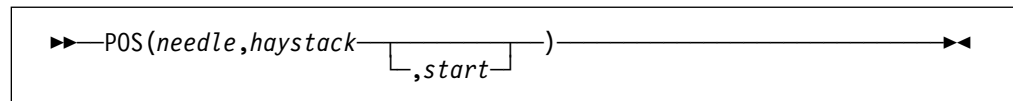


returns the string *target*, which, starting at the *n*th character, is overlaid with the string *new*, padded or truncated to length *length*. (The overlay may extend beyond the end of the original *target* string.) If you specify *length*, it must be a positive whole number or zero. The default value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

Here are some examples:

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

POS (Position)

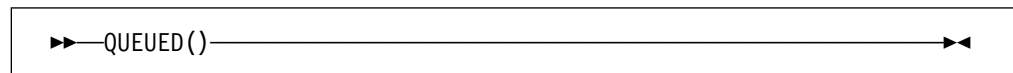


returns the position of one string, *needle*, in another, *haystack*. (See also the LASTPOS function.) Returns 0 if *needle* is the null string or is not found or if *start* is greater than the length of *haystack*. By default the search starts at the first character of *haystack* (that is, the value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

Here are some examples:

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

QUEUED

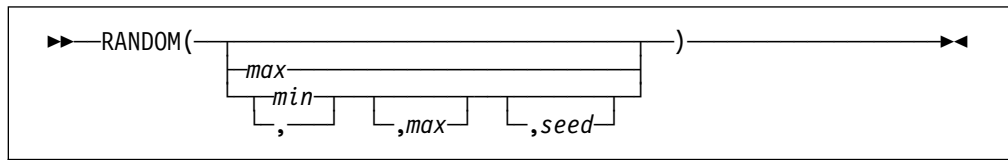


returns the number of lines remaining in the external data queue when the function is called. This includes all buffers, not just the current buffer. For more information on the queue, see “Queuing Interfaces” on page 170.

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. The *min* and *max* default to 0 and 999, respectively. To start a repeatable sequence of results, use a specific *seed* as the third argument, as described in Note 1. This *seed* must be a positive whole number ranging from 0 to 999999999.

Here are some examples:

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

Notes:

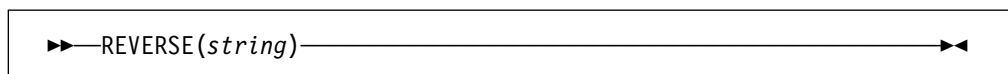
1. To obtain a predictable sequence of quasi-random numbers, use RANDOM a number of times, but specify a *seed* only the first time. For example, to simulate 40 throws of a 6-sided, unbiased die:

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

The numbers are generated mathematically, using the initial *seed*, so that as far as possible they appear to be random. Running the program again produces the same sequence; using a different initial *seed* almost certainly produces a different sequence. If you do not supply a *seed* the first time RANDOM is called, REXX/400 generates one from the time-of-day clock.

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

REVERSE



returns *string*, swapped end for end.

Here are some examples:

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

RIGHT

```
▶▶—RIGHT(string,length  
          └─┬─┘  
          ,pad)—▶▶
```

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

Here are some examples:

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

SIGN

```
▶▶—SIGN(number)—▶▶
```

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

Here are some examples:

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

SOURCELINE

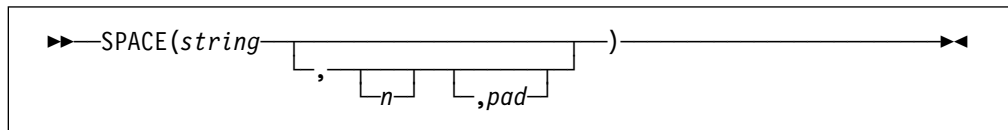
```
▶▶—SOURCELINE(—┬─  
                  └─┬─┘  
                  ,n)—▶▶
```

returns the line number of the final line in the program if you omit *n* or 0 if the implementation does not allow access to the source lines. If you specify *n*, returns the *n*th line in the program 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

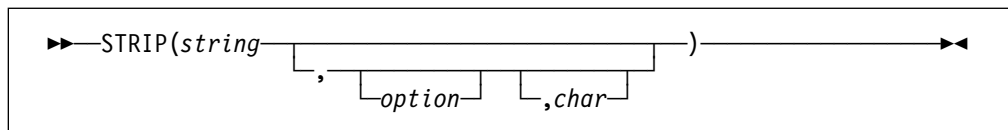


returns the blank-delimited words in *string* with *n pad* characters between each word. If you specify *n*, it must be a positive whole number or zero. If it is 0, all blanks are removed. Leading and trailing blanks are always removed. The default for *n* is 1, and the default *pad* character is a blank.

Here are some examples:

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

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 highlighted letter is needed; all characters following it are ignored.)

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

Leading removes leading characters from *string*.

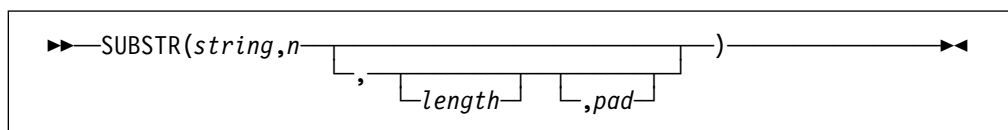
Trailing removes trailing characters from *string*.

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

Here are some examples:

```
STRIP(' abc ')      -> 'abc'
STRIP(' abc ', 'L') -> 'abc '
STRIP(' abc ', 'T') -> ' abc'
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. The *n* must be a positive whole number. If *n* is greater than `LENGTH(string)`, then only pad characters are returned.

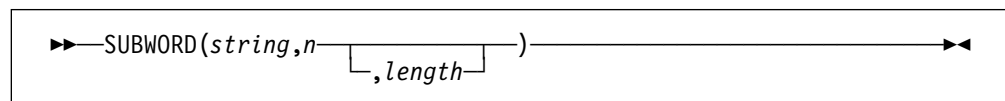
If you omit *length*, the rest of the string is returned. The default *pad* character is a blank.

Here are some examples:

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

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

SUBWORD

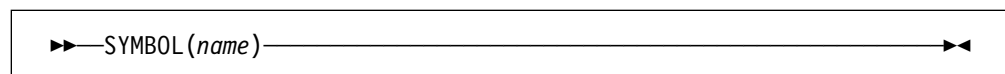


returns the substring of *string* that starts at the *n*th word, and is up to *length* blank-delimited words. The *n* must be a positive whole number. If you omit *length*, it defaults to the number of remaining words in *string*. The returned string never has leading or trailing blanks, but includes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

Here are some examples:

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

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.

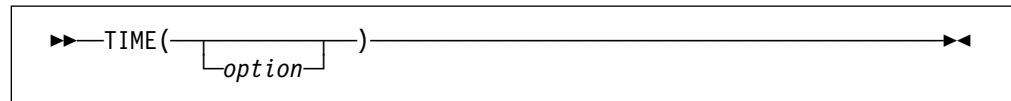
Note: You should specify *name* as a literal string (or it should be derived from an expression) to prevent substitution before it is passed to the function.

Functions

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 highlighted letter is needed; all characters following it are ignored.)

- Civil** returns the time in Civil format: hh:mmxx. The hours may take the values 1 through 12, and the minutes the values 00 through 59. The minutes are followed immediately by the letters am or pm. This distinguishes times in the morning (12 midnight through 11:59 a.m.—appearing as 12:00am through 11:59am) from noon and afternoon (12 noon through 11:59 p.m.—appearing as 12:00pm through 11:59pm). The hour has no leading zero. The minute field shows the current minute (rather than the nearest minute) for consistency with other TIME results.
- Elapsed** returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset. The number has no leading zeros or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.
- Hours** returns up to two characters giving the number of hours since midnight in the format: hh (no leading zeros or blanks, except for a result of 0).
- Long** returns time in the format: hh:mm:ss.uuuuuu (uuuuuu is the fraction of seconds, in microseconds). The first eight characters of the result follow the same rules as for the Normal form, and the fractional part is always six digits.
- Minutes** returns up to four characters giving the number of minutes since midnight in the format: mmmm (no leading zeros or blanks, except for a result of 0).
- Normal** returns the time in the default format hh:mm:ss, as described previously. The hours can have the values 00 through 23, and minutes and seconds, 00 through 59. All these are always two digits. Any fractions of seconds are ignored (times are never rounded up). **This is the default.**
- Reset** returns ssssssss.uuuuuu, the number of seconds.microseconds since the elapsed-time clock (described later) was started or reset and also resets the elapsed-time clock to zero. The number has no leading zeros

or blanks, and the setting of NUMERIC DIGITS does not affect the number. The fractional part always has six digits.

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

Note: REXX/400 provides precision to milliseconds for the L, R, and E options. Trailing zeros are added for the remaining positions.

Here are some examples, assuming that the time is 4:54 p.m.:

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

The elapsed-time clock:

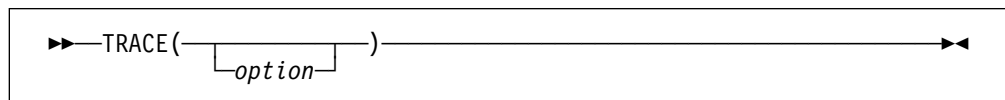
You can use the TIME function to measure real (elapsed) time intervals. On the first call in a program to TIME('E') or TIME('R'), the elapsed-time clock is started, and either call returns 0. From then on, calls to TIME('E') and to TIME('R') return the elapsed time since that first call or since the last call to TIME('R').

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

```
time('E')  -> 0 /* The first call */
/* pause of one second here */
time('E')  -> 1.020000 /* 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 usual TIME/DATE results may be calculated exactly using the elapsed-time clock.

TRACE



returns trace actions currently in effect and, optionally, alters the setting.

If you specify *option*, it selects the trace setting. It must be the valid prefix ? or one of the alphabetic character options associated with the TRACE instruction (that is, starting with A, C, E, F, I, L, N, O, or R) or both. (See the TRACE instruction on page 71 for full details.)

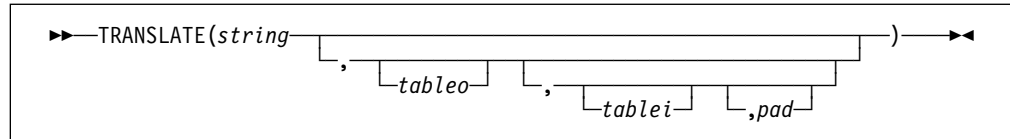
Functions

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('0')   ->  '?R' /* also sets tracing off */
TRACE('?I')  ->  '0'  /* 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*.

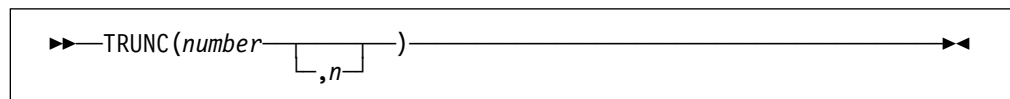
The tables can be of any length. If you specify neither translation table and omit *pad*, *string* is simply translated to uppercase (that is, lowercase a–z to uppercase A–Z), but, if you include *pad*, the language processor translates the entire string to *pad* characters. *tablei* defaults to XRANGE('00'x, 'FF'x), and *tableo* defaults to the null string and is padded with *pad* or truncated as necessary. The default *pad* is a blank.

Here are some examples:

```
TRANSLATE('abcdef')      ->  'ABCDEF'
TRANSLATE('abbc', '&', 'b') ->  'a&&c'
TRANSLATE('abcdef', '12', 'ec') ->  'ab2d1f'
TRANSLATE('abcdef', '12', 'abcd', '.') ->  '12..ef'
TRANSLATE('APQRV', ',,PR') ->  'A Q V'
TRANSLATE('APQRV', XRANGE('00'X, 'Q')) ->  'APQ '
TRANSLATE('4123', 'abcd', '1234') ->  'dabc'
```

Note: The last example shows how to use the TRANSLATE function to reorder the characters in a string. In the example, the last character of any four-character string specified as the second argument would be moved to the beginning of the string.

TRUNC (Truncate)



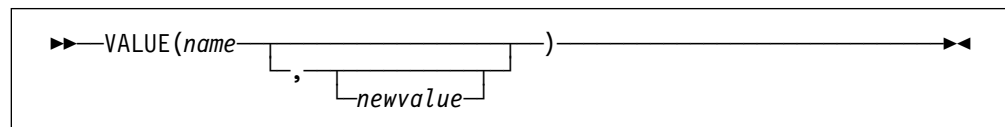
returns the integer part of *number* and *n* decimal places. The default *n* is 0 and returns an integer with no decimal point. If you specify *n*, it must be a positive whole number or zero. The *number* is first rounded according to standard REXX rules, just as though the operation *number*+0 had been carried out. The number is then truncated to *n* decimal places (or trailing zeros are added if needed to make up the specified length). The result is never in exponential form.

Here are some examples:

```
TRUNC(12.3)          -> 12
TRUNC(127.09782,3)  -> 127.097
TRUNC(127.1,3)      -> 127.100
TRUNC(127,2)        -> 127.00
```

Note: The *number* is rounded according to the current setting of NUMERIC DIGITS if necessary before the function processes it.

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. Lowercase characters in *name* are translated to uppercase. Substitution in a compound name (see “Compound Symbols” on page 22) occurs if possible.

If you specify *newvalue*, then the named variable is assigned this new value. This does not affect the result returned; that is, the function returns the value of *name* as it was before the new assignment.

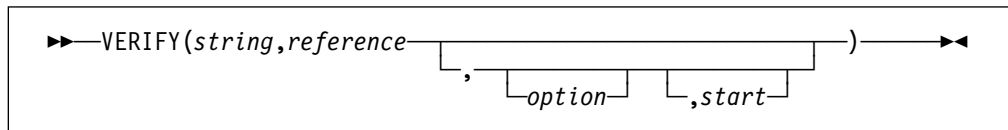
Here are some examples:

```
/* After: Drop A3; A33=7; K=3; fred='K'; list.5='Hi' */
VALUE('a'k)      -> 'A3' /* looks up A3      */
VALUE('a'k|k)    -> '7'  /* looks up A33     */
VALUE('fred')    -> 'K'  /* looks up FRED    */
VALUE(fred)      -> '3'  /* looks up K       */
VALUE(fred,5)    -> '3'  /* looks up K and   */
                  /* then sets K=5    */
VALUE(fred)      -> '5'  /* looks up K       */
VALUE('LIST.'k) -> 'Hi' /* looks up LIST.5 */
```

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.
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, 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 137.)

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 *option* can be either `nomatch` (the default) or `match`. (Only the capitalized and highlighted letter is needed. All characters following it are ignored, and it can be in upper- or lowercase, as usual.) If you specify `match`, the function returns the position of the first character in *string* that **is** in *reference*, or returns 0 if none of the characters are found.

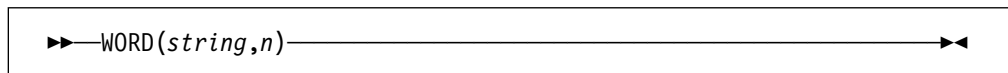
The default for *start* is 1; thus, the search starts at the first character of *string*. You can override this by specifying a different *start* point, which must be a positive whole number.

If *string* is null, the function returns 0, regardless of the value of the third argument. Similarly, if *start* is greater than `LENGTH(string)`, the function returns 0. If *reference* is null, the function returns 0 if you specify `match`; otherwise the function returns the *start* value.

Here are some examples:

<code>VERIFY('123','1234567890')</code>	<code>-></code>	<code>0</code>
<code>VERIFY('1Z3','1234567890')</code>	<code>-></code>	<code>2</code>
<code>VERIFY('AB4T','1234567890')</code>	<code>-></code>	<code>1</code>
<code>VERIFY('AB4T','1234567890','M')</code>	<code>-></code>	<code>3</code>
<code>VERIFY('AB4T','1234567890','N')</code>	<code>-></code>	<code>1</code>
<code>VERIFY('1P3Q4','1234567890',,3)</code>	<code>-></code>	<code>4</code>
<code>VERIFY('123','',N,2)</code>	<code>-></code>	<code>2</code>
<code>VERIFY('ABCDE','',,3)</code>	<code>-></code>	<code>3</code>
<code>VERIFY('AB3CD5','1234567890','M',4)</code>	<code>-></code>	<code>6</code>

WORD



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

Here are some examples:

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

WORDINDEX

▶▶—WORDINDEX(*string*,*n*)—————▶▶

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

Here are some examples:

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

WORDLENGTH

▶▶—WORDLENGTH(*string*,*n*)—————▶▶

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

Here are some examples:

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

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)

```

▶▶ XRANGE( [ start ] [ , end ] ) ◀◀
    
```

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

Here are some examples:

```

XRANGE('a','f')           ->    'abcdef'
XRANGE('03'x,'07'x)       ->    '0304050607'x
XRANGE(, '04'x)           ->    '0001020304'x
XRANGE('i','j')          ->    '898A8B8C8D8E8F9091'x /* EBCDIC */
XRANGE('FE'x,'02'x)       ->    'FEFF000102'x
XRANGE('h','i')          ->    'hi' /* EBCDIC or ASCII */
    
```

X2B (Hexadecimal to Binary)

```

▶▶ X2B(hexstring) ◀◀
    
```

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

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

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

Here are some examples:

```

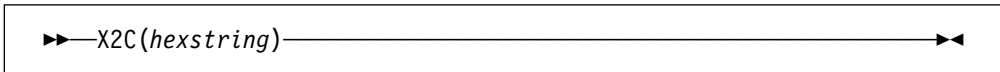
X2B('C3')                 ->    '11000011'
X2B('7')                  ->    '0111'
X2B('1 C1')               ->    '000111000001'
    
```

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

Here are some examples:

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

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. If necessary, it is padded with a leading 0 to make an even number of hexadecimal digits.

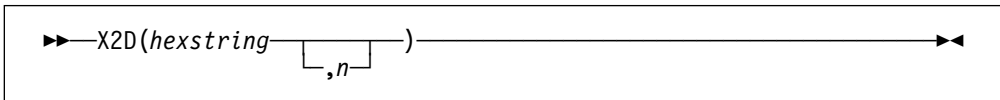
You can optionally include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

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

Here are some examples:

```
X2C('F7F2 A2') -> '72s' /* EBCDIC */
X2C('F7f2a2') -> '72s' /* EBCDIC */
X2C('F')       -> ' ' /* '0F' is unprintable EBCDIC */
```

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 include blanks in *hexstring* (at byte boundaries only, not leading or trailing) to aid readability; they are ignored.

If *hexstring* is null, the function returns 0.

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

Here are some examples:

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

If you specify *n*, the string is taken as a signed number expressed in *n* hexadecimal digits. If the leftmost bit is off, then the number is positive; otherwise, it is a negative number in two's complement notation. In both cases it is converted to a whole number, which may, therefore, be negative. If *n* is 0, the function returns 0.

Functions

If necessary, *hexstring* is padded on the left with 0 characters (note, not “sign-extended”), or truncated on the left to *n* characters.

Here are some examples:

```
X2D('81',2)    ->  -127
X2D('81',4)    ->   129
X2D('F081',4)  -> -3967
X2D('F081',3)  ->   129
X2D('F081',2)  -> -127
X2D('F081',1)  ->    1
X2D('0031',0)  ->    0
```

Implementation maximum: The *hexstring* is limited to 500 hexadecimal digits.

AS/400 System-Specific Function

The SETMSGRC function is specific to REXX/400. It is not part of the SAA definition, and therefore programs that use it may not be supported in other SAA environments.

SETMSGRC

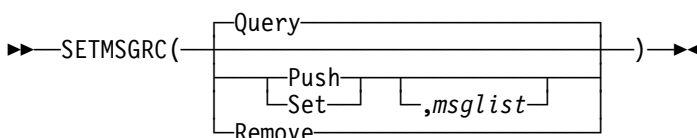
The SETMSGRC function allows you to control what REXX does with *STATUS or *NOTIFY messages sent to the language processor's program message queue while the command is running. By default, the language processor will ignore all *STATUS or *NOTIFY messages sent to its program message queue. This function is not the same as a Monitor Message (MONMSG) command, but allows you the flexibility to trap and handle *STATUS and *NOTIFY exceptions.

The SETMSGRC function can specify individual messages or a range of messages that, if received, will be returned to the program in the RC special variable. Using the SETMSGRC function causes the language processor to treat the specified *STATUS and *NOTIFY messages like *ESCAPE. This function applies to all command environments.

SETMSGRC settings are not saved and restored over function calls. Internal and external REXX functions and subroutines inherit the SETMSGRC settings of their callers, and any changes they make remain in effect when they return to their caller.

Note: This function has no effect on the handling of escape messages.

The syntax of the SETMSGRC function is:



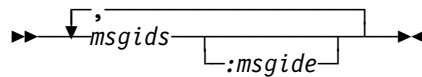
The parameters are listed below. Only the capitalized and boldfaced letter is needed; all characters following it are ignored.)

Query No action is taken. The function simply gives the return value described in “SETMSGRC Return Value” on page 109. **This is the default value.**

- Set Replaces the current list of message IDs with the list specified in the *msglist* parameter. If *msglist* is null or not specified, then the current list of IDs will be empty, that is, all *NOTIFY and *STATUS messages will be ignored.
- Push Saves the current list of message IDs and creates a new one from the *msglist* parameter, making it the current list. If *msglist* is null or not specified, then the new current list of IDs will be empty.
- Remove Discards the current list of message IDs and makes the last previously saved list the current list. If there is no previous list, the current list is discarded, the new current list is set empty, and the string *EMPTY is returned.
- msglist* String containing a set of message IDs or message ranges that are to be set into the current message ID list. The format of this parameter is shown in “Message ID List Format (Input).”

It should be noted that, while no direct interface is provided to change the current message list, the function is available to the user by using the Query parameter to retrieve the current list of messages, then using REXX string manipulation functions to change it, and finally, using either Set or Push to establish the changed list.

Message ID List Format (Input): The *msglist* parameter is a string specified in the following form:



Where:

- msgids* Specifies either a single message identifier or the starting message identifier of a range of message identifiers that will, if received, be returned in the RC variable.
- msgide* Specifies the ending message identifier of a range of message identifiers.

Ranges of messages can also be specified as single message IDs within the list, according to the Monitor Message (MONMSG) command convention. See the *CL Reference* for further information on this convention. For example, CPF0000 indicates all message identifiers beginning with CPF, while CPF1200 indicates all message identifiers beginning with CPF12. These examples would be functionally identical with the ranges CPF0000:CPFFFFFF and CPF1200:CPF12FF, respectively.

Note: The use of “wildcards” in message IDs for the MONMSG command are limited to either 2 or 4 zeros at the end of the message ID. That characteristic is respected here as well. For example, the identifier CPF2000 is equivalent to the range CPF2000:CPF20FF, **not** CPF2000:CPF2FFF as some might expect. Also, the use of explicit ranges overrides any wildcards that might occur at either end of a range (for example, the range CPF2300:CPF2300 would cause only message CPF2300 to be returned — CPF2301 through CPF23FF would be excluded).

SETMSGRC Return Value: Regardless of the input parameters, the value returned by SETMSGRC is always a string containing the current list of message IDs that will be in effect after completion of this function. The string has the same form as the **msglist** input parameter (described below), with the following caveats:

Functions

1. There will be no blanks between message IDs, commas, and colons.
2. Adjacent and overlapping ranges and single IDs will be combined. For example:

```
CPF2345:CPF3056, CPF3000, CPF2344, CPF2789, CPI1234:CPZ1234, CPZ4567
```

would be combined into

```
CPF2344:CPF30FF,CPI1234:CPZ1234,CPZ4567
```

The return value is always guaranteed to be valid as input to another SETMSGRC invocation that would set up exactly the same message list.

Note: To make sure that all *STATUS and *NOTIFY messages are returned through RC, use the following:

```
x = setmsgsrc('SE', 'AAA0000:ZZZFFF')
```

ERROR and FAILURE Conditions: When a *STATUS or *NOTIFY message is received by the language processor, either the ERROR or FAILURE conditions are raised. If the command environment to which the command was issued is *COMMAND*, the ERROR condition will be raised. In all other command environments, the FAILURE condition will be raised, since they are treated in the same way as *ESCAPE messages. For more information on these conditions see Chapter 7, “Conditions and Condition Traps” on page 137.

Chapter 5. Parsing

The parsing instructions are ARG, PARSE, and PULL (see “ARG” on page 33, “PARSE” on page 56, and “PULL” on page 62).

The data to parse is a *source string*. Parsing splits up the data in a source string and assigns pieces of it into the variables named in a template. A *template* is a model specifying how to split the source string. The simplest kind of template consists of only a list of variable names. Here is an example:

```
variable1 variable2 variable3
```

This kind of template parses the source string into blank-delimited words. More complicated templates contain patterns in addition to variable names.

String patterns Match characters in the source string to specify where to split it. (See “Templates Containing String Patterns” on page 113 for details.)

Positional patterns Indicate the character positions at which to split the source string. (See “Templates Containing Positional (Numeric) Patterns” on page 114 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 lowercase a–z in the source string to uppercase A–Z. If you want to convert characters to uppercase, use PARSE UPPER VALUE. See “Using UPPER” on page 118 for a summary of the effect of parsing instructions on case.)

All of the parsing instructions assign the parts of a source string into the variables named in a template. There are various parsing instructions because of differences in the nature or origin of source strings. (A summary of all the parsing instructions is on page 119.)

Parsing

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 plus (+) or minus (-) or equal sign (=) before the left parenthesis. (See page 118 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 121) in which a template with a string pattern does *not* omit matching data in the source string.) We used the pattern ', ' (with a blank) instead of ', ' (no blank) because, without the blank in the pattern, the variable `fn` receives ' John' (including a blank).

If the source string *does not contain a match for a string pattern*, then any variables preceding the unmatched string pattern get all the data in question. Any variables after that pattern receive the null string.

A null string is never found. It always matches the end of the source string.

Templates Containing Positional (Numeric) Patterns

A *positional pattern* is a number that identifies the character position at which to split data in the source string. The number must be a whole number.

An *absolute positional pattern* is

- A number with no plus (+) or minus (-) sign preceding it or with an equal sign (=) preceding it
- A variable in parentheses with an equal sign before the left parenthesis. (See page 118 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
FIELDS:	LASTNAME	FIRST	PSEUDONYM	end of record

The following example uses this record structure.


```

/* Parsing with absolute positional patterns in template */
record.1='Clemens Samuel Mark Twain '
record.2='Evans Mary Ann George Eliot '
record.3='Munro H.H. Saki '
do n=1 to 3
  parse var record.n lastname 11 firstname 21 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end
/* Says 'By George!' after record 2 */

```

The source string is first split at character position 11 and at position 21. The language processor assigns characters 1 to 10 into lastname, characters 11 to 20 into firstname, and characters 21 to 40 into pseudonym.

The template could have been:

```
1 lastname 11 firstname 21 pseudonym
```

instead of

```
lastname 11 firstname 21 pseudonym
```

Specifying the 1 is optional.

Optionally, you can put an equal sign before a number in a template. An equal sign is the same as no sign before a number in a template. The number refers to a particular character position in the source string. These two templates work the same:

```
lastname 11 first 21 pseudonym
```

```
lastname =11 first =21 pseudonym
```

A *relative positional pattern* is a number with a plus (+) or minus (-) sign preceding it. (It can also be a variable within parentheses, with a plus (+) or minus (-) sign preceding the left parenthesis; for details see “Parsing with Variable Patterns” on page 118.)

The number specifies the relative character position at which to split the source string. The plus or minus indicates movement right or left, respectively, from the start of the string (for the first pattern) or from the position of the last match. The position of the last match is the first character of the last match. Here is the same example as for absolute positional patterns done with relative positional patterns:

```

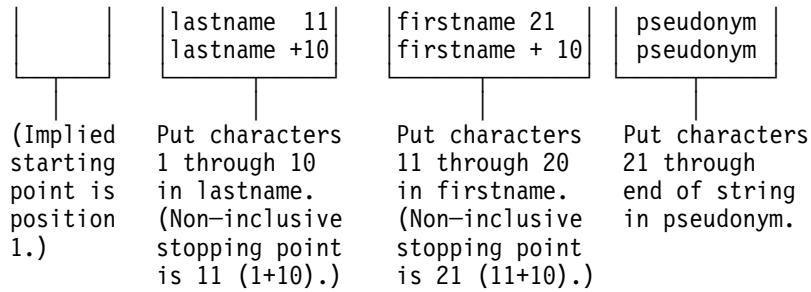
/* Parsing with relative positional patterns in template */
record.1='Clemens Samuel Mark Twain '
record.2='Evans Mary Ann George Eliot '
record.3='Munro H.H. Saki '
do n=1 to 3
  parse var record.n lastname +10 firstname + 10 pseudonym
  If lastname='Evans' & firstname='Mary Ann' then say 'By George!'
end
/* same results */

```

Blanks between the sign and the number are insignificant. Therefore, +10 and + 10 have the same meaning. Note that +0 is a valid relative positional pattern.

Absolute and relative positional patterns are interchangeable (except in the special case (on page 121) 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.

Parsing



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" */
```

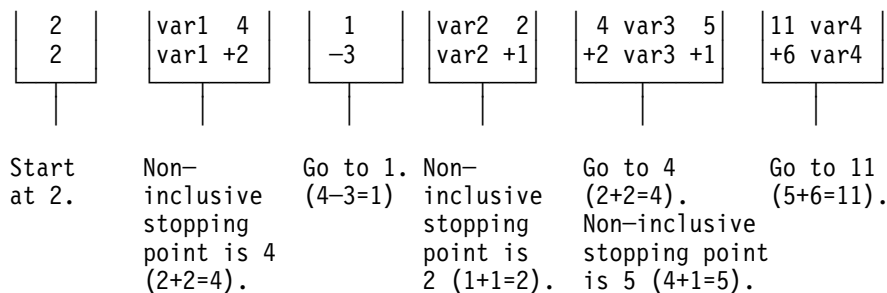
The absolute positional pattern 1 backs up to the first character in the source string.

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 112.)

```

/* 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 112.)

Parsing with Variable Patterns

You may want to specify a pattern by using the value of a variable instead of a fixed string or number. You do this by placing the name of the variable in parentheses. This is a *variable reference*. Blanks are not necessary inside or outside the parentheses, but you can add them if you wish.

The template in the next parsing instruction contains the following literal string pattern ' . '.

```
parse var name fn init ' . ' ln
```

Here is how to specify that pattern as a variable string pattern:

```
strngptrn=' . '
parse var name fn init (strngptrn) ln
```

If no equal, plus, or minus sign precedes the parenthesis that is before the variable name, the value of the variable is then treated as a string pattern. The variable can be one that has been set earlier in the same template.

Example:

```
/* Using a variable as a string pattern */
/* The variable (delim) is set in the same template */
SAY "Enter a date (mm/dd/yy format). =====> " /* assume 11/15/90 */
pull date
parse var date month 3 delim +1 day +2 (delim) year
/* Sets: month='11'; delim='/'; day='15'; year='90' */
```

If an equal, a plus, or a minus sign precedes the left parenthesis, then the value of the variable is treated as an absolute or relative positional pattern. The value of the variable must be a positive whole number or zero.

The variable can be one that has been set earlier in the same template. In the following example, the first two fields specify the starting character positions of the last two fields.

Example:

```
/* Using a variable as a positional pattern */
dataline = '12 26 .....Samuel ClemensMark Twain'
parse var dataline pos1 pos2 6 =(pos1) realname =(pos2) pseudonym
/* Assigns: realname='Samuel Clemens'; pseudonym='Mark Twain' */
```

Why is the positional pattern 6 needed in the template? Remember that word parsing occurs *after* the language processor divides the source string into substrings using patterns. Therefore, the positional pattern =(pos1) cannot be correctly interpreted as =12 until *after* the language processor has split the string at column 6 and assigned the blank-delimited words 12 and 26 to pos1 and pos2, respectively.

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 ARG
PARSE UPPER ARG	
PARSE UPPER LINEIN	PARSE LINEIN
PULL	PARSE PULL
PARSE UPPER 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	Arguments you list when you call the program or arguments in the call to a subroutine or function.
PARSE ARG	
PARSE LINEIN	Next line in the default input stream.
PULL	The string at the head of the external data queue. (If queue empty, uses default input, typically the terminal.)
PARSE PULL	
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 <i>name</i>	Parses the value of <i>name</i> .
PARSE VERSION	System-supplied string specifying 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 REXX program invocation */
/* Program name is PALETTE. Specify 2 primary colors (yellow, */
/* red, blue) on call. Assume call is: palette red blue */
arg var1 var2 /* Assigns: var1='RED'; var2='BLUE' */
If var1<>'RED' & var1<>'YELLOW' & var1<>'BLUE' then signal err
If var2<>'RED' & var2<>'YELLOW' & var2<>'BLUE' then signal err
total=length(var1)+length(var2)
SELECT;
  When total=7 then new='purple'
  When total=9 then new='orange'
  When total=10 then new='green'
  Otherwise new=var1 /* entered duplicates */
END
Say new; exit /* Displays: "purple" */

Err:
say 'Input error--color is not "red" or "blue" or "yellow"'; exit

```

ARG converts alphabetic characters to uppercase before parsing. An example of ARG with the arguments in the CALL to a subroutine is in “Parsing Multiple Strings” on page 121.

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 'c=' 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/400" */

```

PARSE VALUE example is on page 111.

PARSE VAR examples are throughout the chapter, starting on page 112.

PARSE VERSION

```

parse version . level .
say level /* Displays: "3.48" */

```

PULL works the same as PARSE PULL except that PULL converts alphabetic characters to uppercase before parsing.

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 call a program or CALL a subroutine or function.) Each comma is an instruction to the parser to move on to the next string.

Example:

```
/* Parsing multiple strings in a subroutine          */
num='3'
musketeers="Porthos Athos Aramis D'Artagnon"
CALL Sub num,musketeers /* Passes num and musketeers to sub */
SAY total; say fourth /* Displays: "4" and " D'Artagnon"    */
EXIT
```

```
Sub:
parse arg subtotal, . . . fourth
total=subtotal+1
RETURN
```

Note that when a REXX program is started as a command, only one argument string is recognized. You can pass multiple argument strings for parsing:

- When one REXX program calls another REXX program with the CALL instruction or a function call.
- When programs written in other languages start a REXX program.

If there are more templates than source strings, each variable in a leftover template receives a null string. If there are more source strings than templates, the language processor ignores leftover source strings. If a template is empty (two commas in a row) or contains no variable names, parsing proceeds to the next template and source string.

Combining String and Positional Patterns: A Special Case

There is a special case in which absolute and relative positional patterns do not work identically. We have shown how parsing with a template containing a string pattern skips over the data in the source string that matches the pattern (see page 114). But a template containing the sequence:

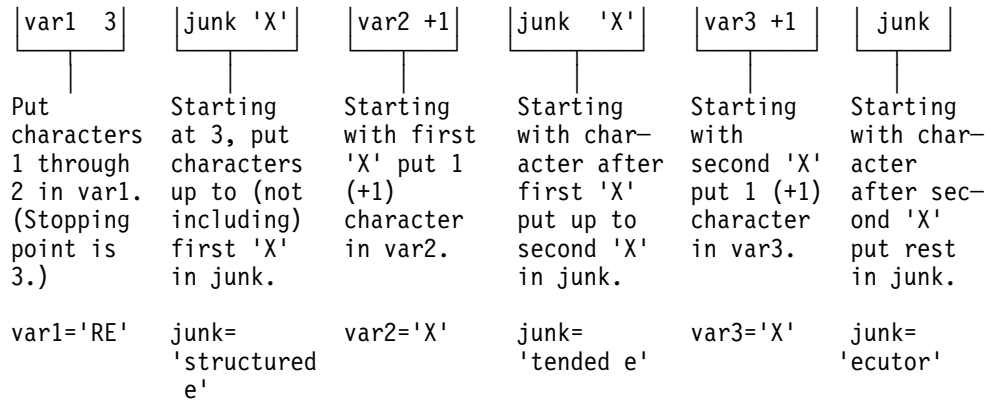
- string pattern
- variable name
- *relative* positional pattern

does *not* skip over the matching data. A relative positional pattern moves relative to the first character matching a string pattern. As a result, assignment includes the data in the source string that matches the string pattern.

Parsing

```
/* 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 must be in SBCS characters. See "PARSE" on page 180 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.

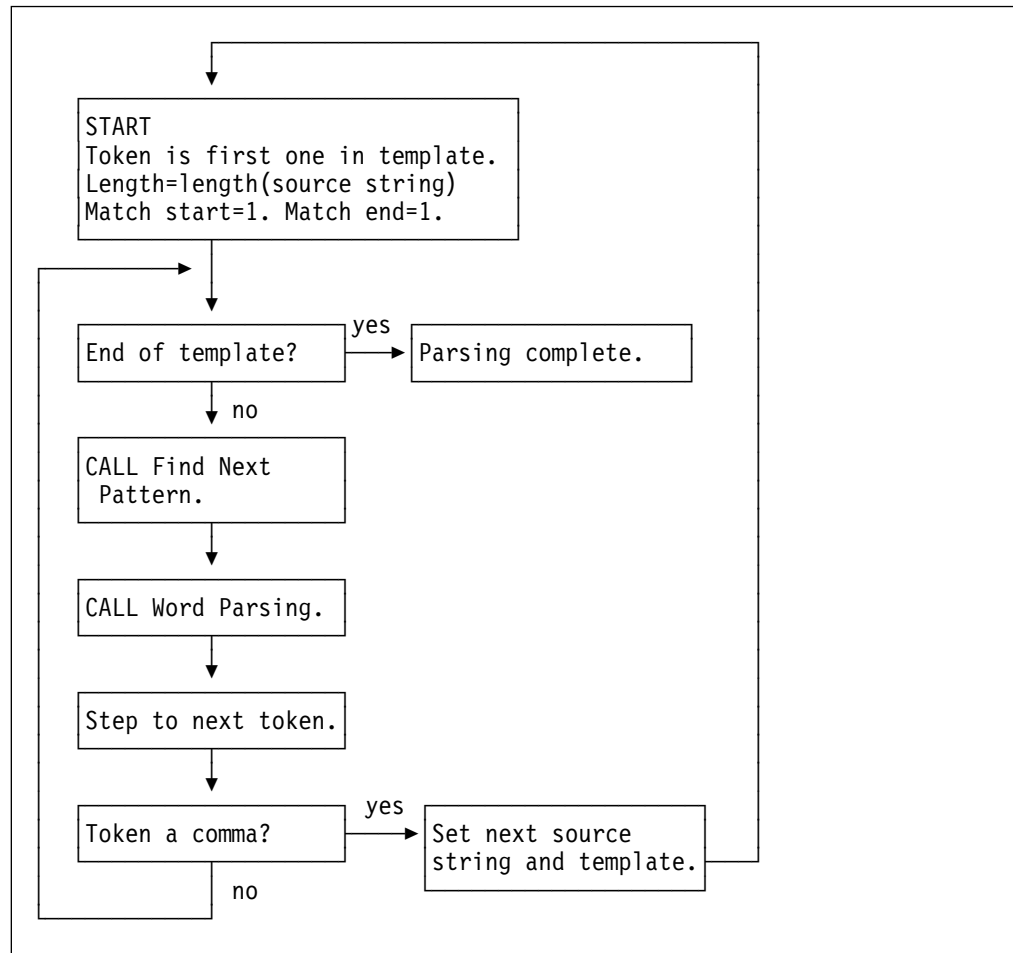


Figure 2. Conceptual Overview of Parsing

Parsing

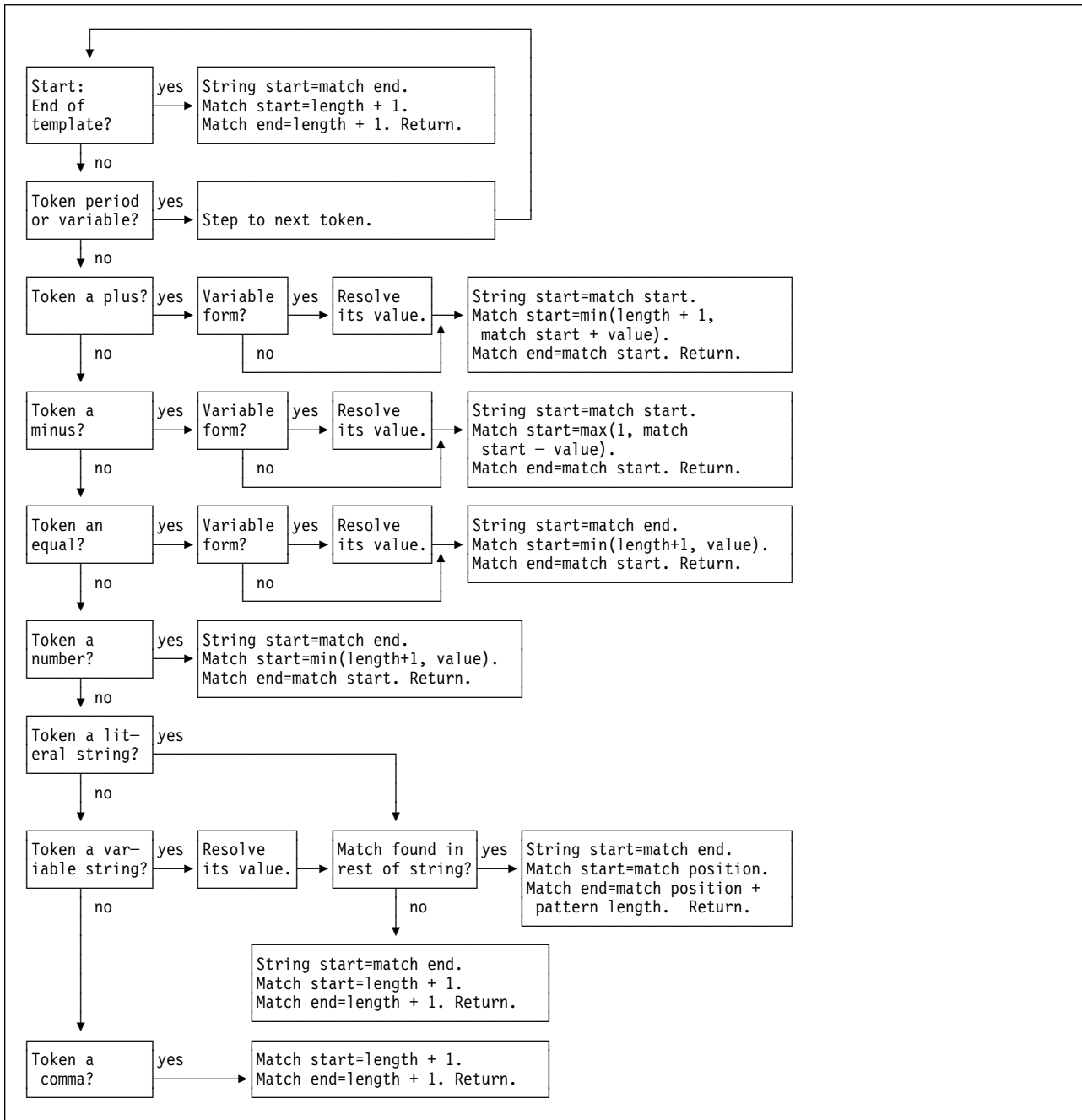


Figure 3. Conceptual View of Finding Next Pattern

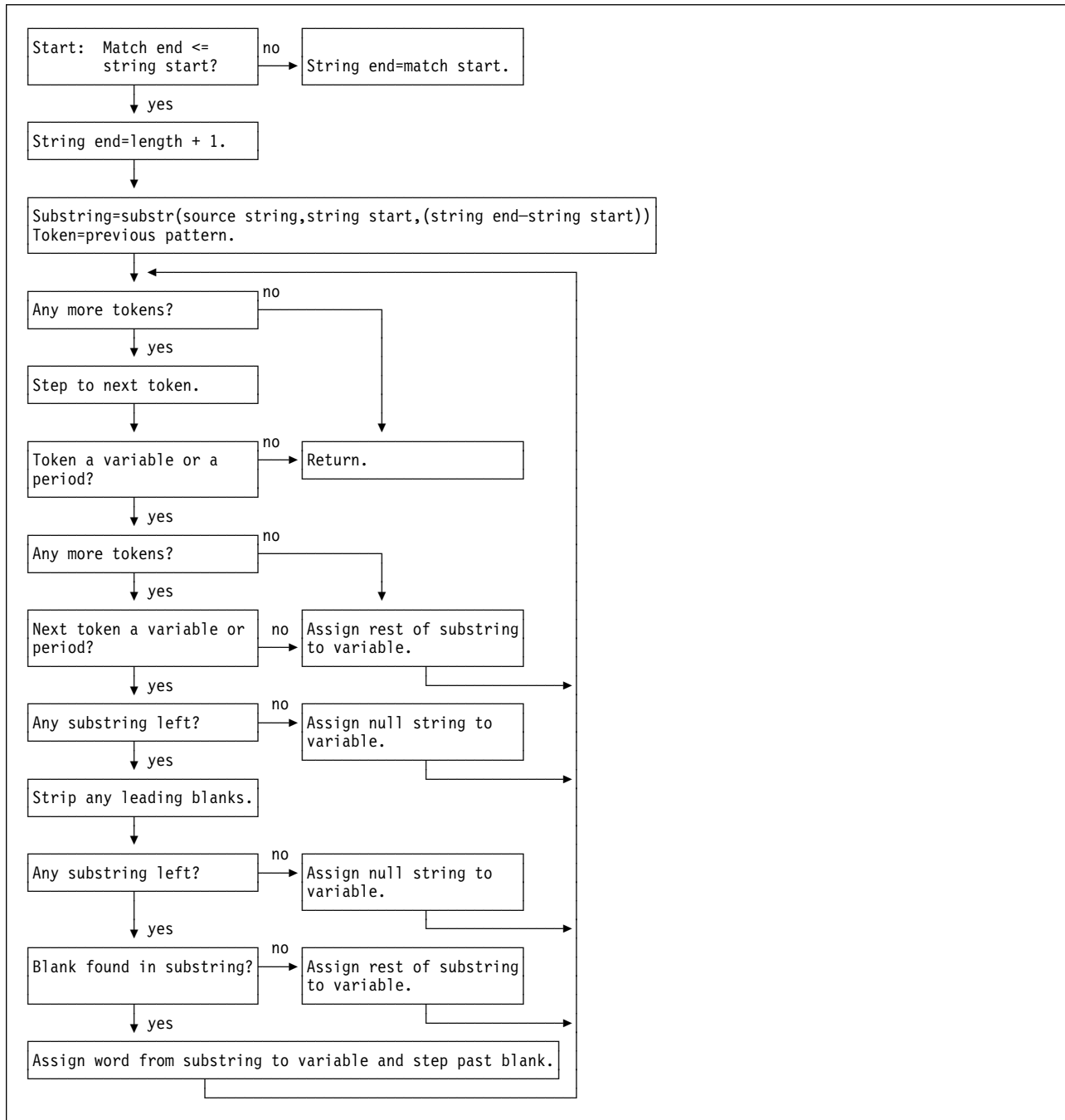


Figure 4. Conceptual View of Word Parsing

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 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          */

```

In exponential notation, a number includes an exponent, a power of ten by which the number is multiplied before use. The exponent indicates how the decimal point is shifted. Thus, in the preceding examples, 4E9 is simply a short way of writing 4000000000, and 0.73e-7 is short for 0.000000073.

The **arithmetic operators** include addition (+), subtraction (-), multiplication (*), power (**), division (/), prefix plus (+), and prefix minus (-). In addition, there are two further division operators: integer divide (%) divides and returns the integer part; remainder (//) divides and returns the remainder.

The result of an arithmetic operation is formatted as a character string according to definite rules. The most important of these rules are as follows (see the "Definition" section for full details):

- Results are calculated up to some maximum number of significant digits (the default is 9, but you can alter this with the NUMERIC DIGITS instruction to give whatever accuracy you need). Thus, if a result requires more than 9 digits, it would usually be rounded to 9 digits. For example, the division of 2 by 3 would result in 0.66666667 (it would require an infinite number of digits for perfect accuracy).
- Except for division and power, trailing zeros are preserved (this is in contrast to most popular calculators, which remove all trailing zeros in the decimal part of results). So, for example:

```

2.40 + 2    ->  4.40
2.40 - 2    ->  0.40
2.40 * 2    ->  4.80
2.40 / 2    ->  1.2

```

Numbers and Arithmetic

This behavior is desirable for most calculations (especially financial calculations).

If necessary, you can remove trailing zeros with the STRIP function (see page 98), or by division by 1.

- A zero result is always expressed as the single digit 0.
- Exponential form is used for a result depending on its value and the setting of NUMERIC DIGITS (the default is 9). If the number of places needed before the decimal point exceeds the NUMERIC DIGITS setting, or the number of places after the point exceeds twice the NUMERIC DIGITS setting, the number is expressed in exponential notation:

```
1e6 * 1e6    ->    1E+12          /* not 1000000000000 */
1 / 3E10     ->    3.33333333E-11 /* not 0.000000000033333333 */
```

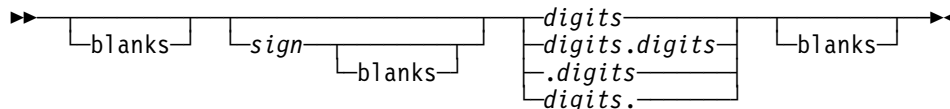
Definition

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 133 for an extension of this definition.) The decimal point may be embedded in the number, or may be a prefix or suffix. The group of digits (and optional decimal point) constructed this way can have leading or trailing blanks and an optional sign (+ or -) that must come before any digits or decimal point. The sign can also have leading or trailing blanks.

Therefore, **number** is defined as:



blanks

are one or more spaces

sign

is either + or -

digits

are one or more of the decimal digits 0–9.

Note that a single period alone is not a valid number.

Precision

Precision is the maximum number of significant digits that can result from an operation. This is controlled by the instruction:

```
NUMERIC DIGITS expression ;
```

The *expression* is evaluated and must result in a positive whole number. This defines the precision (number of significant digits) to which calculations are carried out. Results are rounded to that precision, if necessary.

If you do not specify *expression* in this instruction, or if no NUMERIC DIGITS instruction has been processed since the start of a program, the default precision is used. The REXX standard for the default precision is 9.

Note that NUMERIC DIGITS can set values below the default of nine. However, use small values with care—the loss of precision and rounding thus requested affects all REXX computations, including, for example, the computation of new values for the control variable in DO loops.

Arithmetic Operators

REXX arithmetic is performed by the operators +, -, *, /, %, //, and ** (add, subtract, multiply, divide, integer divide, remainder, and power), which all act on two terms, and the prefix plus and minus operators, which both act on a single term. This section describes the way in which these operations are carried out.

Before every arithmetic operation, the term or terms being operated upon have leading zeros removed (noting the position of any decimal point, and leaving only one zero if all the digits in the number are zeros). They are then truncated (if necessary) to DIGITS + 1 significant digits before being used in the computation. (The extra digit is a “guard” digit. It improves accuracy because it is inspected at the end of an operation, when a number is rounded to the required precision.) The operation is then carried out under up to double that precision, as described under the individual operations that follow. When the operation is completed, the result is rounded if necessary to the precision specified by the NUMERIC DIGITS instruction.

Rounding is done in the traditional manner. The digit to the right of the least significant digit in the result (the “guard digit”) is inspected and values of 5 through 9 are rounded up, and values of 0 through 4 are rounded down. Even/odd rounding would require the ability to calculate to arbitrary precision at all times and is, therefore, not the mechanism defined for REXX.

A conventional zero is supplied in front of the decimal point if otherwise there would be no digit before it. Significant trailing zeros are retained for addition, subtraction, and multiplication, according to the rules that follow, except that a result of zero is always expressed as the single digit 0. For division, insignificant trailing zeros are removed after rounding.

The FORMAT built-in function (see page 91) 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.

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:

$$\begin{array}{r} xxx.xxx00 \\ + 0yy.yyyyy \\ \hline zzz.zzzzz \end{array}$$

The result is then rounded to the current setting of NUMERIC DIGITS if necessary (taking into account any extra “carry digit” on the left after addition, but otherwise counting from the position corresponding to the most significant digit of the terms being added or subtracted). Finally, any insignificant leading zeros are removed.

The prefix operators are evaluated using the same rules; the operations +number and -number are calculated as 0+number and 0-number, respectively.

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:

$$zzzz.zzzzzzzz$$

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:

$$\begin{array}{r} zzzz \\ xxxxx \overline{) yyy00} \end{array}$$

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 afterward. Therefore, the significant digits of a result are not in any way dependent on the position of the decimal point in either of the terms involved in the operation.

Arithmetic Operation Rules—Additional Operators

The operation rules for the power (**), integer divide (%), and remainder (//) operators 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. (The second term in the operation must be a whole number and is rounded to DIGITS digits, if necessary, as described under “Numbers Used Directly by REXX” on page 135.) If negative, the absolute value of the power is used, and then the result is inverted (divided into 1). For calculating the power, the number is effectively multiplied by itself for the number of times expressed by the power, and finally trailing zeros are removed (as though the result were divided by 1).

In practice (see Note 1 on page 132 for the reasons), the power is calculated by the process of left-to-right binary reduction. For $a^{**}n$: n is converted to binary, and a temporary accumulator is set to 1. If $n = 0$ the initial calculation is complete. (Thus, $a^{**}0 = 1$ for all a , including $0^{**}0$.) Otherwise each bit (starting at the first nonzero bit) is inspected from left to right. If the current bit is 1, the accumulator is multiplied by a . If all bits have now been inspected, the initial calculation is complete; otherwise the accumulator is squared and the next bit is inspected for multiplication. When the initial calculation is complete, the temporary result is divided into 1 if the power was negative.

The multiplications and division are done under the arithmetic operation rules, using a precision of DIGITS + L + 1 digits. L is the length in digits of the integer part of the whole number n (that is, excluding any decimal part, as though the built-in function TRUNC(n) had been used). Finally, the result is rounded to NUMERIC DIGITS digits, if necessary, and insignificant trailing zeros are removed.

Integer Division

The **% (integer divide) operator** divides two numbers and returns the integer part of the result. The result returned is defined to be that which would result from repeatedly subtracting the divisor from the dividend while the dividend is larger than the divisor. During this subtraction, the absolute values of both the dividend and the divisor are used: the sign of the final result is the same as that which would result from regular division.

The result returned has no fractional part (that is, no decimal point or zeros following it). If the result cannot be expressed as a whole number, the operation is in error and will fail—that is, the result must not have more digits than the current setting of NUMERIC DIGITS. For example, `10000000000%3` requires 10 digits for the result (3333333333) and would, therefore, fail if NUMERIC DIGITS 9 were in effect.

Remainder

The **// (remainder) operator** returns the remainder from integer division and is defined as being the residue of the dividend after the operation of calculating integer division as previously described. The sign of the remainder, if nonzero, is the same as that of the original dividend.

This operation fails under the same conditions as integer division (that is, if integer division on the same two terms would fail, the remainder cannot be calculated).

Additional Operator Examples

Following are some examples using the power, integer divide, and remainder operators:

```
/* Again with: Numeric digits 5 */
2**3      ->    8
2**-3     ->   0.125
1.7**8    ->  69.758
2%3       ->    0
2.1//3    ->   2.1
10%3      ->    3
10//3     ->    1
-10//3    ->   -1
10.2//1   ->   0.2
10//0.3   ->   0.1
3.6//1.3  ->   1.0
```

Notes:

1. A particular algorithm for calculating powers is used, because it is efficient (though not optimal) and considerably reduces the number of actual multiplications performed. It, therefore, gives better performance than the simpler definition of repeated multiplication. Because results may differ from those of repeated multiplication, the algorithm is defined here.
2. The integer divide and remainder operators are defined so that they can be calculated as a by-product of the standard division operation. The division process is ended as soon as the integer result is available; the residue of the dividend is the remainder.

Numeric Comparisons

The comparison operators are listed in “Comparison” on page 17. You can use any of these for comparing numeric strings. However, you should not use `==`, `\==`, `-==`, `>>`, `\>>`, `->>`, `<<`, `\<<`, and `-<<` for comparing numbers because leading and trailing blanks and leading zeros are significant with these operators.

A comparison of numeric values is effected by subtracting the two numbers (calculating the difference) and then comparing the result with 0. That is, the operation:

`A ? Z`

where `?` is any numeric comparison operator, is identical with:

`(A - Z) ? '0'`

It is, therefore, the *difference* between two numbers, when subtracted under REXX subtraction rules, that determines their equality.

A quantity called **fuzz** affects the comparison of two numbers. This controls the amount by which two numbers may differ before being considered equal for the purpose of comparison. The FUZZ value is set by the instruction:

```
►—NUMERIC FUZZ—┬──────────┴──►;►►
                  └expression┘
```

Here *expression* must result in a positive whole number or zero. The default is 0.

The effect of FUZZ is to temporarily reduce the value of DIGITS by the FUZZ value for each numeric comparison. That is, the numbers are subtracted under a precision of DIGITS minus FUZZ digits during the comparison. Clearly the FUZZ setting must be less than DIGITS.

Thus if DIGITS = 9 and FUZZ = 1, the comparison is carried out to 8 significant digits, just as though NUMERIC DIGITS 8 had been put in effect for the duration of the operation.

Example:

```
Numeric digits 5
Numeric fuzz 0
say 4.9999 = 5      /* Displays "0"    */
say 4.9999 < 5     /* Displays "1"    */
Numeric fuzz 1
say 4.9999 = 5     /* Displays "1"    */
say 4.9999 < 5     /* Displays "0"    */
```

Exponential Notation

The preceding description of numbers describes “pure” numbers, in the sense that the character strings that describe numbers can be very long. For example:

`10000000000 * 10000000000`

would give

`100000000000000000000`

and

`.00000000001 * .00000000001`

Numbers and Arithmetic

would give

```
0.000000000000000000000001
```

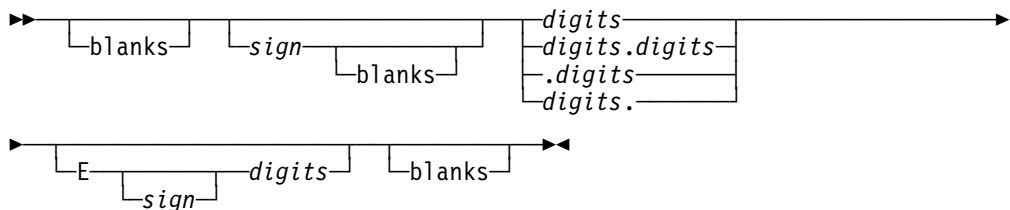
For both large and small numbers some form of exponential notation is useful, both to make long numbers more readable, and to make execution possible in extreme cases. In addition, exponential notation is used whenever the “simple” form would give misleading information.

For example:

```
numeric digits 5  
say 54321*54321
```

would display 2950800000 in long form. This is clearly misleading, and so the result is expressed as 2.9508E+9 instead.

The definition of numbers is, therefore, extended as:



The integer following the E represents a power of ten that is to be applied to the number. The E can be in uppercase or lowercase.

Certain character strings are numbers even though they do not appear to be numeric to the user. Specifically, because of the format of numbers in exponential notation, strings, such as 0E123 (0 raised to the 123 power) and 1E342 (1 raised to the 342 power), are numeric. In addition, a comparison such as 0E123=0E567 gives a true result of 1 (0 is equal to 0). To prevent problems when comparing nonnumeric strings, use the strict comparison operators.

Here are some examples:

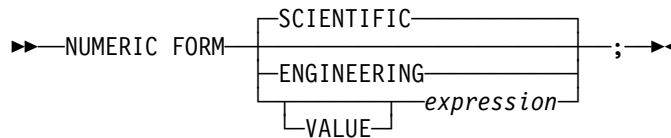
```
12E7   =   120000000      /* Displays "1" */  
12E-5  =   0.00012       /* Displays "1" */  
-12e4  =  -120000       /* Displays "1" */  
0e123  =   0e456        /* Displays "1" */  
0e123  ==  0e456        /* Displays "0" */
```

The preceding numbers are valid for input data at all times. The results of calculations are returned in either conventional or exponential form, depending on the setting of NUMERIC DIGITS. If the number of places needed before the decimal point exceeds DIGITS, or the number of places after the point exceeds twice DIGITS, exponential form is used. The exponential form REXX generates always has a sign following the E to improve readability. If the exponent is 0, then the exponential part is omitted—that is, an exponential part of E+0 is never generated.

You can explicitly convert numbers to exponential form, or force them to be displayed in long form, by using the FORMAT built-in function (see page 91).

Scientific notation is a form of exponential notation that adjusts the power of ten so a single nonzero digit appears to the left of the decimal point. **Engineering notation** is a form of exponential notation in which from one to three digits (but not

simply 0) appear before the decimal point, and the power of ten is always expressed as a multiple of three. The integer part may, therefore, range from 1 through 999. You can control whether Scientific or Engineering notation is used with the instruction:



Scientific notation is the default.

```
/* after the instruction */
Numeric form scientific
```

```
123.45 * 1e11    ->    1.2345E+13
```

```
/* after the instruction */
Numeric form engineering
```

```
123.45 * 1e11    ->    12.345E+12
```

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, the result of any arithmetic operation is rounded (if necessary) according to the setting of NUMERIC DIGITS. Similarly, when REXX directly uses a number (which has not necessarily been involved in an arithmetic operation), the same rounding is also applied. It is just as though the number had been added to 0.

In the following cases, the number used must be a whole number, and the largest number you can use is 999999999.

- The positional patterns in parsing templates (including variable positional patterns)
- The power value (right hand operand) of the power operator
- The values of *expr* and *exprf* in the DO instruction
- The values given for DIGITS or FUZZ in the NUMERIC instruction
- Any number used in the numeric option in the TRACE instruction.

Notes:

1. FUZZ must always be less than DIGITS.
2. Values of *expr* and *exprf* in the DO instruction are limited by the current numeric precision.

Errors

Two types of errors may occur during arithmetic:

- Overflow or Underflow

This error occurs if the exponential part of a result would exceed the range that the language processor can handle, when the result is formatted according to the current settings of NUMERIC DIGITS and NUMERIC FORM. The language defines a minimum capability for the exponential part, namely the largest number that can be expressed as an exact integer in default precision. Because the default precision is 9, AS/400 supports exponents in the range -999999999 through 999999999.

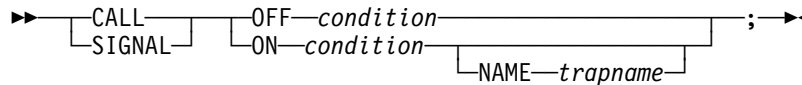
Because this allows for (very) large exponents, overflow or underflow is treated as a syntax error.

- Insufficient storage

Storage is needed for calculations and intermediate results, and on occasion an arithmetic operation may fail because of lack of storage. This is considered a terminating error as usual, rather than an arithmetic error.

Chapter 7. Conditions and Condition Traps

A **condition** is a specified event or state that CALL ON or SIGNAL ON can trap. A condition trap can modify the flow of execution in a REXX program. Condition traps are turned on or off using the ON or OFF subkeywords of the SIGNAL and CALL instructions (see “CALL” on page 35 and “SIGNAL” on page 68).



condition and *trapname* are single symbols that are taken as constants. Following one of these instructions, a condition trap is set to either ON (enabled) or OFF (disabled). The initial setting for all condition traps is OFF.

If a condition trap is enabled and the specified *condition* occurs, control passes to the routine or label *trapname* if you have specified *trapname*. Otherwise, control passes to the routine or label *condition*. CALL or SIGNAL is used, depending on whether the most recent trap for the condition was set using CALL ON or SIGNAL ON, respectively.

Note: If you use CALL, the *trapname* can be an internal label, a built-in function, or an external routine. If you use SIGNAL, the *trapname* can be only an internal label.

The conditions and their corresponding events that can be trapped are:

ERROR

raised if a command indicates an error condition upon return. It is also raised if any command indicates failure and neither CALL ON FAILURE nor SIGNAL ON FAILURE is active. The condition is raised at the end of the clause that called the command but is ignored if the ERROR condition trap is already in the delayed state. The **delayed state** is the state of a condition trap when the condition has been raised but the trap has not yet been reset to the enabled (ON) or disabled (OFF) state.

For more information on trapping command errors, see “Command Interface” on page 153.

FAILURE

raised if a command indicates a failure condition upon return. The condition is raised at the end of the clause that called the command but is ignored if the FAILURE condition trap is already in the delayed state.

For more information on trapping command failures, see “Command Interface” on page 153.

HALT

raised if an external attempt is made to interrupt and end execution of the program. The condition is usually raised at the end of the clause that was being processed when the external interruption occurred. This option is not implemented for REXX programs running in the default system environment. In such files, the SIGNAL ON HALT instruction is ignored.

Conditions and Condition Traps

Note: Application programs that use the REXX language processor may specify the RXHLT exit to stop execution of a REXX program. See “System Exit Interfaces” on page 157.

NOVALUE

raised if an uninitialized variable is used:

- As a term in an expression
- As the *name* following the VAR subkeyword of a PARSE instruction
- As a variable reference in a parsing template, a PROCEDURE instruction, or a DROP instruction.

Note: SIGNAL ON NOVALUE can trap any uninitialized variables except tails in compound variables.

```
/* The following does not raise NOVALUE. */
signal on novalue
a.=0
say a.z
say 'NOVALUE is not raised.'
exit

novalue:
say 'NOVALUE is raised.'
```

You can specify this condition only for SIGNAL ON.

SYNTAX

raised if any language processing error is detected while the program is running. This includes all kinds of processing errors, including true syntax errors and “run-time” errors, such as attempting an arithmetic operation on nonnumeric terms. You can specify this condition only for SIGNAL ON.

Any ON or OFF reference to a condition trap replaces the previous state (ON, OFF, or DELAY, and any *trapname*) of that condition trap. Thus, a CALL ON HALT replaces any current SIGNAL ON HALT (and a SIGNAL ON HALT replaces any current CALL ON HALT), a CALL ON or SIGNAL ON with a new trap name replaces any previous trap name, any OFF reference disables the trap for CALL or SIGNAL, and so on.

Action Taken When a Condition Is Not Trapped

When a condition trap is currently disabled (OFF) and the specified condition occurs, the default action depends on the condition:

- For HALT and SYNTAX, the processing of the program ends, and a message 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 processed automatically. You can specify the *trapname* after the NAME subkeyword of the CALL ON or SIGNAL ON instruction. If you do not

specify a *trapname*, the name of the condition itself (ERROR, FAILURE, HALT, NOVALUE, or SYNTAX) is used.

For example, the instruction `call on error` enables the condition trap for the ERROR condition. If the condition occurred, then a call to the routine identified by the name ERROR is made. The instruction `call on error name commanderror` would enable the trap and call the routine COMMANDERROR if the condition occurred.

The sequence of events, after a condition has been trapped, varies depending on whether a SIGNAL or CALL is processed:

- If the action taken is a SIGNAL, execution of the current instruction ceases immediately, the condition is disabled (set to OFF), and the SIGNAL takes place in exactly the same way as usual (see page 68).

If any new occurrence of the condition is to be trapped, a new CALL ON or SIGNAL ON instruction for the condition is required to re-enable it when the label is reached. For example, if SIGNAL ON SYNTAX is enabled when a SYNTAX condition occurs, then, if the SIGNAL ON SYNTAX label name is not found, a usual syntax error termination occurs.

- If the action taken is a CALL (which can occur only at a clause boundary), the CALL is made in the usual way (see page 35) except that the call does not affect the special variable RESULT. If the routine should RETURN any data, then the returned character string is ignored.

Because these conditions (ERROR, FAILURE, and HALT) can arise during execution of an INTERPRET instruction, execution of the INTERPRET may be interrupted and later resumed if CALL ON was used.

As the condition is raised, and before the CALL is made, the condition trap is put into a delayed state. This state persists until the RETURN from the CALL, or until an explicit CALL (or SIGNAL) ON (or OFF) is made for the condition. This delayed state prevents a premature condition trap at the start of the routine called to process a condition trap. When a condition trap is in the delayed state it remains enabled, but if the condition is raised again, it is either ignored (for ERROR or FAILURE) or (for the other conditions) any action (including the updating of the condition information) is delayed until one of the following events occurs:

1. A CALL ON or SIGNAL ON, for the delayed condition, is processed. In this case a CALL or SIGNAL takes place immediately after the new CALL ON or SIGNAL ON instruction has been processed.
2. A CALL OFF or SIGNAL OFF, for the delayed condition, is processed. In this case the condition trap is disabled and the default action for the condition occurs at the end of the CALL OFF or SIGNAL OFF instruction.
3. A RETURN is made from the subroutine. In this case the condition trap is no longer delayed and the subroutine is called again immediately.

On RETURN from the CALL, the original flow of execution is resumed (that is, the flow is not affected by the CALL).

Notes:

1. In all cases, the condition is raised immediately upon detection. If SIGNAL ON traps the condition, the current instruction is ended, if necessary. Therefore, the instruction during which an event occurs may be only partly processed. For example, if SYNTAX is raised during the evaluation of the expression in an assignment, the assignment does not take place. Note that the CALL for ERROR, FAILURE, and HALT 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 DELAY, and any *trapname*) of each condition trap is saved on entry to a subroutine and is then restored on RETURN. This means that CALL ON, CALL OFF, SIGNAL ON, and SIGNAL OFF can be used in a subroutine without affecting the conditions set up by the caller. See the CALL instruction (page 35) 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 called by a CALL, even if the external routine is a REXX program. On entry to any REXX program, all condition traps have an initial setting of OFF.
4. While user input is processed during interactive tracing, all condition traps are temporarily set OFF. This prevents any unexpected transfer of control—for example, should the user accidentally use an uninitialized variable while SIGNAL ON NOVALUE is active. For the same reason, a syntax error during interactive tracing does not cause exit from the program but is trapped specially and then ignored after a message is given.
5. The system interface detects certain execution errors either before execution of the program starts or after the program has ended. SIGNAL ON SYNTAX cannot trap these errors.

Note that a **label** is a clause consisting of a single symbol followed by a colon. Any number of successive clauses can be labels; therefore, multiple labels are allowed before another type of clause.

Condition Information

When any condition is trapped and causes a SIGNAL or CALL, this becomes the current trapped condition, and certain condition information associated with it is recorded. You can inspect this information by using the CONDITION built-in function (see page 84).

The condition information includes:

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

The current condition information is replaced when control is passed to a label as the result of a condition trap (CALL ON or SIGNAL ON). Condition information is

saved and restored across subroutine or function calls, including one because of a CALL ON trap. Therefore, a routine called by a CALL ON can access the appropriate condition information. Any previous condition information is still available after the routine returns.

Descriptive Strings

The descriptive string varies, depending on the condition trapped.

ERROR	The string that was processed and resulted in the error condition.
FAILURE	The string that was processed and resulted in the failure condition.
HALT	Any string associated with the halt request. This can be the null string if no string was provided.
NOVALUE	The derived name of the variable whose attempted reference caused the NOVALUE condition. The NOVALUE condition trap can be enabled only using SIGNAL ON.
SYNTAX	Any string the language processor associated with the error. This can be the null string if you did not provide a specific string. Note that the special variables RC and SIGL provide information on the nature and position of the processing error. You can enable the SYNTAX condition trap only by using SIGNAL ON.

Special Variables

A special variable is one that may be set automatically during processing of a REXX program. There are three special variables: RC, RESULT, and SIGL. None of these has an initial value, but the program may alter them. (For information about RESULT, see page 65.)

The Special Variable RC

For ERROR and FAILURE, the REXX special variable RC is set to the command return code, as usual, before control is transferred to the condition label.

For SIGNAL ON SYNTAX, RC is set to the syntax error number.

The Special Variable SIGL

Following any transfer of control because of a CALL or SIGNAL, the program line number of the clause causing the transfer of control is stored in the special variable SIGL. Where the transfer of control is because of a condition trap, the line number assigned to SIGL is that of the last clause processed (at the current subroutine level) before the CALL or SIGNAL took place. This is especially useful for SIGNAL ON SYNTAX when the number of the line in error can be used, for example, to control a text editor. Typically, code following the SYNTAX label may PARSE SOURCE to find the source of the data, then call an editor to edit the source file positioned at the line in error. Note that in this case you may have to run the program again before any changes made in the editor can take effect.

Conditions and Condition Traps

Alternatively, SIGL can be used to help determine the cause of an error (such as the occasional failure of a function call) as in the following example:

```
signal on syntax
a = a + 1      /* This is to create a syntax error */
say 'SYNTAX error not raised'
exit

/* Standard handler for SIGNAL ON SYNTAX */
syntax:
  say 'REXX error' rc 'in line' sigl ':' "ERRORTXT"(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 uses *standard* streams for its display input and output. The standard streams are automatically available to a REXX program. The three standard streams REXX uses are:

- Standard input** The standard input stream is an input only stream with the predefined label STDIN.
- Standard output** The standard output stream is an output only stream with the predefined label STDOUT.
- Standard error** The standard error stream is an output only stream with the predefined label STDERR.

REXX determines what the stream characteristics are and what operations are valid for those streams.

The external data queue is also available to REXX programs for some line operations.

Default Standard Streams

The standard input stream (STDIN), standard output stream (STDOUT), and standard error stream (STDERR) are set, by default, as follows:

- If a REXX program is being run interactively, the standard streams all default to the Integrated Language Environment (ILE) Session Manager. STDIN uses the ILE Session Manager for input and STDOUT and STDERR use it for output. These streams provide a line-at-a-time input and output to the display station.
- If a REXX program is being run in batch mode, the standard input stream defaults to the file QINLINE while the standard output and standard error streams default to the file QPRINT.

The standard streams open automatically when each is needed by the REXX program. The SAY instruction uses STDOUT, the PULL instruction uses STDIN when the external data queue is empty, and the TRACE instruction uses STDIN for input and STDERR for output.

The standard stream defaults can be overridden with CL commands. Because of the way standard streams are opened, these CL commands must be issued before the first use of each stream by the REXX program. To override the standard input stream, the file name on the CL command must be STDIN. Similarly, for the standard output and standard error streams, the names must be STDOUT and STDERR, respectively. The stream defaults can be changed in the following ways:

- STDOUT and STDERR can be redirected to a printer file other than the default printer file.
- STDIN can be redirected to a database physical file or to a named inline spooled file.

Since the standard streams are used by the SAY, PULL and TRACE instructions you must be aware of the fact that overriding the standard streams will affect how these instructions will operate.

Note: For more details on the behavior of standard streams, see the *ILE C/400 Programmer's Guide*, SC09-2069.

Standard Stream Open

Each standard stream is opened when the first attempt is made to read from or write to it by the REXX program. Therefore, a REXX procedure may issue override commands for the standard streams before they are first used within this invocation of REXX, and they will take effect. If the overrides are issued after the first use, the override will not affect the current invocation of REXX.

The External Data Queue

The external data queue is a queue of character strings that can only be accessed by line operations. It is external to REXX procedures because other programs may have access to the queue when REXX relinquishes control to some other program. Non-REXX programs access the external data queue through the QREXQ interface. For more information about the QREXQ interface, see "Queue Application Programming Interface" on page 170. It is intended to be used as an inter-program communication device and has the following characteristics:

- Data in the queue is arbitrary and no characters have any special meaning or effect.
- Lines can be removed from the queue using the REXX instructions PULL and PARSE PULL.
- When the queue is empty, a PULL or PARSE PULL instruction will read lines from the default character input stream, the ILE Session Manager, see "Terminal Input and Output" on page 149.
- If the queue is empty at the time of a request through the QREXQ interface the "queue empty" return code will be issued.
- Lines can be added to the head of the queue with the PUSH instruction, or to the tail with the QUEUE instruction.
- The queue may be logically subdivided into independent buffers.
- The QUEUED function returns the number of lines in the entire queue.
- The external data queue is globally available to all programs running within the same job. One job cannot access another job's external data queue.

Size Limits

The maximum size of all the data contained in the queue is 15.5MB. The size of a single data item is limited to 32,767 bytes. Items which are longer than 32,767 will be truncated with no error indication.

If, by using the PUSH or QUEUE instructions, the maximum size of the queue would be exceeded, the queue is left unchanged and the following indication(s) are returned to the program trying to perform the function:

- If the operation is a REXX program executing a PUSH or QUEUE instruction, then the diagnostic message CPD7CF8, "REXX external data queue is full," is sent to the language processor's program message queue. REXX Error 48, "Failure in system service," (CPD7CB0) will be set, which in turn will raise the SYNTAX condition.

Note: If there is a *SIGNAL ON SYNTAX* instruction in effect, the REXX program can issue a RCVMSG command to receive the diagnostic and determine the cause of Error 48.

- If the operation was a call to the QREXQ interface, then the escape message CPF7CF8 “External data queue is full,” is issued to the caller’s program message queue and the return code parameter will be set to indicate “No space available.” For more information about the QREXQ interface, see “Queue Application Programming Interface” on page 170.

Damage Handling

If the REXX language processor determines, while executing a queue operation, that the external data queue has been damaged, the following action is taken:

- If the operation was a REXX PUSH, PULL, or QUEUE instruction, the diagnostic message “REXX external data queue is damaged,” (CPD7CF7) is sent to the language processor’s program message queue. Then REXX Error 48, “Failure in system service,” (CPD7CB0) is set, which in turn causes the *SYNTAX* condition to be raised.

Note: If there is a *SIGNAL ON SYNTAX* instruction in effect, the REXX program can issue a RCVMSG command to receive the diagnostic and determine the cause of the Error 48.

- If the operation was a call to QREXQ, CPF7CF7 is issued and the return code parameter is set to indicate “queue damaged.”

In either case, the queue will be explicitly marked “damaged” to prevent further queue operations. To clear this condition, the Remove REXX Buffer (RMVREXBUF) command with the *ALL parameter can be used. This will delete the external data queue and create a new one. All data on the queue will be lost.

Note: Once the queue has been damaged, no new REXX programs can be started, because the REXX language processor’s initialization code needs to refer to the queue. In this situation, escape message CPD7CF7, “REXX external data queue is damaged,” will be issued and the language processor will end.

CL Queue Commands

The Add REXX Buffer (ADDREXBUF) and Remove REXX Buffer (RMVREXBUF) commands allow REXX procedures to establish and delete new buffers in the external data queue. See the *CL Reference* for the full syntax and use of these commands.

Example of the use of the REXX queue

```

/*                                                    */
/*          push/pull                                */
/*                                                    */
push date() time()          /* push date and time    */
do 1000                     /* lets pass some time  */
  NOP                       /* doing nothing        */
end                          /* end of loop          */
pull a                      /* pull them            */
say 'Pushed at ' a ', Pulled at ' date() time() /* say now and then */

```

Chapter 9. AS/400 System Interfaces

This chapter describes the use of the REXX language processor with programs written in other languages and with other user-defined applications, as well as some REXX/400 specific topics. The topics discussed in this chapter are:

- Entering REXX Source Code
- Starting the REXX Language Processor
- Terminal Input and Output
- Pseudo-CL Variables
- Security
- Return Codes and Values
- Starting the Language Processor from an Application
- Command Interface
- Data Types and Structures
- External Functions and Subroutines
- Variable Pool Interface
- System Exit Interfaces
- Queuing Interfaces.

Note: All examples of calling conventions are shown in the ILE/C programming language. Refer to the Introduction for a list of other applicable languages.

REXX on the AS/400 System

Using REXX on the AS/400 system is discussed in detail in the *REXX/400 Programmer's Guide*, SC41-5728. The following sections describe some of the tools available for developing REXX/400 programs.

Entering REXX Source Code

Because REXX is an interpreted language, a REXX program is not compiled. Rather, the source code of the program is run by starting the REXX language processor and specifying the name of a REXX program (its source file and member) to run. A source file, named QREXSRC, is available in the library QGPL.

The source code for REXX programs can be created using any system text facility, including the Source Entry Utility (SEU). The SEU does not provide prompting or syntax checking for REXX source entry. See the *ADTS/400: Source Entry Utility* for more information.

Although the REXX language processor will accept a member of any source type, using the source type **REXX** for source members that contain REXX programs provides certain advantages in performance. The REXX language processor converts the REXX source type into an internal form and saves it to be used for subsequent calls to that program. For source types other than REXX, the internal form is not saved when the REXX program ends. This means that every time the program is run, this internal form will be re-created.

Starting the REXX Language Processor

REXX programs are run by invoking the REXX language processor and specifying the name of the REXX program to be run.

REXX can be started in one of two ways: 1) by using the Start REXX Procedure (STRREXPRC) command or 2) by using a CL command that has a REXX program as its command processing program (CPP).

In addition, REXX provides an application programming interface (API) to allow programs written in other languages to make direct use of the language processor. These interfaces are discussed in “Starting the Language Processor from an Application” on page 151.

Starting REXX with the Start REXX Procedure (STRREXPRC) Command

The Start REXX Procedure (STRREXPRC) command starts the REXX language processor for a specified REXX program. The STRREXPRC command can be used wherever a CL command is used. This is a way to run a REXX procedure from a CL program. Another way is CALL QREXX in the QSYS library, see page 151 for more information.

For more information on the STRREXPRC command, see the *CL Reference*, SC41-5722.

Starting REXX from a Command Definition Object (CDO)

The REXX language processor is started when a CL command that has a REXX program specified as its command processing program (CPP) is run. A command which has a REXX program as the CPP can be used like any other CL command. In addition, a CDO can be called from within a program written in another programming language by using the CALL QCMDEXEC facility. See the *CL Reference*, SC41-5722, for more information.

The REXX language processor options available to a CL command using a REXX program as its CPP differ from those available on STRREXPRC:

- You must specify *REXX as the CPP when the CL command is created. That is, the CPP cannot be changed to or from *REXX by using the Change Command (CHGCMD) command. Once you create a command, the only way to change the CPP to or from *REXX is to delete the command and re-create it.
- You can specify system exit programs and default command environments to be used with the REXX program CPP when the CL command is created. These, too, can be changed thereafter only by using CHGCMD.
- You can use more than one CL command parameter to provide input data to be passed to the REXX program CPP. This differs from the STRREXPRC command, which provides only one parameter field for data to be passed (as a single string) to the REXX program.

A CL command using a REXX program as its CPP can define parameters as needed. Most of the facilities of command definition are available for defining the parameters. For more information, see the *CL Reference*, SC41-5722.

- When a REXX program acts as a CPP for a CL command, the input argument string it receives differs from that passed by the STRREXPRC command. The

system builds this argument string from the input string, to which data from the command definition object itself may be added. Only one argument is passed to REXX by a command. The command string will be expanded to include keywords and defaults which may not have been explicitly entered.

Terminal Input and Output

When a REXX program is running in interactive mode, all terminal input and output is displayed by the ILE Session Manager. The ILE Session Manager is a system facility that provides line mode access to the display for output and the keyboard for input. This line mode operation is fixed and cannot be changed by the use of display files.

The ILE Session Manager display provides the following features:

- Paging backward and forward with the PgUp/PgDn keys
- Retrieval of previous entries with function key F9
- Exit with function key F3
- End-of-file signalling with function key F4
- Print the scroller with function key F6
- Scroll to the top with function key F17
- Scroll to the bottom with function key F18
- Scroll to the left with function key F19
- Scroll to the right with function key F20
- Issue commands from the user window with function key F21
- Output of characters below '40'X.

Pseudo-CL Variables in REXX programs

In CL programming, parameters to a command can be specified as explicit values or as variables. Values may be returned from a command in CL variables though the use of RTNVAL(*YES) parameters. RTNVAL(*YES) parameters can be specified when a command is created or changed. This is detailed in the *CL Reference*, SC41-5722, REXX programs that call CL commands can use REXX variables for these parameters. Variable names must conform to both the standard rules for naming CL variables and to the rules for naming REXX symbols (see “Assignments and Symbols” on page 21).

- The variable name must be immediately preceded by an ampersand (&) and have a maximum of 10 characters, not including the ampersand.
- The variable name must begin with an alphabetic character and may contain any alphabetic or numeric character (A-Z, a-z, 0-9), a period (.), or an underscore (_).
- DBCS characters are not allowed.

Notes:

1. The period (.), which has no specific significance in CL variable names, is used in REXX to build compound symbols and arrays.
2. The CL command environment detects all DBCS strings on CL command lines, independent of REXX's own OPTIONS ETMODE and OPTIONS EXMODE settings. For more information about DBCS support in REXX, see Appendix A, “Double-Byte Character Set (DBCS) Support” on page 177.

Compound Pseudo-CL Variables

Compound symbols (see “Compound Symbols” on page 22) can also be used to pass parameters to CL commands.

During processing, REXX builds a derived name from the compound name by substituting the values of the tail components for the components. REXX then accesses the variable pool using the derived name. This allows REXX programmers to simulate arrays, as well as more complex forms of associative storage.

Consider the set of all REXX compound variables beginning with a given stem. A REXX programmer can initialize every variable in this set to a given value by initializing the stem to that value.

The following is an example of using stem variables.

```
Say A.B           /* Program displays 'A.B'    */
A. = 'AS/400'
Say A.           /* Program displays 'AS/400'  */
Say A.B         /* Program displays 'AS/400'  */
A.2 = 'second'
B = 2
Say A.B         /* Program displays 'second'  */
A.THIRD = 3
C = 'THIRD'
Say A.C         /* Program displays '3'      */
```

The order in which return values are assigned to REXX variables depends on the order in which they occur in the command string. This has special significance when using compound variables for the RTNVAL(*YES) parameters as demonstrated in the compound variable example.

The following is an example of using pseudo-CL variables.

```
'RTVJOBA JOB(&JOBNAME) NBR(&NUM) CURLIB(&LIB.NUM) '
Say 'The name of this job is' jobname'.'
Say ' Its number is' num','
Say ' and its current library is' lib.num'.'

/* However...*/
num = 'ABC'
'RTVJOBA JOB(&JOBNAME) CURLIB(&LIB.NUM) NBR(&NUM) '
Say 'The name of this job is' jobname'.'
Say ' Its number is' num','
Say ' but its current library is not' lib.num'.'
/* Note that because the value of NUM has changed, the variable
LIB.NUM no longer is the one containing the current library.
LIB.ABC contains the current library, because NUM='ABC' is before
the command */
```

Security

Security of REXX procedures is managed at a source file level. REXX runs from source members which are authorized only at the file level. Authorities and owners cannot be set for individual members. For more information on AS/400 system security, see the *Security – Reference* book.

Application Interfaces

In this section, 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 through the use of handlers for command, external function and system exit processing.

Commands	Commands are single clauses consisting of just an expression. The expression is evaluated and the result is passed as a command string to the currently addressed environment.
External Functions	External functions are direct extensions to the capabilities of REXX. An application or REXX program can provide a function to augment the set of built-in REXX functions. Unlike commands which generally correspond to the application's usual command set, functions are generally very REXX specific and may only have a meaning from within REXX.
System Exits	System exits allow REXX to operate under programmer-defined variations of the operating environment. By using these exits, specific REXX actions can be run as calls to routines provided by the caller of the language processor, instead of using code of the language processor itself.

In addition, REXX also provides the means for applications to manipulate the variables in REXX programs (the variable pool interface) and to use and manipulate the REXX external data queue (queuing services).

Return Codes and Values

Most system interfaces require that an integer code be returned from the interface to indicate success or failure of the call. This is commonly called a return code. Many interfaces also have a specific return value, not indicating success or failure, but rather a value requested for use by the function. The return value in many cases is set by the variable pool interface from commands to the RC variable differs from return codes passed across interfaces to indicate success or failure. For example, when using a system exit to handle commands, the following happens:

1. The command is processed and a return value is set through the variable pool interface. The return value is placed in the special variable RC as the command's return code.
2. The return code parameter of the system exit program is set to an integer which indicates success or failure of the system exit. This has no effect on the value in the RC variable.

Starting the Language Processor from an Application

An application can start the REXX language processor by calling the program QREXX in the QSYS library. The parameter structure that can be passed is:

Source Member Name (required)

The name of the source member that contains the REXX program to be run. This parameter must be a 10-byte character field.

Source File Name (required)

The qualified name of the source file that contains the member that is specified in parameter 1 above. This parameter must be a 20-byte character field. The file name occupies the first ten positions of this field and is left justified. The library name occupies the second 10 positions and is also left justified. The library name can be *LIBL, *CURLIB, or an actual library name.

Argument String (required)

A structure consisting of a 2-byte binary field followed by a character field. The 2-byte binary field contains a positive integer value that specifies the length of the character field. The character field must be at least as long as specified by the 2-byte binary field but may be longer. REXX will only process the specified length. All data within the character field is treated as character data. The character field is the argument string that will be made available to the REXX program. The maximum length of the character field is 32,767. If there is no argument string to be passed, then the length field must be zero.

Command Environment (required)

The name of the initial command environment when the REXX program is started. This is a 20-byte character field. If the host CL command environment is to be used, either the value COMMAND or *COMMAND can be placed in this field, left justified. If this field is entirely blank, the value *COMMAND is assumed. If a program is to be used, the first 10 positions contain the name of the program, left justified, while the next ten bytes contain the library qualifier, left justified. The library qualifier can be *LIBL, *CURLIB, or an actual library name.

The CPICOMM, *CPICOMM, EXECSQL, and *EXECSQL values can also be used as the initial command environment when the REXX program is started.

The CPICOMM environment, which is the communications element of the SAA Common Programming Interface (CPI), lets you issue CPI-Communications commands. For more information on CPI-Communications, see the *CPI Communications Reference*.

The Structured Query Language (SQL) environment, called EXECSQL, lets you use SQL, which is the standard database interface language used by DB2/400. For more information on SQL statements, see the *DB2 for AS/400 SQL Reference*, or for information on EXECSQL, see the *DB2 for AS/400 SQL Programming*.

Note: To use the EXECSQL environment on a system which does not have the DB2/400 Query Management and SQL Development Kit Version 3 LPP, 5763-ST1, installed, see the *DB2 for AS/400 SQL Programming* for special instructions for handling the REXX program.

System Exits (required)

The array of structures that identify the system exits to be used by this invocation of the REXX language processor. For each, a structure comprising two subfields is used:

- A 20-byte character field that specifies the name of the program to be used for this exit code. The first 10 positions specify the name of the program, left justified. The second 10 positions specify the name of the library, also left justified. The library name can be given as *LIBL, *CURLIB, or an actual library name.
- A 2-byte binary field that specifies one of the valid system exit codes.

One such structure must be provided for each system exit being used. The total number of structures must equal the number specified in the *Number of System Exits* parameter.

Number of System Exits (required)

A 2-byte binary field that contains an integer value. This value specifies the number of system exits that are being provided by the *System Exit* parameter. If the value is 0, no system exits will be in use.

Note: Above parameters are positional and must be supplied. However, a null parameter will work for parameters 3-6.

Command Interface

The CL Command Environment (COMMAND)

The CL command environment is the default command environment. It processes CL commands from within a REXX program.

Special handling is required when a command has RTNVAL(*YES) parameters. See "Pseudo-CL Variables in REXX programs" on page 149 for details.

Command Logging: The AS/400 system uses the *LOGCLPGM job attribute whenever a REXX program issues a command in the CL command environment COMMAND. In other words, if *LOGCLPGM *YES is active, then CL commands used in REXX programs are logged by the system when they are run. Note that if REXX tracing is active, commands may appear twice in the job log, once as trace output and once as logged by the system.

Return Codes from COMMAND: A command that ends without issuing an escape message will cause the RC variable to be set to zero (0). Otherwise, the RC variable is set to the message ID of the escape message received.

REXX monitors for escape messages issued by a CL command called by a REXX program. Any command resulting in an escape message of CPF0001 or CPF9999 will raise a FAILURE condition. Once the FAILURE condition is raised, the special variable RC will be set to that particular escape message ID.

Any command resulting in an escape message other than CPF0001 or CPF9999 will raise an ERROR condition. Once the ERROR condition is raised, the special variable RC will be set to that particular escape message ID.

Notes:

1. If a command is longer than the maximum length of 6,000 bytes, a return code of -1 is immediately returned to the REXX program in the RC variable, and the command is not issued. The failure condition is raised.
2. Status and notify messages from the user-defined and CL command environments can be trapped using the Set Message Return Code (SETMSGRC) function. For more information on the SETMSGRC function, see "SETMSGRC" on page 108.

The CPI-Communications (CPICOMM) and Structured Query Language (EXECSQL) Command Environments

The CPICOMM environment, which is the communications element of the SAA Common Programming Interface (CPI), lets you issue CPI-Communications calls. For more information on CPI-Communications, see the *CPI Communications Reference*.

The Structured Query Language (SQL) environment, called EXECSQL, lets you use SQL, which is the standard database interface language used by DB2/400. For more information on SQL statements, see the *DB2 for AS/400 SQL Reference*, or for information on EXECSQL, see the *DB2 for AS/400 SQL Programming*.

Note: To use the EXECSQL environment on a system which does not have the DB2/400 Query Management and SQL Development Kit Version 3 LPP, 5763-ST1, installed, see the *DB2 for AS/400 SQL Programming* for special instructions for handling the REXX program.

Return Codes from CPICOMM: The REXX variable RC indicates either a successful call to the CPI-Communications Interface or a failure to call the CPI-Communications Interface. If the call completes successfully, the REXX RC variable is set to zero. If the call cannot be completed, the REXX RC variable is set to a negative number and the failure condition is raised. For more information on values returned in the REXX RC variable for the CPICOMM environment, refer to the *REXX/400 Programmer's Guide*, SC41-5728.

Return Codes from EXECSQL: The REXX variable RC indicates the overall status of a call to the SQL environment. If the call completes successfully, the REXX RC variable is set to zero. A positive number indicates an SQL warning, and a negative number indicates an SQL error condition. For more information on values returned in the REXX RC variable for the EXECSQL environment, refer to the *REXX/400 Programmer's Guide*, SC41-5728, the *DB2 for AS/400 SQL Reference*, SC41-5612, or the *DB2 for AS/400 SQL Programming*, SC41-5611.

User-Defined Command Environments

When a user-defined environment is active or addressed, a REXX command is passed to the specified program as a parameter.

When it returns, the environment must pass back to REXX a return code that indicates if the command was successfully processed. (See "ADDRESS" on page 30 for the format of user-defined environment names.)

The program specified as the command environment must conform to REXX linkage conventions. These conventions make use of a two-byte binary integer data type, so not all programming languages can be used to code the program. The value returned is placed into a special REXX variable, named RC, which can then be used by the REXX program.

A standard system call to the program specified by the environment name is made, and the return value passed back by the program sets RC. The program must accept the following parameters:

1. The command in SHORT_VARSTRING format. The SHORT_VARSTRING structure passes varying length strings between the REXX language processor and command environments. It is defined as follows:
 - A two-byte integer containing the length of the string

- The actual string.
2. The return value string for the RC variable in a SHORT_VARSTRING structure. This buffer is pre-allocated by REXX to a length of 500 bytes and set to 0 (for example, length=1, string='0'). The program must put the return value for the RC variable into this buffer and set the length to the length of the data in the buffer.
 3. A two-byte integer to be set by the program to the proper Error or Failure code. This parameter is initialized to 0. The following are the valid return codes for this parameter, and the action the language processor takes as a result:

0	Command executed successfully. No action taken by the language processor.
1	A command ERROR occurred. The language processor raises the ERROR condition.
2	A command FAILURE occurred. The language processor raises the FAILURE condition.

Return Codes from User-Defined Command Environments: Return codes from user-defined command environments are obtained from the return value string parameter. The only restriction is that the maximum length of a return code is 500 bytes. Escape messages may be used in place of the return value string parameter. If an escape message is issued to the language processor by a user-defined command environment, it will be treated exactly as an escape in the CL command environment, except that for all escape messages the failure condition is raised. For example, RC is set to the message ID of the escape message, the message is left marked new, or other appropriate actions are taken. The value, if any, in the return value string parameter is then ignored.

Note: If a command is longer than the maximum length of 6,000 bytes, a return code of -1 is immediately returned to the REXX program in the RC variable, and the command is not issued. The failure condition is raised.

Data Types and Structures

Many of the interfaces described in this chapter require a method by which arguments may be passed. The data structure named RXSTRING is used for communication between the REXX language processor and application programs. A RXSTRING contains a pointer to a character string followed by a four-byte binary value containing the length of the string. For example, in ILE/C an RXSTRING could be declared as follows:

```
typedef struct {
    char          *rxstrptr; /* Pointer to the string */
    unsigned long rxstrlen; /* Length of the string */
} RXSTRING
```

REXX arguments may either be given a value (which may be null, '') or not specified at all.

- If an argument is specified, then the appropriate parts of the RXSTRING will be given values.
- If a null string ('') is specified as the argument, then *rxstrlen* will be zero, and *rxstrptr* will be nonzero.
- If no argument is specified, then *rxstrptr* will be zero.

All of the interface examples in this section are written in ILE/C.

External Functions and Subroutines

External functions and subroutines may be written in REXX or in other languages that support the system-dependent interface. Details on invoking REXX external functions and subroutines are provided in Chapter 4, "Functions" on page 75. This section describes the system-dependent interface to external functions and subroutines written in languages other than REXX.

Search Order

External functions and subroutines are located by using the methods used to resolve a REXX CALL reference. They can reside in any library, and will be found as long as the library is in the job's current library list. Note that explicit library qualification is not supported.

Arguments

All external functions, including those described here, require a method by which string arguments may be passed. A maximum of 20 arguments in an external function or subroutine invocation is allowed.

The following is an example:

```
typedef struct                /* The RXSTRING structure.    */
{ char      *rxstrptr ;      /* Pointer to string data.    */
  unsigned long rxstrlen ;   /* Length of string.         */
} RXSTRING;                  /*                               */

main(int argc, char *argv[])
{

    /* Declaration of Local variables to receive the passed para- */
    /* meters follows:                                           */
    RXSTRING  *args  ;      /* Parameter array (see note 1). */
    short int  numargs ;    /* Number of elements in the     */
                                /* parameter array.              */
    short int  func_sub ;   /* Function or external          */
                                /* subroutine call.              */
    short int  *errflag ;   /* Success/Failure flag.        */

    /* Some code to get the information passed into the declared */
    /* local variables following.                                  */

    args      = (RXSTRING *) argv[1] ; /* (See note 2.)                */
    numargs   = *(short int *) argv[2] ; /* Maximum of 20 allowed.      */
    func_sub  = *(short int *) argv[3] ; /* (See note 3.)                */
    errflag   = (short int *) argv[4] ; /* (See note 4.)                */

    /* All the remaining ILE/C program statements.              */
}
```

The arguments are described as:

ARGS is the pointer to an array of RXSTRINGs.

NUMARGS is a two-byte integer which indicates the number of parameters.

FUNC_SUB is a two-byte integer flag which indicates how the program was called. The possible values are:

1 indicates a subroutine invocation. For example,

```
Call EXTFUNC 1, 'A'
```

2 indicates a function invocation. For example,

```
x = EXTFUNC(1, 'A')
```

ERRFLAG is a two-byte integer return code which indicates success or failure. If zero (0), the routine was called correctly and no serious errors occurred. If nonzero, the language processor will assume that the routine was called incorrectly. It will issue Error 40, "Incorrect call to routine" and raise the SYNTAX condition. If no SIGNAL ON SYNTAX instruction is in effect, the REXX program will be terminated.

If the function wants to indicate an error situation to the language processor, it should return a nonzero code in the *ERRFLAG* parameter. Escape messages will be monitored, and kept as diagnostic messages, but will be treated like nonzero return codes. The language processor will issue Error 40, "Incorrect call to routine" and raise the SYNTAX condition.

Return Value

All external functions must have a returned value.

External subroutines can have a returned value. This value is used to replace the function invocation which is automatically placed in the special variable RESULT. To set this value, the external functions and subroutines use the variable pool interface, with function code SHVEXTFN.

System Exit Interfaces

User-written system exit programs can be used to define how an application which is using the language processor controls certain services. Each exit interface provides access to a separate type of service required by the language processor. The exit may either replace or supplement the default action taken by the REXX language processor. See "Exit Conditions" on page 160 for more information. System exits can be written to control how REXX programs perform the following tasks:

- Process external functions
- Process commands
- Manipulate the external data queue
- Perform input and output
- Halt processing
- Trace
- Initialize processing
- Terminate processing.

When the REXX language processor is called, the exit routines may be specified with the EXITPGM parameter of the Start REXX Procedure (STRREXPRC) command, the REXEXITPGM of the Create Command (CRTCMD) and Change Command (CHGCMD) commands, or the *exitpgm* parameter on a call to the QREXX interface. See "Starting the Language Processor from an Application" on

page 151 for details on how the exit programs are specified by the QREXX interface. The specified system exits will be used whenever the language processor requires that service.

The descriptions of each exit interface, explanations for when they are called, and the required conventions are provided in the following sections.

Notes:

1. Because pointers are required for passing parameters to system exit programs, system exits may only be written in languages which support a pointer data type.
2. When exits are specified, all REXX programs called as external functions or subroutines will also run using the exits specified for the top level program. REXX programs called with a command will not use these exits unless they are specified on the command.

System Exits and the Variable Pool

Many of the exits require that an arbitrary length character string is passed back to the language processor as a result. The *SHVEXIT* function of the QREXVAR program is available for this purpose. QREXVAR may only be called once per invocation of the exit to return a value through *SHVEXIT*. If it is called more than once, QREXVAR will return the code indicating that the requested service is not available.

In general, the *SHVEXIT* function is the only variable pool function available to system exit programs. However, the RXFNC, RXCMD, RXINI, and RXTER exits have all the variable pool functions available to them.

System Exit Functions and Subfunctions

The following table lists the system exit functions and subfunctions defined for the REXX language processor. These are passed to the exit programs in the *FUNC* and *SUBFUNC* parameters.

Table 2. System Exit Functions and Subfunctions

Function	Subfunction	Description
RXFNC 2		Process external functions
	RXFNCCAL 1	Currently the only subcode value
RXCMD 3		Process host commands
	RXCMDHST 1	Currently the only subcode value
RXMSQ 4		Manipulate queue
	RXMSQPLL 1	Pull a line from queue
	RXMSQPSH 2	Place a line on queue
	RXMSQSIZ 3	Return number of lines on queue
RXSIO 5		Session input and output
	RXSIOSAY 1	SAY a line to STDOUT
	RXSIOTRC 2	Trace output
	RXSIOTRD 3	Read from session char stream
	RXSIODTR 4	DEBUG read from session char stream
RXHLT 7		Halt processing
	RXHLTCLR 1	Clear HALT indicator
	RXHLTTST 2	Test HALT indicator
RXTRC 8		Test external trace indicator
	RXTRCTST 1	Currently the only subcode value
RXINI 9		Initialization processing
	RXINIEXT 1	Currently the only subcode value
RXTER 10		Termination processing
	RXTEREXT 1	Currently the only subcode value

Entry Conditions

A system exit is identified when REXX is called in order to control a specific REXX service. A single program can serve as a handler for more than one exit, if desired. The exit program parameter list must be declared as follows:

FUNC is the integer code, two-bytes in length.

SUBFUNC is the subfunction code for the exit, two-bytes in length.

RC is the four-byte integer return code to be set by the exit.

PARM is a pointer to the exit specific parameter structure that is identified by the exit function and subfunction codes in the *FUNC* and *SUBFUNC* parameters. See the exit descriptions for the format of the parameter structure for each exit.

The following example shows how an exit program would access the parameters passed to it.

```
main (argc, argv)
int argc;          /* Number of parameters, should be five (5)*/
char *argv[];     /* Parameter array */

/* Local variables */
int *func;        /* the function */
int *subfunc;    /* the sub-function */
int *rc;         /* exit return code */
RXxxx_PARM *parm; /* must be cast to RXxxx_PARM depending
                  /* on func and subfunc */

/* Some executable code to get the parameters into local variables */
func = (int *) argv[1]; /* Function is first parm */
subfunc = (int *) argv[2]; /* Sub-function is second parm */
rc = (int *) argv[3]; /* Return Code is the third parm,
                      /* referenced in the program by *rc */
parm = (RXxxx_PARM *) argv[4];
                      /* Exit-specific parameter block is fourth,*/
                      /* referenced in the program by *parm */
```

Exit Conditions

On return from the exit, the return code indicates the results of the exit call and the parameter structure is updated by the exit interface. The return code can signal one of three actions:

- 0** Exit performed successfully. The parameter structure has been updated as appropriate for that exit.
- 1** Exit chose not to handle the request. The language processor will handle the request using the default means.
- 1** Exit detected an error during processing. REXX Error 48, "Failure in system service," (CPD7CB0) will be raised. The exit will not be disabled.

If an exit program sends an escape message to the language processor, then Error 48, "Failure in system service," (CPD7CB0) is raised, and the exit is disabled for the duration of this invocation of the language processor.

Exit Definitions

RXFNC This exit is called before an external function or subroutine is called by the REXX program. The exit is responsible for locating and invoking the requested routine.

Currently this service supports only one subfunction. The subfunction request code is specified by the second parameter. The RXFNC subfunction and parameter structure are:

RXFNCAL Process external functions.

```

struct rxfnc_parm {
  struct {
    unsigned char rxfferr;      /* Character field with 1 byte length */
                                /* 1 = Invalid call to routine      */
                                /* 0 = Valid call to routine         */
    unsigned char rxffnfnfnd;  /* Character field with 1 byte length */
                                /* 1 = Routine not found             */
                                /* 0 = Routine found                */
    unsigned char rxffsub;     /* Character field with 1 byte length */
                                /* 1 = Subroutine call              */
                                /* 0 = Function call                */
  } rxfnc_flags;
  short int rxfnc_argc;        /* Integer field with 2 byte length */
                                /* Number of arguments in list      */
  RXSTRING rxfnc_name;        /* Routine name                      */
  PRXSTRING rxfnc_arguments; /* Pointer to argument list          */
}

```

The function to be called is defined by the *rxfnc_name* RXSTRING. The arguments to the function are contained in the array pointed to by *rxfnc_arguments*. The individual elements of the array are RXSTRINGs that define each argument.

The flag *rxffsub* is on if the routine is called using the CALL instruction rather than as a function.

On return from the exit, values in *rxfnc_flags* indicate the processing status of the function.

- If neither *rxfferr* nor *rxffnfnfnd* is on, the routine was called successfully and ran.
- If *rxffnfnfnd* is on, the exit handled the request, but could not locate the requested function or subroutine. The language processor will raise REXX Error 43, "Routine not found," (CPD7CAB).
- If *rxfferr* is on, the exit handled the request, but an error occurred during execution of the function or subroutine. In this situation, the language processor will raise REXX Error 40, "Incorrect call to routine," (CPD7CA8).

Function return values are passed back to the language processor using the *SHVEXIT* function of the QREXVAR routine. This routine may only be called once per invocation of the exit to return a value. If the routine was called as a function and a result was not returned, the language processor will give REXX Error 44, "Function did not return data," (CPD7CAC). If the routine was called as a subroutine, the returned result is optional.

RXCMD This exit is called whenever the language processor is going to call a command. The exit is responsible for locating and invoking the command, given string and the current environment name.

RXCMDHST Invoke a host command.

```
struct rxcmd_parm {
  struct {
    unsigned char rxfcfail; /* Character field with 1 byte length */
                          /* 1 = Command FAILURE occurred */
                          /* (can be trapped using SIGNAL or */
                          /* CALL ON FAILURE) */
                          /* 0 = Command FAILURE did not occur */
    unsigned char rxfcerr; /* Character field with 1 byte length */
                          /* 1 = Command ERROR occurred */
                          /* (can be trapped using SIGNAL */
                          /* or CALL ON ERROR) */
                          /* 0 = Command ERROR did not occur */
  } rxcmd_flags;
  RXSTRING rxcmd_address; /* Current address name */
  RXSTRING rxcmd_command; /* Command to be processed */
}
```

The *rxcmd_address* RXSTRING describes the current address setting under which the command is to be issued and *rxcmd_command* is the actual command being issued.

The flags *rxfcfail* and *rxfcerr* are used by the exit to indicate that an ERROR or FAILURE condition has occurred. The definition of what constitutes an ERROR or FAILURE of a command is under the control of the exit. Commands flagged as an ERROR will trigger a SIGNAL ON ERROR or CALL ON ERROR trap, if one is in effect. Commands flagged with a FAILURE condition will have a similar effect, with SIGNAL ON FAILURE and CALL ON FAILURE. If both FAILURE and ERROR are both being trapped, ERRORS are ignored.

Notes:

1. To cause CL commands to be processed from this exit, the system program QCMDEXC must be used.
2. The QREXVAR interface is fully enabled during calls to the RXCMD exits.

RXMSQ Manipulate queue.

This service supports a number of subfunctions for external data queue services as specified by the second parameter. The parameter structure depends on the particular subfunction called. The RXMSQ subfunctions and their parameter structures are:

RXMSQPLL Pull a line from the queue. This exit is called to process requests to pull a line from the external data queue for PULL and PARSE PULL instructions. This exit only handles the data queue requests, the session input portion of the PULL and PARSE PULL commands would be handled by the RXSIO set of exits. RXMSQSIZ is called first, to determine if there are items in the queue. If the queue is empty, RXSIO is called.

There are no parameters for this exit. The data to be returned by the exit must be returned by using the SHVEXIT function of the QREXVAR routine. This routine may only be called once per invocation of the

exit to return a value. If the exit handles the call, but returns no data by using SHVEXIT, it is treated as though a null line was returned.

RXMSQPSH Place a line on the queue. This exit is called by the PUSH and QUEUE instructions to place a line on the external data queue.

```
struct rxmsq_parm {
  struct {
    unsigned char rxfmlifo; /* Char. field - 1 byte length */
                          /* 1 = Stack the line LIFO */
                          /* 0 = Stack the line FIFO */
  } rxmsq_flags;
  RXSTRING rxmsq_value; /* Data to be stacked */
}
```

The line to be placed on the queue is the result of evaluating the expression specified on a PUSH or QUEUE instruction. It is the responsibility of the exit to handle truncation of this string if the exit has a restriction on the maximum width of the queue. The queuing order is indicated by the flag *rxfmlifo*.

RXMSQSIZ Return the number of lines in the queue. This exit is called when the QUEUED() built-in function determines the current size of the data queue.

```
struct rxmsq_parm {
  unsigned long rxmsq_size; /* Number of lines in stack */
}
```

On return from the exit, *rxmsq_size* contains the size of the data queue as a four-byte integer.

Note: To access the external data queue from an exit program, the QREXQ interface must be used.

RXSIO Session input and output.

This service supports a number of subfunctions for performing session input and output. The second parameter specifies which RXSIO subfunction is being requested. The exit parameter structure depends on the particular subfunction called. The RXSIO subfunctions and their parameter structures are:

RXSIO SAY Session output processing. This exit is called to process the output from the SAY instruction.

```
struct rxsio_parm {
  RXSTRING rxsio_value; /* Actual data to display */
}
```

The line to be displayed on the standard output stream is the result of evaluating the expression specified on a SAY instruction. This string may be any length up to the maximum possible string size of approximately 16MB. It is the responsibility of the exit to handle truncation of this string if the string is too long.

RXSIO TRACE TRACE output processing. This exit is called to process output resulting from the TRACE instruction.

```
struct rxsio_parm {
  RXSTRING rxsio_value; /* Pointer to line to display */
}
```

The line to be displayed at the session is the result of a traced line. This string may be any length up to 16MB. It is the responsibility of the exit to handle truncation of this string if the string is too long.

RXSIOTRD Read from session. This exit is called to handle the session input and output portion of the PULL, PARSE PULL, and PARSE LINEIN instructions. This exit is only called for the actual input and output portion of these instructions. Requests to read from the external data queue are handled using the RXMSQ set of exits.

There are no parameters for this exit. The data to be returned by the exit must be returned by using the SHVEXIT function of the QREXVAR routine. This routine may only be called once per invocation of the exit to return a value. If the exit handles the call, but returns no data using SHVEXIT, it is treated as though a null line was returned.

RXSIODTR Trace read from session. This exit subfunction processes input requests from REXX interactive debug pauses.

There are no parameters for this exit. The data to be returned by the exit must be returned using the SHVEXIT function of the QREXVAR routine. SHVEXIT may only be set once per invocation of the exit to return a value. If the exit handles the call, but returns no data using SHVEXIT, it is treated as though a null line was returned.

RXHLT Halt processing.

This service supports a number of subfunctions for processing external *halt* requests. The second parameter specifies which subfunction is to be processed. The exit parameter structure depends on the particular subfunction called. The RXHLT subfunctions and their parameter structures are:

RXHLTCLR Clear halt indicator. This exit is called after the language processor has recognized a halt request and the halt request has been trapped by the REXX program using SIGNAL ON HALT or CALL ON HALT. This exit has no exit-specific parameter structure. It signals the exit handling HALT processing that the condition has been recognized and should be cleared.

RXHLTTST Test halt indicator. This exit is called after execution of each clause of the REXX program to check if a halt condition should be raised.

```
struct rxhlt_parm {
    struct {
        unsigned char rxfhhalt;    /* Char. field - 1 byte length */
                                   /* 1 = HALT cond. occurred */
                                   /* 0 = HALT cond. did not occur */
    } rxhlt_flags;
}
```

On return from this exit, *rxfhalt* will indicate whether a halt condition has occurred. The exit may also return a string that will be available using the `CONDITION(D)` built-in function. This string is returned by using the *SHVEXIT* function of the `QREXVAR` routine. This routine may only be called once per invocation of the exit to return a value.

Notes:

1. If the halt condition raised by a call to this exit is trapped by a `SIGNAL ON HALT` or `CALL ON HALT` instruction, then a call to the `RXHLTCLR` subfunction will occur to indicate that the condition has been trapped and should be cleared.
2. Both the `HALT` and `TRACE` exits are called by the REXX language processor each time a clause boundary within the REXX procedure is encountered. The `HALT` exit is called first, but if the exit indicates that a `HALT` should take place, the `TRACE` exit is still called. Thus, it is possible to both raise the `HALT` condition and alter the trace setting at one clause.

RXTRC This exit is called after execution of each clause of the REXX program to check if tracing should be turned on or off.

RXTRCTST Test external trace indicator.

```
struct rxtrc_parm {
  struct {
    unsigned char rxftrace; /* Character field with 1 byte length */
                          /* 1 = Trace on external trace setting */
                          /* 0 = Trace off external trace setting */
  } rxtrc_flags;
}
```

On return from this exit, *rxftrace* will indicate whether an external trace condition has occurred.

The language processor keeps a shadow copy of the *rxtrctst* return value. When this setting changes from `OFF` to `ON`, the language processor will turn on interactive debug mode as if the instruction `TRACE ?R` had been issued within the program. If the language processor detects a change in this setting from `ON` to `OFF`, then the language processor will turn off tracing as if the instruction `TRACE OFF` had been issued.

Note: Both the `HALT` and `TRACE` exits are called by the REXX language processor each time a clause boundary within the REXX procedure is encountered. The `HALT` exit is called first, but if the exit indicates that a `HALT` should take place, the `TRACE` exit is still called. Thus, it is possible to both raise the `HALT` condition and alter the trace setting at one clause.

RXINI This exit is called before interpretation of the first instruction of a program (including programs called as external functions and subroutines).

RXINIEXT Initialization processing.

This exit has no exit-specific parameter structure. It is called before the first instruction of the program is interpreted. The `QREXVAR` interface is enabled when this exit is called.

RXTER This exit is called after interpretation of the last instruction of a program (including programs called as external functions and subroutines).

RXTEREXT Termination processing.

This exit has no exit-specific parameter structure. It is called after the last instruction of the program is interpreted. The QREXVAR interface is enabled when this exit is called.

The variable pool interface is used by other programs to manipulate the set of variables available to the last active REXX program.

The QREXVAR Interface

QREXVAR, located in the QSYS library, is the program that performs the functions of the variable pool interface. Its calling syntax is:

```
#pragma linkage(QREXVAR,OS)
```

```
extern void QREXVAR (SHVBLOCK *, short int *);
```

```
    QREXVAR(shvblock_list_ptr, &return_code);
```

where:

shvblock_list_ptr

is a pointer to a list of shared variable request blocks.

return_code

is a two-byte integer indicating success or failure.

A linked list of shared variable request blocks is passed to the REXX function QREXVAR, with *shvblock_list_ptr* serving as the pointer to that list. Each request block can specify a different operation for the language processor to perform. Each request block is processed in turn, and control returns to the caller after the last block is processed or after a severe error has occurred. Therefore, the language processor can perform multiple operations with a single call to QREXVAR.

The two-byte *SHVRET* integer return code returned from the QREXVAR call contains the return code from the entire set of requests. The return code for each individual shared variable request block is found in a field of the block. The possible return codes from QREXVAR are:

0 or positive

Entire shared variable request block list was processed. The return code is the composite OR of the low-order *shvret* bytes. The bit positions, and corresponding meanings, are shown in "Shared-Variable Request Block."

-1

Entry conditions were not valid. The variable pool interface is not enabled at this time. The caller is not allowed to access the variable pool at this time or a specific variable pool function is not currently enabled.

Note: Many of the parameters for the QREXVAR interface are passed in the form of an RXSTRING. See "Data Types and Structures" on page 155 for additional detail.

Shared-Variable Request Block

The layout of a shared variable request block is as follows:

```
typedef struct shvnode {
    struct shvnode *shvnext;      /* chain pointer (0 if last block)*/

    RXSTRING      shvname;      /* Pointer to variable name      */
    RXSTRING      shvvalue;     /* Pointer to value buffer      */
    unsigned char shvcode;     /* Individual function code      */
    unsigned char shvret;      /* Individual return code flags  */
} SHVBLOCK;
```

The variable pool can have the following function codes.

Table 3. Variable Pool Function Codes

Function (shvcode)	Value	Description
SHVSET	0	Set variable from given value
SHVFETCH	1	Fetch value of variable
SHVDROPV	2	Drop variable
SHVSYSET	3	Symbolic name Set variable
SHVSYFET	4	Symbolic name Fetch variable
SHVSYDRO	5	Symbolic name Drop variable
SHVNEXTV	6	Fetch "next" variable
SHVPRIV	7	Fetch private information
SHVEXIT	8	Set system exit return value
SHVEXTFN	9	Set external function value

The variable pool can return the following values.

Table 4. Variable Pool Return Code Flags

Flag (shvret)	Value (Hex)	Description
SHVCLEAN	0x00	Execution was successful
SHVNEWV	0x01	Variable did not exist
SHVLVAR	0x02	Last variable transferred
SHVTRUNC	0x04	Truncation occurred (Fetch)
SHVBADN	0x08	Invalid variable name
SHVMEMFL	0x10	Out of memory failure
SHVBADF	0x80	Invalid function code (shvcode)

The SHVCODE functions use either a symbolic or direct interface to work with compound variables.

Symbolic The symbolic interface, used by *SHVSYSET*, *SHVSYFET* and *SHVSYDRO*, evaluates compound variables in the same way REXX does. See “Compound Symbols” on page 22 for more information.

Direct The direct interface does not evaluate compound variables. Therefore, any characters are allowed following a valid REXX stem. This interface is used by *SHVSET*, *SHVFETCH*, *SHVDROPV*, *SHVNEXTV*, *SHVPRIV*, *SHVEXIT*, and *SHVEXTFN*.

The specific actions for each function code are as follows:

SHVSET and SHVSYSET

Set variable. The *shvname* RXSTRING describes the name of the variable to be set, and *shvvalue* describes the value which is to be assigned to it. The name is validated to ensure that it does not contain invalid characters, and the variable is then set from the value given. If the name is just a stem, all variables with that stem are set, just as though this was a REXX assignment. *SHVNEWV* is set if the variable did not exist before the operation.

SHVFETCH and SHVSYFET

Fetch variable. The *shvname* RXSTRING describes the name of the variable to be fetched. *shvvalue* is a buffer into which the data is to be copied, and the RXSTRING length contains the length of the buffer. The name is validated to ensure that it does not contain invalid characters, and the variable is then located and copied to the buffer. The *shvvalue* RXSTRING length is updated with the total length of the variable. If the value was truncated because the buffer was not big enough, the *SHVTRUNC* bit is set and the length is changed to the correct length. If the variable is shorter than the length of the buffer, no padding takes place. If the name is a stem, the initial value of that stem, if any, is returned.

SHVNEWV is set if the variable did not exist before the operation, and in this case the value copied to the buffer is the derived name of the variable. The variable still does not exist after this call.

SHVDROPV and SHVSYDRO

Drop variable. The *shvname* RXSTRING describes the name of the variable to be dropped. *shvvalue* is not used. The name is validated to ensure that it does not contain invalid characters, and the variable is then dropped, if it exists. If the name given is a stem, all variables starting with that stem are dropped. If the variable does not exist the *shvnewv* flag is set.

SHVNEXTV

Fetch next variable. This function may be used to search through all the variables known to the language processor — all those of the current generation, excluding those “hidden” by PROCEDURE instructions. The order in which the variables are revealed is not specified.

The language processor maintains a pointer to its list of variables. This is reset to point to the first variable in the list whenever:

1. A host command is issued
2. An external function or subroutine is called
3. Any function other than *SHVNEXTV* is processed through the QREXVAR interface.

Whenever a *SHVNEXTV* function is processed, the name and value of the next variable available are copied to two buffers supplied by the caller.

shvname is a buffer into which the name is to be copied. The total length of the name is put into the length field of *shvname*, and if the name was truncated, because the buffer was not big enough, the *SHVTRUNC* bit is set and the length is changed to the correct length. If the name is shorter than the length of the buffer, no padding takes place. The value of the variable is copied to the *shvvalue* buffer area using exactly the same protocol as for the Fetch operation.

If *shvret* has *SHVLVAR* set, the end of the list of known variables has been found, the internal pointers have been reset, and no valid data has been copied to the user buffers. If *SHVTRUNC* is set, either the name or the value has been truncated.

By repeatedly executing the *SHVNEXTV* function, until the *SHVLVAR* flag is set, a user program may locate all the REXX variables of the current generation.

SHVPRIV Fetch private information. This interface is identical with the *SHVFETCH* interface, except that the name refers to certain fixed information items that are available. The entire name must be specified for the fetch to occur. The following names are recognized:

PARM The number of arguments supplied to the procedure will be placed in the caller's buffer. The number will be formatted as a character string.

PARM.n The *n*th argument string will be placed in the caller's buffer. If the *n*th argument was not supplied to the program, whether omitted, null, or fewer than N parameters were specified, then a null string will be returned.

SOURCE Fetch source string. The source string, as it is returned by the PARSE SOURCE instruction (see page 57), is copied to the user's buffer.

VERSION Fetch version string. The version string, as it is returned by the PARSE VERSION instruction, is copied to the user's buffer.

***CCSID** A string describing the national language environment of the current program. Four numbers, formatted as character strings separated by blanks, are placed in the user's buffer:

ccsid The Coded Character Set ID (CCSID) of the file containing the REXX program.

cpage The single-byte code page REXX uses to tokenize the program source.

etmode The OPTIONS ETMODE setting:

- 1—on, REXX will respect DBCS characters in the program source and variable names.
- 0—off, DBCS is not respected in source, SO/SI characters are not valid in variable names.

exmode The OPTIONS EXMODE setting:

- 1—on, when performing string operations, REXX will respect DBCS characters in variable data values.

- 0—off, DBCS in data is just another single-byte value.

***SRCPGMID** A string unique to the currently executing program. This string may be used to determine if subsequent QREXVAR operations are directed to the same instance of a REXX program.

SHVEXIT Set system exit return value. Many system exits must have the capability to return an exit value to the language processor. This interface is identical with the *SHVSET* interface, except that no name is specified.

SHVEXTFN

Set function return value. An external function written in a language other than REXX must have the capability to return a function value to the language processor. The value passed on this call is treated like the argument of a RETURN instruction. For more information on the RETURN instruction, see "RETURN" on page 65. This interface is identical with the *SHVSET* interface, except that no name is specified.

Notes:

1. QREXVAR only provides access to the variable pool of the latest active REXX program.
2. The full QREXVAR interface is only enabled during execution of commands and external routines (functions and subroutines), and during initiation, termination, command, and external function processing exits. It is enabled for *SHVEXIT* operations only from system exits. It is enabled for *SHVEXTFN* operations only from external functions. An attempt to call the QREXVAR entry point when no REXX procedure is active will result in a return code of -1, Invalid entry conditions.
3. An attempt to call the QREXVAR entry point with a pointer that is incorrect for the current program state, will result in exception message CPF9872 being signaled to the application. In addition, if the return code pointer is a valid pointer, then a return code of -1, Invalid Entry conditions, will also be set. Either or both of the above indicates that the requested Variable Pool operations were not successful.

Queuing Interfaces

Queue Application Programming Interface

The QREXQ interface is called with five parameters in the following order:

1. A one-byte field containing the function code that defines what function the caller is requesting. The following values are valid:

A	Add a line of data to the queue
N	Create a new queue buffer
P	Pull a line of data from the head of the queue
Q	Query the number of lines in the queue
R	Remove buffers from the queue.

2. A data buffer that either contains data to be placed on the queue, for the *Add* function, or is the target where data removed from the queue will be placed (for the *Pull* function). It is ignored for other functions.
3. An unsigned four-byte integer⁴ whose use depends on the function requested, as follows:

A	Contains the length of the data to be placed on the queue (INPUT)
N	Contains the number of the newly-created queue buffer (OUTPUT)
P	Contains the length of the data buffer (see item 2 above) into which the data will be placed (INPUT), then is set to the actual length of the data placed in the data buffer (OUTPUT)
Q	Contains the number of lines in the queue (OUTPUT)
R	If the operation flag is 1 (see item 4 below), contains the number of the queue buffer to be removed. Otherwise, it is unused.
4. An unsigned two-byte integer⁴ field containing an operation flag.
 - For the *Add* function, this flag indicates whether the data is to be added at the head of the queue in LIFO order (flag = 1) or at the tail of the queue in FIFO order (flag = 0).
 - For the *Remove* function, the flag indicates where the entire queue is to be cleared (flag=0) or if a specific queue buffer is to be removed (flag=1). Setting this flag=0 is the same as Remove REXX Buffer (RMVREXBUF) with a *ALL, and setting this flag=1 requires that the number of the buffer to be removed is indicated by setting the third parameter to R.
5. An unsigned two-byte integer, containing the return code from the requested function. The possible values are as follows:

0	OK - the function completed successfully
1	Not enough space available in the queue space object (valid only for <i>Add</i>). This is accompanied by escape message CPF7CF8.
2	Queue is empty (valid only for <i>Pull</i>)
3	Queue space object is damaged and should be reinitialized using the RMVREXBUF command. This is accompanied by escape message CPF7CF7.
4	Flag field is invalid (the FIFO/LIFO flag for <i>Add</i> , or the remove all/remove specific flag for <i>Remove</i>)
5	Data buffer is too small (valid only on <i>Pull</i> operations)
6	Data item is too large (valid only on <i>Add</i> operations)
7	Function code is invalid.

Sample Invocation of the Queuing Services

Here is an example of how the REXX queuing services might be called:

⁴ In RPG, COBOL, and PL/I, the unsigned byte is binary. In FORTRAN and C, the unsigned byte is an integer.

```
#pragma linkage(QREXQ,OS);
void QREXQ(char *, char *, unsigned int *, char *, unsigned short *);
char linetoadd[13] = "Line to add";
unsigned char function;
long int buffer_length;
char flag;
unsigned short return_code;

function = 'A';
buffer_length = 12;
flag = '\0';    /* Add to tail of queue */

QREXQ(&function, linetoadd, &buffer_length, &flag, &return_code);
/* Note: The following code assumes that a trap for escape messages */
/* is in effect that just ignores the escape and continues processing */
if (return_code == 1) {
    printf("Queue is full");
}
else if (return_code == 3) {
    printf("Queue is damaged");
}
else if (return_code) {
    printf("Unknown error with queue service");
}
}
```

Chapter 10. Debug Aids

In addition to the TRACE instruction, described in “TRACE” on page 70, there are the following debug 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 below for the exceptions). When the language processor pauses, three debug actions are available:

1. **Entering a null line** (no blanks even) 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-process the clause last traced. For example: if an IF clause is about to take the wrong branch, you can change the value of the variable(s) on which it depends, and then re-process it.

Once the clause has been re-processed, the language processor pauses again.

3. **Anything else entered** is treated as a **line** of one or more clauses, and processed immediately. 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 processed, 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:

- At any time, if TRACE 0 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 run without pause for debug input. TRACE n (that is, positive result) allows execution to continue, skipping the next n pauses (when interactive debug is or becomes active). TRACE -n (that is, negative result) allows execution to continue without pause and with tracing inhibited for n clauses that would otherwise be traced. The trace action selected by a TRACE instruction is saved and restored across subroutine calls. This means that if you are stepping through a program (say after using TRACE ?R to trace Results) and then enter a subroutine in which you have no interest, you can enter TRACE 0 to turn tracing off. No further instructions in the subroutine are traced, but on return to the caller, tracing is restored.

Debug Aids

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 processed in interactive debug you have considerable control over execution.

Some examples:

```
Say expr      /* displays the result of evaluating the      */
              /* expression.                               */

name=expr     /* alters the value of a variable.                       */

Trace 0       /* (or Trace with no options) turns off                 */
              /* interactive debug and all tracing.                   */

Trace ?A      /* turns off interactive debug but continue             */
              /* tracing all clauses.                                  */

exit          /* terminates execution of the program.                 */
```

Exceptions: Some clauses cannot safely be re-processed, 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-processed.)

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 here.

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 in a DO instruction, and the THEN (which acts as a clause terminator in this case) following an IF or WHEN clause.

Apart from these cases, only simple symbols that are the first token in a clause and that are not followed by an "=" or ":" are checked to see if they are instruction keywords; the symbols may be freely used elsewhere in clauses without being taken to be keywords.

Be careful of host commands or commands with the same name as REXX keywords, such as the CL command CALL. You can create problems if you inadvertently indicate the CL CALL command when you mean the REXX CALL instruction.

Several options are available to ensure that you indicate a command rather than an instruction.

At least the first words in command line can be enclosed within quotation marks, as shown in the following example:

```
'CALL QCMDEXC'
```

This also has an advantage in that it is more efficient, and the SIGNAL ON NOVALUE condition may be used to check the integrity of your program.

An alternative strategy is to precede such command strings with two adjacent quotation marks, which will have the effect of concatenating the null string on to the front, as shown in the following example.

```
''CALL QCMDEXC
```

Note, however, that a SIGNAL ON NOVALUE will be triggered because QCMDEXC is an unassigned value.

A third option is to enclose the entire expression, or the first symbol, in parentheses, as shown in the following example.

```
(CALL QCMDEXC)
```

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 processed host command (or command). Following the SIGNAL events, SYNTAX, ERROR, and FAILURE, RC is set to the code appropriate to the event: the syntax error number or the command return code. RC is unchanged following a NOVALUE or HALT event.

Note: Commands processed manually from debug mode do not cause the value of RC to change.

RESULT is set by a RETURN instruction in a subroutine that has been CALLED if the RETURN instruction specifies an expression. If the RETURN instruction has no expression on it, RESULT is dropped (becomes uninitialized.)

SIGL contains the line number of the clause currently executing when the last transfer of control to a label took place. (This could be caused by a SIGNAL, a CALL, 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. 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 called and the source of the program (which is available using the PARSE SOURCE instruction, see page 57).

In addition, PARSE VERSION (see page 57) 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 NUMERIC settings can be obtained using the DIGITS, FORM, and FUZZ built-in functions.

Appendix A. Double-Byte Character Set (DBCS) Support

A Double-Byte Character Set supports languages that have more characters than can be represented by 8 bits (such as Korean Hangeul and Japanese kanji). REXX has a full range of DBCS functions and handling techniques.

These include:

- Symbol and string handling capabilities with DBCS characters
- An option that allows DBCS characters in symbols, comments, and literal strings.
- An option that allows data strings to contain DBCS characters.
- A number of functions that specifically support the processing of DBCS character strings
- Defined DBCS enhancements to current instructions and functions.

Note: The use of DBCS does not affect the meaning of the built-in functions as described in Chapter 4, “Functions” on page 75. This explains how the characters in a result are obtained from the characters of the arguments by such actions as selecting, concatenating, and padding. The appendix describes how the resulting characters are represented as bytes. This internal representation is not usually seen if the results are printed. It may be seen if the results are displayed on certain terminals.

General Description

The following characteristics help define the rules used by DBCS to represent extended characters:

- Each DBCS character consists of 2 bytes.
- There are no DBCS control characters.
- The codes are within the ranges defined in the table, which shows the valid DBCS code for the DBCS blank. You cannot have a DBCS blank in a simple symbol, in the stem of a compound variable, or in a label.

Table 5. DBCS Ranges

Byte	EBCDIC
1st	X'41' to X'FE'
2nd	X'41' to X'FE'
DBCS blank	X'4040'

- DBCS alphanumeric and special symbols

A DBCS contains double-byte representation of alphanumeric and special symbols corresponding to those of the Single-Byte Character Set (SBCS). In EBCDIC, the first byte of a double-byte alphanumeric or special symbol is X'42' and the second is the same hex code as the corresponding EBCDIC code.

Here are some examples:

X'42C1' is an EBCDIC double-byte A

X'4281' is an EBCDIC double-byte a

X'427D' is an EBCDIC double-byte quote

- No case translation

In general, there is no concept of lowercase and uppercase in DBCS.

- Notational conventions

This appendix uses the following notational conventions:

DBCS character	->	.A .B .C .D
SBCS character	->	a b c d e
DBCS blank	->	'.'
EBCDIC shift-out (X'0E')	->	<
EBCDIC shift-in (X'0F')	->	>

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

Enabling DBCS Data Operations

The OPTIONS instruction controls how REXX regards DBCS data. To enable DBCS operations, use the EXMODE option. (See page 54 for more information.)

If OPTIONS ETMODE is in effect, the language processor does validation to ensure that SO and SI are paired. Otherwise, the contents of the comment are not checked. The comment delimiters (/ * and * /) must be SBCS characters.

Symbols and Strings

In DBCS, there are DBCS-only symbols and strings and mixed symbols and strings.

DBCS-Only Symbols and Mixed SBCS/DBCS Symbols

A DBCS-only symbol consists of only non-blank DBCS codes as indicated in Table 5 on page 177.

A mixed DBCS symbol is formed by a concatenation of SBCS symbols, DBCS-only symbols, and other mixed DBCS symbols. In EBCDIC, the SO and SI bracket the DBCS symbols and distinguish them from the SBCS symbols.

The default value of a DBCS symbol is the symbol itself, with SBCS characters translated to uppercase.

A *constant symbol* must begin with an SBCS digit (0–9) or an SBCS period. The delimiter (period) in a compound symbol must be an SBCS character.

In EBCDIC:

Mixed symbol	->	<.A.B>.
--------------	----	---------

DBCS-Only Strings and Mixed SBCS/DBCS Strings

A DBCS-only string consists of only DBCS characters. A mixed SBCS/DBCS string is formed by a combination of SBCS and DBCS characters. In EBCDIC, the SO and SI bracket the DBCS data and distinguish it from the SBCS data. Because the SO and SI are needed only in the mixed strings, they are not associated with the DBCS-only strings.

In EBCDIC:

DBCS-only string	->	.A.B.C
Mixed string	->	ab<.A.B>
Mixed string	->	<.A.B>
Mixed string	->	ab<.C.D>ef

Validation

The user must follow certain rules and conditions when using DBCS.

DBCS Symbol Validation

DBCS symbols are valid only if you comply with the following rules:

- The DBCS portion of the symbol must be an even number of bytes in length
- DBCS alphanumeric and special symbols are regarded as different to their corresponding SBCS characters. Only the SBCS characters are recognized by REXX in numbers, instruction keywords, or operators
- DBCS characters cannot be used as special characters in REXX
- SO and SI cannot be contiguous
- Nesting of SO or SI is not permitted
- SO and SI must be paired
- No part of a symbol consisting of DBCS characters may contain a DBCS blank.
- Each part of a symbol consisting of DBCS characters must be bracketed with SO and SI.

These examples show some possible misuses:

<.A.BC>	->	Incorrect because of odd byte length
<.A.B><.C>	->	Incorrect contiguous SO/SI
<>	->	Incorrect contiguous SO/SI (null DBCS symbol)
<.A<.B>.C>	->	Incorrectly nested SO/SI
<.A.B.C	->	Incorrect because SO/SI not paired
<.A. .B>	->	Incorrect because contains blank
' . A<.B><.C>	->	Incorrect symbol

Mixed String Validation

The validation of mixed strings depends on the instruction, operator, or function. If you use a mixed string with an instruction, operator, or function that does not allow mixed strings, this causes a **syntax error**.

The following rules must be followed for mixed string validation:

- DBCS strings must be an even number of bytes in length, unless you have SO and SI.

EBCDIC only:

- SO and SI must be paired in a string.
- Nesting of SO or SI is not permitted.

These examples show some possible misuses:

```
'ab<cd'      ->  INCORRECT - not paired
'<.A<.B>.C>' ->  INCORRECT - nested
'<.A.BC>'    ->  INCORRECT - odd byte length
```

The end of a comment delimiter is not found within DBCS character sequences. For example, when the program contains `/* < */`, then the `*/` is not recognized as ending the comment because the scanning is looking for the `>` (SI) to go with the `<` (SO) and not looking for `*/`.

When a variable is created, modified, or referred to in a REXX program under `OPTIONS EXMODE`, it is validated whether it contains a correct mixed string or not. When a referred variable contains a mixed string that is not valid, it depends on the instruction, function, or operator whether it causes a syntax error.

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 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>'
```

```

PARSE VAR x2 w1 '<>' w2
          w1  -> 'abc<>def <.A.B><><.C.D>'
```

```

PARSE VAR x2 w1 '<><>' w2
          w1  -> 'abc<>def <.A.B><><.C.D>'
```

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.

PUSH and QUEUE

The PUSH and QUEUE instructions add entries to the external data queue. Since a queue entry is limited to 32,767, *expression* may be truncated. If the truncation splits a DBCS string, REXX will ensure that the DBCS data integrity, that is the double-byte boundary, will be kept under OPTIONS EXMODE.

Note: In EBCDIC, DBCS data integrity includes maintaining the SO-SI pairing.

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>'
```

3. **Character concatenation**—String concatenation can only be done with valid mixed strings. In EBCDIC, adjacent SI and SO (or SO and SI) that are a result of string concatenation are removed. Even during implicit concatenation as in the DELSTR function, unnecessary SO and SI are removed.
4. **Character comparison**—Valid mixed strings are used when comparing strings on a character basis. A DBCS character is always considered greater than an SBCS one if they are compared. In all but the strict comparisons, SBCS blanks, DBCS blanks, and leading and trailing contiguous SO and SI (or SI and SO) in EBCDIC are removed. SBCS blanks may be added if the lengths are not identical.

In EBCDIC, contiguous SO and SI (or SI and SO) between nonblank characters are also removed for comparison.

Note: The strict comparison operators do not cause syntax errors even if you specify mixed strings that are not valid.

In EBCDIC:

```
'<.A>' = '<.A. >'    ->  1    /* true */
'<><><.A>' = '<.A><><>'' ->  1    /* true */
'<> <.A>' = '<.A>'    ->  1    /* true */
'<.A><><.B>' = '<.A.B>' ->  1    /* true */
'abc' < 'ab<. >'    ->  0    /* false */
```

5. **Word extraction from a string**—“Word” means that characters in a string are delimited by an SBCS or a DBCS blank.

In EBCDIC, leading and trailing contiguous SO and SI (or SI and SO) are also removed when *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>'

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 75.

ABBREV

In EBCDIC:

ABBREV('<.A.B.C>', '<.A.B>') -> 1
 ABBREV('<.A.B.C>', '<.A.C>') -> 0
 ABBREV('<.A><.B.C>', '<.A.B>') -> 1
 ABBREV('aa<>bbccdd', 'aabbcc') -> 1

Applying the character comparison and character extraction from a string rules.

COMPARE

In EBCDIC:

COMPARE('<.A.B.C>', '<.A.B><.C>') -> 0
 COMPARE('<.A.B.C>', '<.A.B.D>') -> 3
 COMPARE('ab<>cde', 'abcdx') -> 5
 COMPARE('<.A><>', '<.A>', '<. >') -> 0

Applying the character concatenation for padding, character extraction from a string, and character comparison rules.

COPIES

In EBCDIC:

COPIES('<.A.B>', 2) -> '<.A.B.A.B>'
 COPIES('<.A><.B>', 2) -> '<.A><.B.A><.B>'
 COPIES('<.A.B><>', 2) -> '<.A.B><.A.B><>'

Applying the character concatenation rule. 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>'
```

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'
```

Applying the character concatenation for padding and character extraction from a string rules.

LENGTH

In EBCDIC:

```
LENGTH('<.A.B><.C.D><<') -> 4
```

Applying the counting characters rule.

REVERSE

In EBCDIC:

```
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>'
```

Applying the word extraction from a string and character concatenation rules.

STRIP

In EBCDIC:

```
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><>'
```

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>'
```

Applying the word extraction from a string and character concatenation rules.

SYMBOL

In EBCDIC:

```
Drop A.3 ; <.A.B>=3
SYMBOL('<.A.B>') -> 'VAR'
SYMBOL(<.A.B>) -> 'LIT' /* has tested "3" */
SYMBOL('a.<.A.B>') -> 'LIT' /* has tested A.3 */
```

TRANSLATE

In EBCDIC:

```
TRANSLATE('abcd','<.A.B.C>','abc') -> '<.A.B.C>d'
TRANSLATE('abcd','<<<.A.B.C>','abc') -> '<.A.B.C>d'
TRANSLATE('abcd','<<<.A.B.C>','ab<>c') -> '<.A.B.C>d'
TRANSLATE('a<>bcd','<<<.A.B.C>','ab<>c') -> '<.A.B.C>d'
TRANSLATE('a<>xcd','<<<.A.B.C>','ab<>c') -> '<.A>x<.C>d'
```

Applying the character extraction from a string, character comparison, and character concatenation rules.

VALUE

In EBCDIC:

```
Drop A3 ; <.A.B>=3 ; fred='<.A.B>'
VALUE('fred') -> '<.A.B>' /* looks up FRED */
VALUE(fred) -> '3' /* looks up <.A.B> */
VALUE('a'<.A.B>) -> 'A3'
```

VERIFY

In EBCDIC:

```
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:

W = '<>.A. .B>.C. .D>'

WORD(W,1) -> '<.A>'
WORDINDEX(W,1) -> 2
WORDLENGTH(W,1) -> 1

Y = '<>.A. .B>.C. .D>'

WORD(Y,1) -> '<.A>'
WORDINDEX(Y,1) -> 1
WORDLENGTH(Y,1) -> 1

Z = '<.A .B>.C> <.D>'

WORD(Z,2) -> '<.B>.C>'
WORDINDEX(Z,2) -> 3
WORDLENGTH(Z,2) -> 2

Applying the word extraction from a string and (for WORDINDEX and WORDLENGTH) counting characters rules.

WORDS

In EBCDIC:

W = '<>.A. .B>.C. .D>'

WORDS(W) -> 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

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).

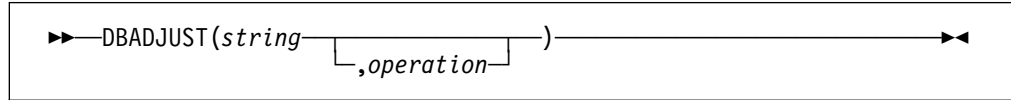
Counting Option

In EBCDIC, when specified in the functions, the counting option can control whether 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.

Function Descriptions

The following are the DBCS functions and their descriptions.

DBADJUST



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 highlighted letter is needed; all characters following it are ignored.

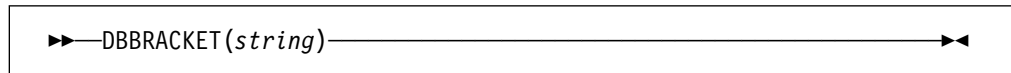
Blank changes contiguous characters to blanks (X'4040').

Remove removes contiguous characters, and is the default.

Here are some EBCDIC examples:

```
DBADJUST('<.A><.B>a<>b', 'B')    ->   '<.A. .B>a  b'  
DBADJUST('<.A><.B>a<>b', 'R')    ->   '<.A.B>ab'  
DBADJUST('<><.A.B>', 'B')        ->   '<. .A.B>'
```

DBBRACKET

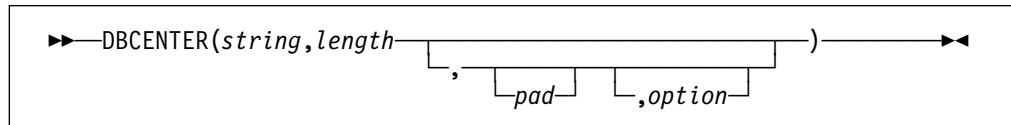


In EBCDIC, adds SO and SI brackets to a DBCS-only string. If *string* is not a DBCS-only string, a SYNTAX error results. That is, the input string must be an even number of bytes in length and each byte must be a valid DBCS value.

Here are some EBCDIC examples:

```
DBBRACKET(' .A.B')           ->   '<.A.B>'  
DBBRACKET('abc')             ->   SYNTAX error  
DBBRACKET('<.A.B>')           ->   SYNTAX error
```

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.

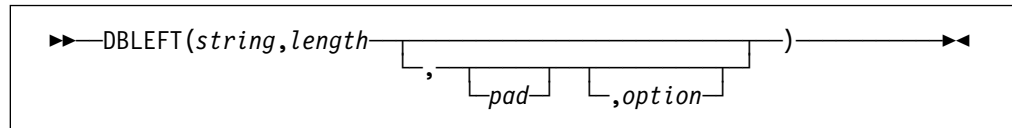
Here are some EBCDIC examples:

```

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>'

```

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.

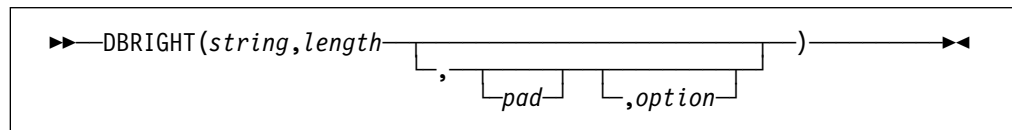
Here are some EBCDIC examples:

```

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>'

```

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.

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>'

```

DBRLEFT

```

▶▶—DBRLEFT(string,length—_,option)—▶▶

```

returns the remainder from the DBLEFT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBRLEFT('ab<.A.B>',4)           -> '<.B>'
DBRLEFT('ab<.A.B>',3)           -> '<.A.B>'
DBRLEFT('ab<.A.B>',4,'Y')       -> '<.A.B>'
DBRLEFT('ab<.A.B>',3,'Y')       -> '<.A.B>'
DBRLEFT('ab<.A.B>',8)           -> ''
DBRLEFT('ab<.A.B>',9,'Y')       -> ''

```

DBRRIGHT

```

▶▶—DBRRIGHT(string,length—_,option)—▶▶

```

returns the remainder from the DBRIGHT function of *string*. If *length* is greater than the length of *string*, returns a null string.

The *option* controls the counting rule. **Y** counts SO and SI within mixed strings as one each. **N** does not count the SO and SI and is the default.

Here are some EBCDIC examples:

```

DBRRIGHT('ab<.A.B>',4)          -> 'ab'
DBRRIGHT('ab<.A.B>',3)          -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',5)          -> 'a'
DBRRIGHT('ab<.A.B>',4,'Y')      -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',5,'Y')      -> 'ab<.A>'
DBRRIGHT('ab<.A.B>',8)          -> ''
DBRRIGHT('ab<.A.B>',8,'Y')      -> ''

```

DBTODBCS

▶▶ DBTODBCS(*string*) ◀◀

converts all passed, valid SBCS characters (including the SBCS blank) within *string* to the corresponding DBCS equivalents. Other single-byte codes and all DBCS characters are not changed. In EBCDIC, SO and SI brackets are added and removed where appropriate.

Here are some EBCDIC examples:

```
DBTODBCS('Rexx 1988')    ->  '<.R.e.x.x. .1.9.8.8>'
DBTODBCS('<.A> <.B>')    ->  '<.A. .B>'
```

Note: In these examples, the *.x* is the DBCS character corresponding to an SBCS *x*.

DBTOSBCS

▶▶ DBTOSBCS(*string*) ◀◀

converts all passed, valid DBCS characters (including the DBCS blank) within *string* to the corresponding SBCS equivalents. Other DBCS characters and all SBCS characters are not changed. In EBCDIC, SO and SI brackets are removed where appropriate.

Here are some EBCDIC examples:

```
DBTOSBCS('<.S.d>/<.2.-.1>') ->  'Sd/2-1'
DBTOSBCS('<.X. .Y>')        ->  '<.X> <.Y>'
```

Note: In these examples, the *.d* is the DBCS character corresponding to an SBCS *d*. But the *.X* and *.Y* do not have corresponding SBCS characters and are not converted.

DBUNBRACKET

▶▶ DBUNBRACKET(*string*) ◀◀

In EBCDIC, removes the SO and SI brackets from a DBCS-only *string* enclosed by SO and SI brackets. If the *string* is not bracketed, a SYNTAX error results.

Here are some EBCDIC examples:

```
DBUNBRACKET('<.A.B>')    ->  '.A.B'
DBUNBRACKET('ab<.A>')  ->  SYNTAX error
```

DBVALIDATE

▶▶ DBVALIDATE(*string* [, 'C']) ◀◀

returns 1 if the *string* is a valid mixed string or SBCS string. Otherwise, returns 0. Mixed string validation rules are:

1. Only valid DBCS character codes
2. DBCS string is an even number of bytes in length
3. EBCDIC only — Proper SO and SI pairing.

In EBCDIC, if **C** is omitted, only the leftmost byte of each DBCS character is checked to see that it falls in the valid range for the implementation it is being run on (that is, in EBCDIC, the leftmost byte range is from X'41' to X'FE').

Here are some EBCDIC examples:

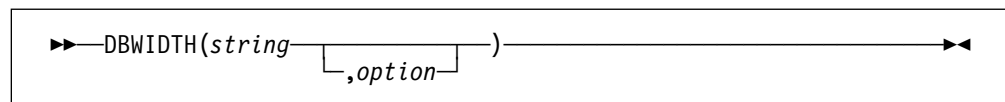
```
z='abc<de'
```

```
DBVALIDATE('ab<.A.B>')    -> 1
DBVALIDATE(z)              -> 0
```

```
y='C1C20E111213140F'X
```

```
DBVALIDATE(y)              -> 1
DBVALIDATE(y,'C')          -> 0
```

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.

Here are some EBCDIC examples:

```
DBWIDTH('ab<.A.B>', 'Y')  -> 8
DBWIDTH('ab<.A.B>', 'N')  -> 6
```


Appendix B. REXX/400 Implementation Limits

The SAA Procedures Language Definition defines a set of minimum values for specific features of the language. All implementations must either meet or exceed these limits to conform to the language definitions. The following table describes the limits imposed by the REXX/400 implementation.

Table 6. AS/400 Implementation Limits

Item	AS/400
Literal strings	250 bytes
Symbol (variable name) length	250 bytes
Nesting control structures	100
Call arguments	20
MIN and MAX function arguments	20
Queue entries	Approximately 500,000 (see note 1)
Queue entry length	32,767 bytes
NUMERIC DIGITS value	space available (see note 2)
Notational exponent value	999 999 999
Hexadecimal strings	250 bytes
Binary strings	100 bytes
C2D input string	250 bytes
D2C output string	250 bytes
D2X input string	500 bytes
X2D input string	500 bytes

Notes:

1. This assumes that the entries are an average of 20 bytes each in length. The total size of the queue is limited to 15.5MB.
2. There is no absolute limit on the NUMERIC DIGITS value. The only limit is that the storage for all REXX variables within an invocation of the language processor may not exceed 16MB.

Implementation Limits

Index

Special Characters

, (comma)
as continuation character 14
in CALL instruction 35
in function calls 75
in parsing template list 33, 121
separator of arguments 35, 75

: (colon)
as a special character 13
in a label 20

? prefix on TRACE option 71

/ (division operator) 16, 129

// (remainder operator) 16, 132

. (period)
as placeholder in parsing 113
causing substitution in variable names 22
in numbers 128

***** (multiplication operator) 16, 129

. tracing flag 73

****** (power operator) 16, 131

**** (NOT operator) 18

\< (not less than operator) 17

\<< (strictly not less than operator) 18

\= (not equal operator) 17

\== (strictly not equal operator) 17

\> (not greater than operator) 17

\>> (strictly not greater than operator) 18

& (AND logical operator) 18

&& (exclusive OR operator) 18

% (integer division operator) 16, 132

+ (addition operator) 16, 129

+++ tracing flag 73

< (less than operator) 17

<< (strictly less than operator) 17

<<= (strictly less than or equal operator) 18

<= (less than or equal operator) 17

<> (less than or greater than operator) 17

= (equal sign)
assignment indicator 21
equal operator 17
immediate debug command 173
in DO instruction 39
in parsing template 115

== (strictly equal operator) 16, 17, 129

- (subtraction operator) 16

> (greater than operator) 17

>> tracing flag 73

>< (greater than or less than operator) 17

>= (greater than or equal operator) 17

>> (strictly greater than operator) 17

>>= (strictly greater than or equal operator) 17

>>> tracing flag 73

>C> tracing flag 73

>F> tracing flag 73

>L> tracing flag 73

>O> tracing flag 73

>P> tracing flag 73

>V> tracing flag 73

| (inclusive OR operator) 18

|| (concatenation operator) 16

Numerics

400/REXX

See REstructured eXtended eXecutor/400
(REXX/400)

A

ABBREV function

description 79
example 79
testing abbreviations 79
using to select a default 79

abbreviations

testing with ABBREV function 79

ABS function

description 79
example 79

absolute value

finding using ABS function 79
function 79
used with power 131

abuttal 16

action taken when a condition is not trapped 138

action taken when a condition is trapped 138

active loops 49

addition

description 130
operator 16

additional operator examples 132

ADDRESS function

description 79
determining current environment 79
example 80

ADDRESS instruction

description 30
example 30, 31
settings saved during subroutine calls 37

address setting 31, 37

ADDREXBUF command 145

- advanced topics in parsing** 121
- algebraic precedence** 18
- alphabetic character word options in TRACE** 71
- alphabets**
 - checking with DATATYPE 86
 - used as symbols 11
- alphanumeric checking with DATATYPE** 86
- altering**
 - flow within a repetitive DO loop 49
 - special variables 26
 - TRACE setting 101
- AND, logical operator** 18
- ANDing character strings together** 81
- application program interfaces**
 - command interface 153, 154
 - data types and structures 155
 - external functions and subroutines 156
 - queuing interface 170
 - system exit interface 157
 - variable pool interface 166
- ARG function**
 - description 80
 - example 80
- ARG instruction**
 - description 33
 - example 33
- ARG option of PARSE instruction** 56
- arguments** 156
 - checking with ARG function 80
 - of functions 33, 75
 - of programs 33
 - of subroutines 33, 35
 - passing to functions 75
 - retrieving with ARG function 80
 - retrieving with ARG instruction 33
 - retrieving with the PARSE ARG instruction 56
- arithmetic**
 - basic operator examples 131
 - comparisons 133
 - errors 136
 - exponential notation example 134
 - numeric comparisons, example 133
 - NUMERIC settings 52
 - operation rules 129
 - operator examples 132
 - operators 16, 127, 129
 - overflow 136
 - precision 128
 - underflow 136
 - whole numbers 135
- array**
 - initialization of 24
 - setting up 22
- AS/400 System**
 - issuing commands to 30

- assigning**
 - data to variables 56
- assignment**
 - description 21
 - indicator (=) 21
 - multiple assignments 116
 - of compound variables 22, 24
- associative storage** 22

B

- B2X function**
 - description 82
 - example 83
- backslash, use of** 13, 18
- Base option of DATE function** 87
- basic operator examples** 131
- binary**
 - description 10
 - digits 10
 - strings
 - implementation maximum 11
 - nibbles 10
 - to hexadecimal conversion 82
- BITAND function**
 - description 81
 - example 81
 - logical bit operations 81
- BITOR function**
 - description 81
 - example 82
 - logical bit operations, BITOR 81
- bits checked using DATATYPE** 86
- BITXOR function**
 - description 82
 - example 82
 - logical bit operations, BITXOR 82
- blanks**
 - adjacent to special character 8
 - as concatenation operator 16
 - in parsing, treatment of 112
 - removal with STRIP function 98
- boolean operations** 18
- bottom of program reached during execution** 45
- bracketed DBCS strings**
 - DBBRACKET function 187
 - DBUNBRACKET function 190
- built-in functions**
 - ABBREV 79
 - ABS 79
 - ADDRESS 79
 - ARG 80
 - B2X 82
 - BITAND 81
 - BITOR 81
 - BITXOR 82

built-in functions *(continued)*

C2D 85
C2X 86
calling 35
CENTER 83
CENTRE 83
COMPARE 83
CONDITION 84
COPIES 85
D2C 89
D2X 90
DATATYPE 86
DATE 87
DBCS functions 187
definition 35
DELSTR 88
DELWORD 89
description 78
DIGITS 89
ERRORTXT 91
FORM 91
FORMAT 91
FUZZ 92
INSERT 93
LASTPOS 93
LEFT 93
LENGTH 94
MAX 94
MIN 94
OVERLAY 95
POS 95
QUEUED 95
RANDOM 96
REVERSE 96
RIGHT 97
SIGN 97
SOURCELINE 97
SPACE 98
STRIP 98
SUBSTR 99
SUBWORD 99
SYMBOL 99
TIME 100
TRACE 101
TRANSLATE 102
TRUNC 103
VALUE 103
VERIFY 104
WORD 104
WORDINDEX 105
WORDLENGTH 105
WORDPOS 105
WORDS 106
X2B 106
X2C 107
X2D 107

built-in functions *(continued)*

XRANGE 106

BY phrase of DO instruction 39**C****C2D function**

description 85
example 85
implementation maximum 85

C2X function

description 86
example 86

CALL instruction

description 35
example 37
implementation maximum 38

calls

recursive 36

CCSID (coded character set identifier)

See coded character set identifier (CCSID)

CENTER function

description 83
example 83

centering a string using

CENTER function 83
CENTRE function 83

CENTRE function

description 83
example 83

changing destination of commands 30**character**

definition 8
position of a string 93
removal with STRIP function 98
strings, ANDing 81
strings, exclusive-ORing 82
strings, ORing 81
to decimal conversion 85
to hexadecimal conversion 86
word options, alphabetic in TRACE 71

character input and output 143**checking arguments with ARG function** 80**CHGCMD command** 148, 157**CL Command Environment (COMMAND)** 153**CL variables in REXX programs** 149**clauses**

assignment 20, 21
commands 21
continuation of 14
description 8, 20
instructions 20
keyword instructions 20
labels 20
null 20

clock, elapsed time

See elapsed-time clock

code page 8**coded character set identifier (CCSID)**

mixed 54

with ETMODE 54

with NOETMODE 54

collating sequence using XRANGE 106**collections of variables 103****colon**

as a special character 13

as label terminators 20

in a label 20

combining string and positional patterns 121**comma**

as continuation character 14

in CALL instruction 35

in function calls 75

in parsing template list 33, 121

separator of arguments 35, 75

command

addressing of 30

alternative destinations 25

clause 21

destination of 30

errors, trapping 137

issuing to host 25

command environments

See *also* environment

COMMAND 153

CPICOMM 152, 154

EXECSQL 152, 154

user-defined 154

command inhibition

See TRACE instruction

comments

description 8

examples 8

common programming interface 2**COMPARE function**

description 83

example 83

comparisons

description 17

numeric, example 133

of numbers 17, 133

of strings

using COMPARE 83

compound

symbols 22

variable

description 22

setting new value 24

concatenation

of strings 16

operator

|| 16

concatenation (continued)

operator (continued)

abuttal 16

blank 16

conceptual overview of parsing 122**condition**

action taken when not trapped 138

action taken when trapped 138

definition 137

ERROR 137

FAILURE 137

HALT 137

information 140

information, definition 37

NOVALUE 138

saved during subroutine calls 37

SYNTAX 138

trap information using CONDITION 84

trapping of 137

traps, notes 140

CONDITION function

description 84

example 84

conditional

loops 39

phrase 42

console

reading from with PULL 62

writing to with SAY 66

constant symbols 22**content addressable storage 22****contents of this book xi****continuation**

character 14

clauses 14

example 14

of data for display 66

control variable 40**controlled loops 40****conversion**

binary to hexadecimal 82

character to decimal 85

character to hexadecimal 86

conversion functions 78

decimal to character 89

decimal to hexadecimal 90

formatting numbers 91

functions 108

hexadecimal to binary 106

hexadecimal to character 107

hexadecimal to decimal 107

COPIES function

description 85

example 85

copying a string using COPIES 85

counting

- option in DBCS 186
- words in a string 106

CPI-Communications command environment (CPICOMM) 154

CPICOMM command environment 152

CPICOMM, use of 30

CRTCMD command 157

D

D2C function

- description 89
- example 90
- implementation maximum 90

D2X function

- description 90
- example 90
- implementation maximum 90

data

- length 15
- terms 15

DATATYPE function

- description 86
- example 87

date and version of the language processor 57

DATE function

- description 87
- example 88

DBADJUST function

- description 187
- example 187

DBBRACKET function

- description 187
- example 187

DBCENTER function

- description 187
- example 188

DBCS

- built-in function descriptions 187
- built-in function examples 183
- characters 177
- counting option 186
- description 177
- enabling data operations 178
- EXMODE 178
- function handling 181
- functions
 - DBADJUST 187
 - DBBRACKET 187
 - DBCENTER 187
 - DBLEFT 188
 - DBRIGHT 188
 - DBRLEFT 189
 - DBRRIGHT 189
 - DBTODBCS 190
 - DBTOSBCS 190

DBCS (continued)

functions (continued)

- DBUNBRACKET 190
- DBVALIDATE 191
- DBWIDTH 191

handling 177

- instruction examples 180
- mixed SBCS/DBCS string 179
- mixed string validation example 180
- mixed symbol 178
- notational conventions 178
- parsing characters 122
- processing functions 186
- ranges 11
- SBCS strings 177
- shift-in (SI) characters 178, 182
- shift-out (SO) characters 178, 182
- string, DBCS-only 179
- string, mixed SBCS/DBCS 179
- strings 54, 177
- strings and symbols 178
- support 191
- symbol validation and example 179
- symbol, DBCS-only 178
- symbol, mixed 178
- symbols and strings 178
- validation, mixed string 179

DBCS-only string 86

DBLEFT function

- description 188
- example 188

DBRIGHT function

- description 188
- example 188

DBRLEFT function

- description 189
- example 189

DBRRIGHT function

- description 189
- example 189

DBTODBCS function

- description 190

DBTOSBCS function

- description 190
- example 190

DBUNBRACKET function

- description 190
- example 190

DBVALIDATE function

- description 191
- example 191

DBWIDTH function

- description 191
- example 191

debugging programs

- See interactive debug

debugging programs *(continued)*

See TRACE instruction

decimal

- arithmetic 127—136
- to character conversion 89
- to hexadecimal conversion 90

default

- environment 25
- selecting with ABBREV function 79

default character streams

See character input and output

delayed state

- description 137

deleting

- part of a string 88
- words from a string 89

delimiters in a clause

- See colon
- See semicolons

DELSTR function

- description 88
- example 89

DELWORD function

- description 89
- example 89

derived names of variables 22

description

- of built-in functions for DBCS 187

DIGITS function

- description 89
- example 89

DIGITS option of NUMERIC instruction 52, 128

displaying data

See SAY instruction

division

- description 130
- operator 16

DO instruction

- See *also* loops
- description 39
- example 41

Double-Byte Character Set

See DBCS

DROP instruction

- description 44
- example 44

dummy instruction

See NOP instruction

E

elapsed-time clock

- measuring intervals with 100
- saved during subroutine calls 37

ELSE keyword

See IF instruction

END clause

- See *also* DO instruction
- See *also* SELECT instruction
- specifying control variable 40

engineering notation 134

environment

- addressing of 30
- default 31, 57
- determining current using ADDRESS function 79
- name, definition 30
- SAA supported 2
- temporary change of 30

equal

- operator 17
- sign
 - in parsing template 114, 115
 - to indicate assignment 13, 21

equality, testing of 17

error

- definition 26
- during execution of functions 77
- from commands 26
- messages
 - retrieving with ERRORTXT 91
- traceback after 73
- trapping 137

ERROR condition of SIGNAL and CALL instructions 141

ERRORTXT function

- description 91
- example 91

ETMODE 54

European option of DATE function 87

evaluation of expressions 15

example

- ABBREV function 79
- ABS function 79
- ADDRESS function 80
- ADDRESS instruction 30, 31
- ARG function 80
- ARG instruction 33
- B2X function 83
- basic arithmetic operators 131
- BITAND function 81
- BITOR function 82
- BITXOR function 82
- built-in function in DBCS 183
- C2D function 85
- C2X function 86
- CALL instruction 37
- CENTER function 83
- CENTRE function 83
- character 14
- clauses 14
- combining positional pattern and parsing into words 117

example (continued)

- combining string and positional patterns 121
- combining string pattern and parsing into words 116
- comments 8
- COMPARE function 83
- CONDITION function 84
- continuation 14
- COPIES function 85
- D2C function 90
- D2X function 90
- DATATYPE function 87
- DATE function 88
- DBADJUST function 187
- DBBRACKET function 187
- DBCENTER function 188
- DBCS instruction 180
- DBLEFT function 188
- DBRIGHT function 188
- DBRLEFT function 189
- DBRRIGHT function 189
- DBTOSBCS function 190
- DBUNBRACKET function 190
- DBVALIDATE function 191
- DBWIDTH function 191
- DELSTR function 89
- DELWORD function 89
- DIGITS function 89
- DO instruction 41
- DROP instruction 44
- ERRORTEXT function 91
- EXIT instruction 45
- exponential notation 134
- expressions 19
- FORM function 91
- FORMAT function 92
- FUZZ function 92
- IF instruction 46
- INSERT function 93
- INTERPRET instruction 47, 48
- ITERATE instruction 49
- LASTPOS function 93
- LEAVE instruction 50
- LEFT function 93
- LENGTH function 94
- MAX function 94
- MIN function 94
- mixed string validation 180
- NOP instruction 51
- numeric comparisons 133
- OVERLAY function 95
- parsing instructions 119
- parsing multiple strings in a subroutine 121
- period as a placeholder 113
- POS function 95
- PROCEDURE instruction 60

example (continued)

- PULL instruction 62
- PUSH instruction 63
- QUEUE instruction 64
- QUEUED function 95
- RANDOM function 96
- REVERSE function 96
- RIGHT function 97
- SAY instruction 66
- SELECT instruction 67
- SIGL, special variable 142
- SIGN function 97
- SIGNAL instruction 68
- simple templates, parsing 111
- SOURCELINE function 97
- SPACE function 98
- special characters 13
- STRIP function 98
- SUBSTR function 99
- SUBWORD function 99
- SYMBOL function 100
- symbol validation 179
- templates containing positional patterns 114
- templates containing string patterns 113
- TIME function 101
- TRACE function 102
- TRACE instruction 72
- TRANSLATE function 102
- TRUNC function 103
- using a variable as a positional pattern 118
- using a variable as a string pattern 118
- VALUE function 103
- VERIFY function 104
- WORD function 105
- WORDINDEX function 105
- WORDLENGTH function 105
- WORDPOS function 105
- WORDS function 106
- X2B function 106
- X2C function 107
- X2D function 107
- XRANGE function 106
- exception conditions saved during subroutine calls 37**
- exclusive OR operator 18**
- exclusive-ORing character strings together 82**
- EXECSQL command environment 152**
- EXECSQL, use of 30**
- execution**
 - by language processor 7
 - of data 47
- EXIT instruction**
 - description 45
 - example 45
- EXMODE**
 - in DBCS 178

EXMODE *(continued)*

with OPTIONS instruction 54

exponential notation

description 127, 133

example 134

usage 12

exponentiation

description 133

operator 16

EXPOSE option of PROCEDURE instruction 59**exposed variable** 59**expressions**

evaluation 15

examples 19

parsing of 57

results of 15

tracing results of 71

external

application program interface 156

arguments within 156

data queue

counting lines in 95

maximum length of items 63, 64

reading from with PULL 62

writing to with PUSH 63

writing to with QUEUE 64

functions

description 76

return values from 157

routine

calling 35

definition 35

search order 156

subroutines

description 76

variables

access with VALUE function 103

external character streams

See character input and output

external data queue 144

damage to 170

extracting

substring 99

word from a string 104

words from a string 99

F**FAILURE condition of SIGNAL and CALL instructions** 137, 141**failure, definition** 26**FIFO (first-in/first-out) stacking** 64**files**

See character input and output

finding

mismatch using COMPARE 83

finding *(continued)*

string in another string 95

string length 94

word length 105

flags, tracing

- 73

+++ 73

>.> 73

>>> 73

>C> 73

>F> 73

>L> 73

>O> 73

>P> 73

>V> 73

flow of control

unusual, with CALL 137

unusual, with SIGNAL 137

with CALL/RETURN 35

with DO construct 39

with IF construct 46

with SELECT construct 67

FOR phrase of DO instruction 39**FOREVER repetitor on DO instruction** 39**FORM function**

description 91

example 91

FORM option of NUMERIC instruction 52, 135**FORMAT function**

description 91

example 92

formatting

DBCS blank adjustments 187

DBCS bracket adding 187

DBCS bracket stripping 190

DBCS EBCDIC to DBCS 190

DBCS string width 191

DBCS strings to SBCS 190

numbers for display 91

numbers with TRUNC 103

of output during tracing 73

text centering 83

text left justification 93, 188

text left remainder justification 189

text right justification 97, 188

text right remainder justification 189

text spacing 98

text validation function 191

function, built-in

See built-in functions

functions 75—108

ABS 79

ADDRESS 79

ARG 80

AS/400 Specific 108

B2X 82

functions (continued)

- BITAND 81
- BITOR 81
- BITXOR 82
- built-in 79—107
- built-in, description 78
- C2D 85
- C2X 86
- call, definition 75
- calling 75
- CENTER 83
- CENTRE 83
- COMPARE 83
- CONDITION 84
- COPIES 85
- D2C 89
- D2X 90
- DATATYPE 86
- DATE 87
- definition 75
- DELSTR 88
- DELWORD 89
- description 75
- DIGITS 89
- ERRORTXT 91
- external 76
- forcing built-in or external reference 77
- FORM 91
- FORMAT 91
- FUZZ 92
- INSERT 93
- internal 76
- LASTPOS 93
- LEFT 93
- LENGTH 94
- MAX 94
- MIN 94
- numeric arguments of 135
- OVERLAY 95
- POS 95
- processing in DBCS 186
- QUEUED 95
- RANDOM 96
- return from 65
- REVERSE 96
- RIGHT 97
- SETMSGRC 108
- SIGN 97
- SOURCELINE 97
- SPACE 98
- STRIP 98
- SUBSTR 99
- SUBWORD 99
- SYMBOL 99
- TIME 100
- TRACE 101

functions (continued)

- TRANSLATE 102
- TRUNC 103
- VALUE 103
- variables in 59
- VERIFY 104
- WORD 104
- WORDINDEX 105
- WORDLENGTH 105
- WORDPOS 105
- WORDS 106
- X2B 106
- X2C 107
- X2D 107
- XRANGE 106

FUZZ

- controlling numeric comparison 133
- option of NUMERIC instruction 52, 133

FUZZ function

- description 92
- example 92

G

general concepts 7—27

global variables

- access with VALUE function 103

GOTO, unusual 137

greater than operator 17

greater than or equal operator (>=) 17

greater than or less than operator (><) 17

group, DO 39

grouping instructions to run repetitively 39

guard digit 129

H

HALT condition of SIGNAL and CALL

- instructions** 137, 141

halt, trapping 137

hexadecimal

- See *also* conversion

- checking with DATATYPE 86

- description 10

- digits 10

- strings

- implementation maximum 10

- to binary, converting with X2B 106

- to character, converting with X2C 107

- to decimal, converting with X2D 107

host commands

- issuing commands to underlying operating system 25

hours calculated from midnight 100

how to use this book 2

I

IF instruction

- description 46
- example 46

ILE Session Manager display

- features provided by 149
- input to PARSE PULL instruction 57
- input to PULL instruction 62
- output of SAY instruction 66

implementation maximum

- binary strings 11
- C2D function 85
- CALL instruction 38
- D2C function 90
- D2X function 90
- hexadecimal strings 10
- literal strings 10
- MAX function 94
- MIN function 94
- numbers 13
- operator characters 15, 23
- PUSH instruction 63
- QUEUE instruction 64
- storage limit 7
- symbols 12
- X2D function 108

implied semicolons 14

imprecise numeric comparison 133

inclusive OR operator 18

indefinite loops 41

indentation during tracing 73

indirect evaluation of data 47

inequality, testing of 17

infinite loops 39

information, more REXX 4

initialization

- of arrays 24
- of compound variables 24

input and output model 143

input and output streams 143

input from the user 143

INSERT function

- description 93
- example 93

inserting a string into another 93

instructions

- ADDRESS 30
- ARG 33
- CALL 35
- definition 20
- DO 39
- DROP 44
- EXIT 45
- IF 46
- INTERPRET 47

instructions (continued)

- ITERATE 49
 - keyword 20
 - description 29
- LEAVE 50
- NOP 51
- NUMERIC 52
- OPTIONS 54
- PARSE 56
- parsing, summary 119
- PROCEDURE 59
- PULL 62
- PUSH 63
- QUEUE 64
- RETURN 65
- SAY 66
- SELECT 67
- SIGNAL 68
- TRACE 70

integer

- arithmetic 127—136
- division
 - description 127, 132
 - operator 16

interactive debug 70

- See *also* TRACE instruction

internal

- functions
 - description 76
 - return from 65
 - variables in 59
- routine
 - calling 35
 - definition 35

INTERPRET instruction

- description 47
- example 47, 48

interpretive execution of data 47

invoking

- built-in functions 35
- routines 35

ITERATE instruction

- See *also* DO instruction
- description 49
- example 49
- use of variable on 49

J

justification, text right, RIGHT function 97

K

keyword

- See *also* instructions
- conflict with commands 175

keyword (*continued*)

- description 29
- mixed case 29
- reservation of 175

L

label

- as target of CALL 35
- as target of SIGNAL 68
- description 20
- duplicate 68
- in INTERPRET instruction 47
- search algorithm 68

language

- processor date and version 57
- structure and syntax 8

language limitations 193

language processor, starting the 148

- from a CDO 148
- using QREXX 151
- using the STRREXPRC command 148

LASTPOS function

- description 93
- example 93

leading

- blank removal with STRIP function 98
- zeros
 - adding with the RIGHT function 97
 - removing with STRIP function 98

LEAVE instruction

- See *also* DO instruction
- description 50
- example 50
- use of variable on 50

leaving your program 45

LEFT function

- description 93
- example 93

LENGTH function

- description 94
- example 94

less than operator (<) 17

less than or equal operator (<=) 17

less than or greater than operator (<>) 17

LIFO (last-in/first-out) stacking 63

limitations, language 193

line input and output

- See character input and output

LINEIN option of PARSE instruction 56

lines from stream 56

list

- template
 - ARG instruction 33
 - PARSE instruction 56
 - PULL instruction 62

literal string

- description 9
- implementation maximum 10
- patterns 113

locating

- string in another string 95
- word in a string 105

logical

- bit operations
 - BITAND 81
 - BITOR 81
 - BITXOR 82
- operations 18

loops

- See *also* DO instruction
- active 49
- execution model 42
- modification of 49
- repetitive 40
- termination of 50

lowercase symbols 11

M

MAX function

- description 94
- example 94
- implementation maximum 94

MIN function

- description 94
- example 94
- implementation maximum 94

minutes calculated from midnight 100

mixed DBCS string 86

model of input and output 143

Month option of DATE function 87

more REXX information 4

multi-way call 36, 69

multiple

- assignments in parsing 116
- string parsing 121

multiplication

- description 130
- operator 16

N

names

- of functions 76
- of programs 57
- of subroutines 35
- of variables 12

National Language Character Set (NLCS) Support

- character length 7
- with labels 20

negation

- of logical values 18
- of numbers 16

nesting of control structures 38

nibbles 10

NLCS (National Language Character Set) Support

- See National Language Character Set (NLCS) Support

NOETMODE 54

NOEXMODE 54

NOP instruction

- description 51
- example 51

Normal option of DATE function 87

not equal operator 17

not greater than operator 17

not less than operator 17

NOT operator 13, 18

notation

- engineering 134
- exponential, example 134
- scientific 134

note

- condition traps 140

NOVALUE condition

- not raised by VALUE function 103
- of SIGNAL instruction 141
- on SIGNAL instruction 138
- use of 175

null

- clauses 20
- strings 9, 15

null instruction

- See NOP instruction

numbers

- arithmetic on 16, 127, 129
- checking with DATATYPE 86
- comparison of 17, 133
- description 12, 127, 128
- formatting for display 91
- implementation maximum 13
- in DO instruction 39
- truncating 103
- use in the language 135
- whole 135

numeric

- comparisons, example 133
- options in TRACE 72

NUMERIC instruction

- description 52
- DIGITS option 52
- FORM option 52, 135
- FUZZ option 52
- settings saved during subroutine calls 37

numeric patterns

- See positional patterns

O

operations

- arithmetic 129
- tracing results 70

operator

- arithmetic
 - description 15, 127, 129
 - list 16
- as special characters 13
- characters
 - description 13
 - implementation maximum 15, 23
- comparison 17, 133
- concatenation 16
- examples 131, 132
- logical 18
- precedence (priorities) of 18

options

- alphabetic character word in TRACE 71
- numeric in TRACE 72

OPTIONS instruction

- description 54

OR, logical

- exclusive 18
- inclusive 18

Ordered option of DATE function 87

ORing character strings together 81

OTHERWISE clause

- See SELECT instruction

output to the user 143

overflow, arithmetic 136

OVERLAY function

- description 95
- example 95

overlying a string onto another 95

overview of parsing 122

P

- (NOT operator) 18

-< (not less than operator) 17

-<< (strictly not less than operator) 18

-= (not equal operator) 17

-== (strictly not equal operator) 17

-> (not greater than operator) 17

->> (strictly not greater than operator) 18

packing a string with X2C 107

pad character, definition 78

page, code 8

parentheses

- adjacent to blanks 13
- in expressions 18
- in function calls 75
- in parsing templates 118

PARSE instruction

description 56

PARSE LINEIN

role in input and output 143

PARSE PULL

role in input and output 143

parsing

advanced topics 121

combining patterns and parsing into words 116

combining string and positional patterns 121

conceptual overview 122

definition 111

description 111—125

equal sign 115

examples

combining positional pattern and parsing into words 117

combining string and positional patterns 121

combining string pattern and parsing into words 116

parsing instructions 119

parsing multiple strings in a subroutine 121

period as a placeholder 113

simple templates 111

templates containing positional patterns 114

templates containing string patterns 113

using a variable as a positional pattern 118

using a variable as a string pattern 118

into words 111

multiple assignments 116

multiple strings 121

patterns

conceptual view 124

positional 111, 114

string 111, 113

period as placeholder 113

positional patterns 111

absolute 114

relative 115

variable 118

selecting words 111

source string 111

special case 121

steps 122

string patterns 111

literal string patterns 113

variable string patterns 118

summary of instructions 119

templates

in ARG instruction 33

in PARSE instruction 56

in PULL instruction 62

treatment of blanks 112

UPPER, use of 118

variable patterns

positional 118

string 118

parsing (continued)

with DBCS characters 122

word parsing

conceptual view 125

description and examples 111

patterns in parsing

combined with parsing into words 116

conceptual view 124

positional 111, 114

string 111, 113

period

as placeholder in parsing 113

causing substitution in variable names 22

in numbers 128

permanent command destination change 30**POS function**

description 95

example 95

position

last occurrence of a string 93

positional patterns

absolute 114

description 111

relative 115

variable 118

powers of ten in numbers 12**precedence of operators 18****precision of arithmetic 128****preface xi****prefix**

operators 16, 18

presumed command destinations 30**PROCEDURE instruction**

description 59

example 60

Procedures Language REXX/400

See REstructured eXtended eXecutor/400 (REXX/400)

programming

interface, common 2

restrictions 7

programs

arguments to 33

retrieving lines with SOURCELINE 97

protecting variables 59**pseudo random number function of RANDOM 96****publications**

REXX 4

REXX/400 Programmer's Guide 4

SAA Common Programming Interface REXX Level 2

Reference 4

PULL instruction

description 62

example 62

role in input and output 143

PULL option of PARSE instruction 57

purpose

of this book xi
SAA 1

PUSH instruction

description 63
example 63
implementation maximum 63
role in input and output 143

Q

QREXQ interface 144, 163, 170, 171

QREXVAR interface 166

SHVBLOCK map 167
symbolic vs. direct 167

QREXX, parameter structure 151

querying TRACE setting 101

queue

See external, data queue

QUEUE instruction

description 64
example 64
implementation maximum 64
role in input and output 143

QUEUED function

description 95
example 95
role in input and output 143

queuing API 170

R

RANDOM function

description 96
example 96

random number function of RANDOM 96

RC (return code)

from CL commands 153
from other commands 155
maximum length 155
not set during interactive debug 173
set by commands 26
special variable 141, 176

recursive call 36

relative positional patterns 115

remainder

description 127, 132
operator 16

reordering data with TRANSLATE function 102

repeating a string with COPIES 85

repetitive loops

altering flow 50
controlled repetitive loops 40
exiting 50
simple DO group 40

repetitive loops (*continued*)

simple repetitive loops 40

reservation of keywords 175

restoring variables 44

restrictions

embedded blanks in numbers 12
first character of variable name 21
in programming 7
maximum length of results 15

REstructured eXtended eXecutor (REXX) Language

See REXX

REstructured eXtended eXecutor/400 (REXX/400)

description 1

RESULT

set by RETURN instruction 36, 65
special variable 176

results

length of 15

retrieving

argument strings with ARG 33
arguments with ARG function 80
lines with SOURCELINE 97

return

code
as set by commands 26
setting on exit 45
string
setting on exit 45

return codes and values 151

RETURN instruction

description 65

returning control from REXX program 65

REVERSE function

description 96
example 96

REXX

description 1
information, more 4
publications 4

REXX/400

See REstructured eXtended eXecutor/400 (REXX/400)

REXX/400 Programmer's Guide 4

RIGHT function

description 97
example 97

RMVREXBUF command 145

rounding

description 129
using a character string as a number 12

routines

See functions
See subroutines

running off the end of a program 45

S

SAA

- books 4
- purpose 1
- solution 1
- supported environments 2

SAY instruction

- description 66
- displaying data 66
- example 66
- role in input and output 143

SBCS strings 177

scientific notation 134

search order

- for functions 77
- for subroutines 36

seconds calculated from midnight 101

security characteristics for REXX/400 150

SELECT instruction

- description 67
- example 67

selecting a default with ABBREV function 79

semicolons

- implied 14
- omission of 29
- within a clause 8

sequence, collating using XRANGE 106

serial input and output

- See character input and output

SETMSGRC function 108

shift-in (SI) characters 178, 182

shift-out (SO) characters 178, 182

SHORT_VARSTRING 154

- return codes from 155

SIGL

- set by CALL instruction 36
- set by SIGNAL instruction 68
- special variable 141, 176
- example 142

SIGN function

- description 97
- example 97

SIGNAL instruction

- description 68
- example 68
- execution of in subroutines 37

significant digits in arithmetic 128

simple

- repetitive loops 40
- symbols 22

single stepping

- See interactive debug

solution, SAA 1

source

- of program and retrieval of information 57

source (continued)

- string 111

source code

- entering 147

SOURCE option of PARSE instruction 57

SOURCELINE function

- description 97
- example 97

SPACE function

- description 98
- example 98

spacing, formatting, SPACE function 98

special

- characters and example 13
- parsing case 121
- RC 176
- RESULT 157, 176
- SIGL 176
- variables
 - RC 26, 141
 - RESULT 36, 65
 - SIGL 36, 141

Standard option of DATE function 88

standard streams 143

- default 143
- STDERR 143
- STDIN 143
- STDOUT 143

stem of a variable

- assignment to 24
- description 22
- used in DROP instruction 44
- used in PROCEDURE instruction 59

stepping through programs

- See interactive debug

steps in parsing 122

stream

- See character input and output

strict comparison 17

strictly equal operator 17

strictly greater than operator 17

strictly greater than or equal operator 17

strictly less than operator 17

strictly less than or equal operator 18

strictly not equal operator 17

strictly not greater than operator 18

strictly not less than operator 18

string

- and symbols in DBCS 178
- as literal constant 9
- as name of function 9
- as name of subroutine 35
- binary specification of 10
- centering using CENTER function 83
- centering using CENTRE function 83
- comparison of 17

string *(continued)*

- concatenation of 16
- copying using COPIES 85
- DBCS 177
- DBCS-only 179
- deleting part, DELSTR function 88
- description 9
- extracting words with SUBWORD 99
- hexadecimal specification of 10
- interpretation of 47
- length of 15
- mixed SBCS/DBCS 179
- mixed, validation 179
- null 9, 15
- patterns
 - description 111
 - literal 113
 - variable 118
- quotation marks in 9
- repeating using COPIES 85
- SBCS 177
- verifying contents of 104

STRIP function

- description 98
- example 98

STREXPRC command 26, 38, 45, 57, 65, 148, 157

structure and syntax 8

Structured Query Language command environment (EXECSQL) 154

subexpression 15

subkeyword 21

subroutines

- calling of 35
- definition 75
- forcing built-in or external reference 36
- naming of 35
- passing back values from 65
- return from 65
- use of labels 35
- variables in 59

subsidiary list 44, 59

substitution

- in expressions 15
- in variable names 22

SUBSTR function

- description 99
- example 99

substring, extracting with SUBSTR function 99

subtraction

- description 130
- operator 16

SUBWORD function

- description 99
- example 99

summary

- parsing instructions 119

symbol

- assigning values to 21
- classifying 21
- compound 22
- constant 22
- DBCS validation 179
- DBCS-only 178
- description 11
- implementation maximum 12
- mixed DBCS 178
- simple 22
- uppercase translation 11
- use of 21
- valid names 12

SYMBOL function

- description 99
- example 100

symbols and strings in DBCS 178

syntax

- diagrams 3
- error
 - traceback after 73
 - trapping with SIGNAL instruction 137
- general 8

syntax checking

- See TRACE instruction

SYNTAX condition of SIGNAL instruction 138, 141

system exit interfaces

- entry conditions 159
- exit definitions 160
- functions and subfunctions 158
- return codes 160
- variable pool 158

system exits

- external function exit 160
- external HALT exit 164
- external trace exit 165
- host command exit 162
- initialization exit 165
- queue exit 162
- RXCMD exit 162
 - RXCMDHST exit 162
- RXFNC exit 160
 - RXFNCAL exit 160
- RXHLT exit 164
 - RXHLTCLR exit 164
 - RXHLTTST exit 164
- RXINI exit 165
 - RXINIEXT exit 165
- RXMSQ exit 162
 - RXMSQPLL exit 162
 - RXMSQPSH exit 163
 - RXMSQSIZ exit 163
- RXSIO exit 163
 - RXSIODTR exit 164
 - RXSIOSAY exit 163
 - RXSIOTRC exit 163

system exits (*continued*)
RXSIO exit (*continued*)
RXSIOTRD exit 164
RXTER exit 166
RXTEREXT exit 166
RXTRC exit 165
RXTRCTST exit 165
session I/O exit 163
termination exit 166

T

tail 22

template

definition 111
in PULL instruction 62
list
 ARG instruction 33
 PARSE instruction 56
parsing
 definition 111
 general description 111
 in ARG instruction 33
 in PARSE instruction 56
 PULL instruction 62

temporary command destination change 30

ten, powers of 134

terminal input and output

display of 149

terminals

reading from with PULL 62
writing to with SAY 66

terms and data 15

testing

abbreviations with ABBREV function 79
variable initialization 99

text formatting

See formatting
See word

THEN

as free standing clause 29
following IF clause 46
following WHEN clause 67

TIME function

description 100
example 101

tips, tracing 72

TO phrase of DO instruction 39

trace

tags 73

TRACE function

description 101
example 102

TRACE instruction

See *also* interactive debug
alphabetic character word options 71

TRACE instruction (*continued*)

description 70
example 72

TRACE setting

altering with TRACE function 101
altering with TRACE instruction 70
querying 101

traceback, on syntax error 73

tracing

action saved during subroutine calls 37
by interactive debug 173
data identifiers 73
execution of programs 70
tips 72

tracing flags

. 73
+++ 73
>.> 73
>>> 73
>C> 73
>F> 73
>L> 73
>O> 73
>P> 73
>V> 73

trailing

blank removed using STRIP function 98
zeros 129

TRANSLATE function

description 102
example 102

translation

See *also* uppercase translation
with TRANSLATE function 102

trap conditions

explanation 137
how to trap 137
information about trapped condition 84
using CONDITION function 84

trapname

description 138

TRUNC function

description 103
example 103

truncating numbers 103

type of data checking with DATATYPE 86

typing data

See SAY instruction

U

unassigning variables 44

unconditionally leaving your program 45

underflow, arithmetic 136

uninitialized variable 21

- unpacking a string**
 - with B2X 82
 - with C2X 86
- UNTIL phrase of DO instruction 39**
- unusual change in flow of control 137**
- UPPER**
 - in parsing 118
 - option of PARSE instruction 56
- uppercase translation**
 - during ARG instruction 33
 - during PULL instruction 62
 - of symbols 11
 - with PARSE UPPER 56
 - with TRANSLATE function 102
- Use option of DATE function 88**
- user input and output 143**
- user-defined command environment 154**
 - return codes from 153
- utterances 143**

V

- validation**
 - DBCS symbol 179
 - mixed string 179
- VALUE function**
 - description 103
 - example 103
- value of variable, getting with VALUE 103**
- VALUE option of PARSE instruction 57**
- VAR option of PARSE instruction 57**
- variable**
 - compound 22
 - compound pseudo-CL in REXX programs 150
 - controlling loops 40
 - description 21
 - dropping of 44
 - exposing to caller 59
 - external collections 103
 - getting value with VALUE 103
 - global 103
 - in internal functions 59
 - in subroutines 59
 - names 12
 - new level of 59
 - parsing of 57
 - patterns, parsing with
 - positional 118
 - string 118
 - pool interface 21
 - positional patterns 118
 - reference 118
 - resetting of 44
 - setting new value 21
 - SIGL 142
 - simple 22

- variable (continued)**
 - special
 - RC 26, 141, 176
 - RESULT 65, 176
 - SIGL 36, 141, 176
 - string patterns, parsing with 118
 - testing for initialization 99
 - valid names 21
- variable pool function code**
 - SHVDROPV 168
 - SHVEXIT 170
 - SHVEXTFN 170
 - SHVFETCH 168
 - SHVNEXTV 168
 - SHVPRIV 169
 - SHVSET 168
 - SHVSYDRO 168
 - SHVSYFET 168
 - SHVSYSET 168
- variable pool interface**
 - functions
 - QREXVAR Service 166
 - Shared-Variable Request Block 167
 - SHVEXIT 157
- VERIFY function**
 - description 104
 - example 104
- verifying contents of a string 104**
- VERSION option of PARSE instruction 57**

W

- Weekday option of DATE function 88**
- WHEN clause**
 - See SELECT instruction
- where to find more information xi**
- WHILE phrase of DO instruction 39**
- who should read this book xi**
- whole numbers**
 - checking with DATATYPE 86
 - description 13, 135
- word**
 - alphabetic character options in TRACE 71
 - counting in a string 106
 - deleting from a string 89
 - extracting from a string 99, 104
 - finding length of 105
 - in parsing 111
 - locating in a string 105
 - parsing
 - conceptual view 125
 - description and examples 111
- WORD function**
 - description 104
 - example 105

word processing

See formatting

WORDINDEX function

description 105

example 105

WORDLENGTH function

description 105

example 105

WORDPOS function

description 105

example 105

WORDS function

description 106

example 106

writing

to the stack

with PUSH 63

with QUEUE 64

X**X2B function**

description 106

example 106

X2C function

description 107

example 107

X2D function

description 107

example 107

implementation maximum 108

XOR, logical 18**XORing character strings together 82****XRANGE function**

description 106

example 106

Z**zeros**

added on the left 97

removal with STRIP function 98

Reader Comments—We'd Like to Hear from You!

AS/400 Advanced Series
REXX/400 Reference
Version 4

Publication No. SC41-5729-00

Overall, how would you rate this manual?

	Very Satisfied	Satisfied	Dissatisfied	Very Dissatisfied
Overall satisfaction				

How satisfied are you that the information in this manual is:

Accurate				
Complete				
Easy to find				
Easy to understand				
Well organized				
Applicable to your tasks				
THANK YOU!				

Please tell us how we can improve this manual:

May we contact you to discuss your responses? Yes No

Phone: (____) _____ Fax: (____) _____ Internet: _____

To return this form:

- Mail it
- Fax it
United States and Canada: **800+937-3430**
Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

Name

Address

Company or Organization

Phone No.



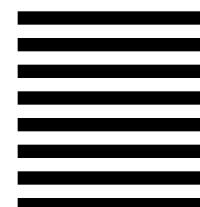
Fold and Tape

Please do not staple

Fold and Tape



NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST-CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986



Fold and Tape

Please do not staple

Fold and Tape



Printed in the United States of America
on recycled paper containing 10%
recovered post-consumer fiber.

SC41-5729-00





AS/400 Advanced Series

REXX/400 Reference

Version 4