

HP Configuration Management

Application Manager and Configuration Server

Software Version: 5.10

REXX Programming Guide

Manufacturing Part Number: T3424-90126

Document Release Date: October 2007

Software Release Date: October 2007



i n v e n t

Legal Notices

Warranty

The only warranties for HP products and services are set forth in the express warranty statements accompanying such products and services. Nothing herein should be construed as constituting an additional warranty. HP shall not be liable for technical or editorial errors or omissions contained herein.

The information contained herein is subject to change without notice.

Restricted Rights Legend

Confidential computer software. Valid license from HP required for possession, use or copying. Consistent with FAR 12.211 and 12.212, Commercial Computer Software, Computer Software Documentation, and Technical Data for Commercial Items are licensed to the U.S. Government under vendor's standard commercial license.

Copyright Notices

© Copyright 1998-2007 Hewlett-Packard Development Company, L.P.

Trademark Notices

Linux is a registered trademark of Linus Torvalds.

Microsoft®, Windows®, and Windows® XP are U.S. registered trademarks of Microsoft Corporation.

OpenLDAP is a registered trademark of the OpenLDAP Foundation.

PREBOOT EXECUTION ENVIRONMENT (PXE) SERVER

Copyright © 1996-1999 Intel Corporation.

TFTP SERVER

Copyright © 1983, 1993

The Regents of the University of California.

OpenLDAP

Copyright 1999-2001 The OpenLDAP Foundation, Redwood City, California, USA.

Portions Copyright © 1992-1996 Regents of the University of Michigan.

OpenSSL License

Copyright © 1998-2001 The OpenSSLProject.

Original SSLeay License
Copyright © 1995-1998 Eric Young (eay@cryptsoft.com)

DHTML Calendar
Copyright Mihai Bazon, 2002, 2003

Documentation Updates

This guide's title page contains the following identifying information:

- Software Version number, which indicates the software version
- Document release date, which changes each time the document is updated
- Software release date, which indicates the release date of this version of the software

To check for recent updates or to verify that you are using the most recent edition, visit the following URL:

ovweb.external.hp.com/lpe/doc_serv/

You will also receive updated or new editions if you subscribe to the appropriate product support service. Contact your HP sales representative for details.

Table 1 lists new features added for the Configuration Management v 5.10 release.

Table 1 New Feature

Chapter	Version	Changes
6	5.10	Page 254, REXX variables and CM object values : Example 2, fixed typo: <code>number_variables = value(object 'vars') - 1</code>

Table 2 indicates changes made to this document for earlier releases.

Table 2 Document changes

Chapter	Version	Changes
Global	5.00	For version 5.00, Radia was rebranded to Configuration Management (CM). This may not include the names of individual modules, variables, or functions included in this guide.
Global	3.1	Beginning with version 3.1 of the Radia Clients, the name of the REXX Interpreter changed from EDMPNLWR.EXE to RADPNLWR.EXE. All references to EDMPNLWR throughout the guide have changed accordingly

Chapter	Version	Changes
3	3.1	Page 38, The CM REXX Executable: new section.
4	3.1	Page 48, ADDRESS: edited environment parameter – removed EDMWIN, Mac, and DOS references; edited note. Added WITH <i>redirect</i> parameter to table. Added 2 new examples: 5 and 6.
5	3.1	Page 155, DATE: syntax modified.
5	3.1	Page 155, DATE Parameters table: option parameter changed to out-option; two new parameters: date_string and in_option.
5	3.1	Page 198, POPEN: new section.
5	3.1	Page 222, TIME: syntax modified.
5	3.1	Page 222, TIME Parameters table: option parameter changed to out_option; two new parameters added: time_string and in_option.
6	3.1	Page 283, NVDOBJECTS new extension.
6	3.1	Page 285, NVDPATHS new extension.
6	3.1	Page 268, EDMGETV new extension.
6	3.1	Page 287 and 288, RADGET and RADSET: Radia 3.1 Client support adds two new extensions. RADGET and RADSET expand the functions of EDMGET and EDMSET, respectively, by permitting reading and writing of objects from different directories.
6	3.1	Page 291, RXXCommandKill: new extension.
6	3.1	Page 292, RXXCommandSpawn: new extension.
6	3.1	Page 293, RXXCommandWait: new extension.
6	3.1	Page 294, RXXOSEndOfLineString: new extension.
6	3.1	Page 295, RXXOSEnvironmentSeparator: new extension.

Chapter	Version	Changes
6	3.1	Page 296, <i>RXXOSName</i> : new extension.
6	3.1	Page 297, <i>RXXOSPathSeparator</i> : new extension.
6	3.1	Page 298, <i>RXXSleep</i> : new extension.
6	3.1	Page 299, <i>WinMessageBox</i> : new extension.
6	3.1	Page 302, <i>WinGetVersion</i> : new extension.

Support

You can visit the HP Software support web site at:

www.hp.com/go/hpsoftwaresupport

This Web site provides contact information and details about the products, services, and support that HP Software offers.

HP Software online software support provides customer self-solve capabilities. It provides a fast and efficient way to access interactive technical support tools needed to manage your business. As a valued support customer, you can benefit by using the support site to:

- Search for knowledge documents of interest
- Submit and track support cases and enhancement requests
- Download software patches
- Manage support contracts
- Look up HP support contacts
- Review information about available services
- Enter into discussions with other software customers
- Research and register for software training

Most of the support areas require that you register as an HP Passport user and sign in. Many also require a support contract.

To find more information about access levels, go to:

http://h20230.www2.hp.com/new_access_levels.jsp

To register for an HP Passport ID, go to:

<http://h20229.www2.hp.com/passport-registration.html>

Syntax Notes

When an instruction keyword can have more than one value, the options are stacked within brackets as in the TRACE instruction:

```
TRACE [option]
      [[VALUE] expression]
```

Type only one of the choices. For example:

```
TRACE E
```

When a required operand can have more than one value, the options are stacked in the same manner as for optional operands. As with optional operands, you type only one of the choices. The syntax diagram for the NUMERIC instruction illustrates a combination of required and optional operands that can have more than one value:

```
NUMERIC      DIGITS [expr1]
              FORM  [SCIENTIFIC]
                  [ENGINEERING]
                  [[VALUE] expr2]
              FUZZ  [expr3]
```

These characters must be typed exactly as shown in the syntax diagrams.

- Literal strings are delimited by either single or double quotes.
- Hexadecimal strings are delimited by single or double quotes followed immediately by the character `x`.

Binary strings are delimited by single or double quotes followed immediately by the character `b`.

The table below describes terms that may be used interchangeably throughout this book, as well as in other HP publications.

Contents

1	Introduction	19
	Introduction to CM REXX.....	20
	About This Book.....	20
	Chapter 2: Language.....	21
	Chapter 3: Operations.....	21
	Chapter 4: Instructions.....	21
	Chapter 5: Built-In Functions	21
	Chapter 6: Using Extensions.....	21
	Chapter 7: Registry Manipulation Functions	22
	Appendix A: Message Summary	22
	Appendix B: Programming Hints.....	22
	Appendix C: System Limitations	22
	Bibliography	22
2	Language	23
	What is a Clause?	24
	Clause Syntax Notes	26
	What is a Symbol?	27
	What is an Expression?	29
	Comparative Operators	31
	Normal Comparative Operators	31
	Strict Comparative Operators	31
	What is a Function?.....	32
	What are Special Variables?.....	33
	What are Condition Traps?.....	34

What is an Input/Output Operation?	35
What is Parsing?	36
3 Operations	37
The CM REXX Executable	38
The RADPNLWR Executable	38
Invoking RADPNLWR	38
RADPNLWR Log Files	40
Executing a REXX Method from Windows	41
Coding CM REXX Programs	42
Including External Functions and Subroutines	42
Executing Host Commands	42
4 Instructions	45
Overview of REXX Instructions	46
Quick Reference	46
ADDRESS	48
ARG	53
CALL	55
DO 60	
DROP	65
EXIT	67
IF 69	
INTERPRET	72
ITERATE	73
LEAVE	74

NOP	75
NUMERIC	76
PARSE	79
Parsing Templates	81
Parsing by Words	82
Parsing by Patterns	83
Parsing by Position	84
Parsing with Placeholders	87
Putting it All Together	88
PROCEDURE	89
PULL	93
PUSH	95
QUEUE	97
RETURN	99
SAY	102
SELECT	104
SIGNAL	106
TRACE	110
UPPER	117

5 Built-In Functions 119

Built-In Functions Overview	120
General Rules for Built-In Functions	122
ABBREV	124
ABS	126
ADDRESS	127
ARG	128

BITAND.....	130
BITOR	131
BITXOR.....	132
B2X	133
CENTER.....	134
CHARIN	136
CHAROUT	138
CHARS	140
CHDIR.....	141
COMPARE	142
CONDITION	144
COPIES	146
CUSERID	148
C2D	149
C2X	150
DATATYPE	152
DATE.....	155
DELSTR	159
DELWORD.....	161
DIGITS	163
D2C	164
D2X.....	165
ERRORTXT	166
FIND.....	168

FORM	170
FORMAT	171
FUZZ.....	175
GETCWD.....	176
GETENV	177
INDEX.....	178
INSERT	180
JUSTIFY	181
LASTPOS	182
LEFT.....	184
LENGTH	186
LINEIN.....	187
LINEOUT.....	189
LINES.....	191
LOWER	193
MAX.....	194
MIN.....	195
OVERLAY	196
POPEN	198
POS.....	200
PUTENV	202
QUEUED.....	203
RANDOM	204
REVERSE	206

RIGHT	207
SIGN	209
SOURCELINE	210
SPACE	211
STREAM	213
STRIP	215
SUBSTR	217
SUBWORD	219
SYMBOL	220
TIME	222
TRACE	225
TRANSLATE	226
TRUNC	228
UPPER	229
USERID	230
VALUE	231
VERIFY	233
WORD	236
WORDINDEX	237
WORDLENGTH	238
WORDPOS	239
WORDS	241
XRANGE	242
X2B	244

X2C	245
X2D	247

6 Using Extensions 249

CM Agent REXX Methods.....	250
Overview of CM REXX Extensions.....	250
REXX, CM, Objects and Object Paths/Folders.....	250
Using Extensions	251
Function Calls and Return Values.....	252
Identifying Variables	253
REXX variables and CM object values	254
The CM REXX Extension List	258
EDMADD	259
EDMATTR.....	261
EDMBLD.....	263
EDMCMD.....	264
EDMDELHEAP	265
EDMDELVAR.....	266
EDMFREE	267
EDMGET.....	268
EDMGETV	270
EDMLOC.....	271
EDMRST	272
EDMSET	273
EDMSORT	275
GET_CHILD_OBJ	276

LOAD_CHILDREN	277
NOWAIT.....	282
NVDOBJECTS.....	283
NVDPATHS	285
NvdVerQueryValueStringFileInfo	286
RADGET.....	287
RADSET	290
RXXCommandKill	291
RXXCommandSpawn	292
RXXCommandWait	293
RXXOSEndOfLineString	294
RXXOSEnvironmentSeparator.....	295
RXXOSName.....	296
RXXOSPathSeparator	297
RXXSleep.....	298
WinMessageBox.....	299
WinExpandEnvironmentStrings	301
WinGetVersion.....	302

7 Registry Manipulation Functions 305

Registry Manipulation Functions.....	306
WinRegCloseKey.....	308
WinRegCreateKey	309
WinRegDeleteKey.....	311

WinRegDeleteValue.....	312
WinRegEnumKey	313
WinRegEnumValue	315
WinRegOpenKey.....	317
WinRegQueryInfoKey	319
WinRegQueryValue.....	321
WinRegSetValue	323
A Message Summary	325
CM REXX Messages.....	325
B Programming Hints	335
Invoking a Built-in Function Like an Instruction.....	335
Failure to Use Commas with CALL and PARSE ARG	335
With CALL.....	335
With PARSE ARG	336
Incorrect Use of Continuation.....	337
Incorrect CALL Syntax	337
Failure to Enclose Command Arguments Within Quotes.....	338
Failure to Close a File	338
C System Limitations.....	339
Implementation-Specific Limits	339

Bibliography	341
Index	343

1 Introduction

This chapter introduces you to CM REXX by first comparing this language to shell programs and programming languages. It then provides an overview of the *REXX Programming Guide* and an explanation of its conventions.

Introduction to CM REXX

CM REXX is an implementation of the REXX programming language as described in *The REXX Language: A Practical Approach to Programming* by M.F. Cowlshaw (1990: Prentice Hall).

CM REXX is an interpreted language that provides a simple way to customize various aspects of CM processing.

REXX programs are easy to write, understand, and modify. User-friendly standard features and a simple syntax enable rapid development and testing. These features include:

- Natural data-typing (nothing to declare)
- Dynamic scoping
- Built-in trace facilities

CM REXX methods (programs) are portable across multiple platforms.

CM REXX conforms to the ANSI standard X3.274:1996, "Programming Language REXX."

About This Book

The *CM REXX Programming Guide* is a reference manual for the CM REXX programming language, and a guide for creating CM REXX methods. CM REXX methods are the procedures you write to customize processing for your CM-managed computing environment.

This guide describes the features, operation, and syntax of CM REXX, as well as the built-in functions that can be called by a program.

This section provides you with an overview of this programming guide, so you can quickly turn to the information you need to start writing CM REXX procedures for your installation. The following chapter summaries should help you find the information you need quickly and easily.

Chapter 2: Language

This chapter summarizes the language structure for those not already familiar with it. The basic elements, terminology, and concepts of CM REXX are presented in a concise format for review and reference.

Chapter 3: Operations

This chapter presents details on the execution of CM REXX programs. It also covers such implementation-specific topics as access to external functions, subroutines, and host command execution.

Chapter 4: Instructions

This chapter explains selected instructions that are provided in the CM REXX programming language. CM REXX instructions consist of one or more clauses that are identified by keywords, and are recognized only after meeting specific conditions.

Chapter 5: Built-In Functions

This chapter explores the powerful set of built-in functions found in CM REXX. These functions are part of the language and are always available to be called by any program.

Chapter 6: Using Extensions

This chapter teaches you how to use the REXX function extensions of CM when you customize CM processing for your CM environment.

Chapter 7: Registry Manipulation Functions

This chapter describes CM REXX functions that enable you to inspect and manipulate the Windows Registry.

Appendix A: Message Summary

This appendix lists and describes all the messages that may be generated by CM REXX. This is a valuable resource for interpreting any error messages you encounter while compiling and executing CM REXX programs.

Appendix B: Programming Hints

This appendix identifies the common programming mistakes to avoid when writing CM REXX programs.

Appendix C: System Limitations

This appendix documents six implementation-specific limitations of CM REXX.

Bibliography

This appendix lists some additional reference books on the REXX language.

2 Language

CM REXX is implemented according to the language definition contained in *The REXX Language: A Practical Approach to Programming*, by M. F. Cowlshaw (1990: Prentice Hall). The elements of the language are described in detail by Cowlshaw. This chapter summarizes the language structure.

What is a Clause?

The basic element of the CM REXX language is the clause. A clause is composed of one or more tokens preceded or followed by zero or more blanks and optionally terminated by a semicolon. Tokens in a clause can be any of the following:

- Literal string
- Hexadecimal string
- Binary string
- Symbol
- Operator
- Special character

The following table lists the tokens and their meanings.

Table 2.1 ~ Tokens and their Meanings

Token	Explanation
literal string	<p>A literal string is a sequence that can include any character. It is enclosed in single or double quotes. A literal string that includes no characters is known as a null string. Examples include:</p> <pre>'Hello world!' "What's in a name?" ' /* Null string */</pre>
hexadecimal string	<p>A hexadecimal string is a series of hexadecimal digits grouped in pairs, enclosed in quotes, and followed immediately by the character 'x' (upper- or lowercase). The pairs of hexadecimal digits can be optionally separated by one or more blanks. Examples include:</p> <pre>'c1c3'x "abcdef"X '61 62 63'x ""x</pre>

Table 2.1 ~ Tokens and their Meanings

Token	Explanation
binary string	<p>A binary string is a series of binary digits grouped in fours, enclosed in quotes, and followed immediately by the character 'b' (upper- or lowercase). The groups of binary digits can be optionally separated by one or more blanks. Examples include:</p> <pre>'0001'b '10011001'B "1111 0000"b 'b</pre>
symbol	<p>A symbol is any group of alphanumeric characters. Symbols can also include the characters ".", " ", "?", and "_". If a symbol begins with a digit, it can also include the letter "e" (upper- or lowercase) followed optionally by a plus or minus sign ("+" or "-") and one or more digits. A symbol can be a constant, a keyword, or a variable, depending upon the context in which it is used. Additional details are provided in the section entitled <i>What is a Symbol</i> on page 27. Examples include:</p> <pre>abc data.1 new_data 17 31416E-4</pre>
operator	<p>An operator is a character used to indicate operations in expressions. For a complete list of operators supported in CM REXX, see <i>What is an Expression</i> on page 29. Examples include:</p> <pre>+ - > >> =</pre>
special characters	<p>Special characters include both the operator characters and the characters ".", ";", ":", "(", and ")". Special characters function as token delimiters.</p>

A CM REXX clause can be any of the following types:

- Instruction
- Label
- Null clause

The following table lists the types of clauses and their meanings.

Table 2.2 ~ Clauses and their Meanings

Clause	Explanation
instruction	An instruction describes an action to be performed by the interpreter. Instructions can be any of the following: assignment An instruction of the form <i>symbol = expression</i> , that assigns a value to a variable. keyword An instruction that begins with a keyword that identifies the operation to be performed; examples of instruction keywords include PARSE, DO, CALL, and RETURN. command An instruction comprised simply of an expression, which is evaluated and passed to an external environment for processing.
label	A label is a clause composed of a single symbol followed by a colon. Labels identify the target of CALL or SIGNAL instructions or the beginning of an internal function.
null clause	A null clause is any clause comprised only of blanks or comments.

Clause Syntax Notes

A comment is any sequence of characters preceded by a forward slash and an asterisk (/*) and followed by an asterisk and a backward slash (*/). Comments can appear anywhere in the program and can be nested.

A clause in a CM REXX program can span more than one line. Continuation is indicated by a comma. The comma is replaced by a blank when the lines are concatenated during program execution.

For example, the program fragment:

```
list_of_months = Jan Feb Mar Apr May Jun Jul,  
                Aug Sep Oct Nov Dec  
say list_of_months
```

produces the following output:

```
JAN FEB MAR APR MAY JUN JUL AUG SEP OCT NOV DEC
```

What is a Symbol?

A symbol in CM REXX is any group of characters A-Z, a-z, 0-9, ".", "|", "?", or "_". The meaning of a symbol is derived from its context. The following table lists the types of symbols and their meanings.

Table 2.3 ~ Symbols and their Meanings

Symbol	Explanation
compound symbol	<p>A compound symbol does not begin with a digit, contains at least one embedded period, and cannot end with a period. The name begins with a stem (see description of stem symbols) followed by a period, followed by a tail. The tail can be a constant symbol, a simple symbol, or null. Before a compound symbol is used, the values of any simple symbols in the tail are substituted, creating the derived name of the compound symbol. The default value of a compound symbol is one of the following:</p> <ul style="list-style-type: none">The value assigned to the stem.The symbol name translated to uppercase if no value has been assigned to the stem. <p>Examples include:</p> <pre>data.1 = 5 say data.1</pre> <p>Output is 5; the tail is a constant symbol so this compound symbol does not have a derived name.</p> <pre>x = 3 data.3 = 7 say data.x</pre> <p>Output is 7; the value of the simple symbol <i>x</i> is substituted to produce the derived name <code>data.3</code>, which has been assigned the value 7.</p>

Table 2.3 ~ Symbols and their Meanings

Symbol	Explanation
constant symbol	<p>A constant symbol begins with a digit and can include the letter <i>e</i> followed optionally by a plus or minus sign and one or more digits. The value of a constant symbol cannot be changed. Examples include:</p> <pre> 10 3.1416 15e-3 </pre>
simple symbol	<p>A simple symbol does not begin with a numeric digit and does not contain any embedded periods. Its default value (i.e., when no value is assigned to the symbol) is the symbol itself, translated to uppercase. A simple symbol can be used as a variable and can be assigned a value. Examples include:</p> <pre> list file1 Date </pre>
stem	<p>A stem does not begin with a numeric digit and contains only one period, which must be the last character. It can be assigned a value, which effectively assigns that value to all compound symbols that begin with this stem. Stems can represent a collection (or array) of variables. Examples of stems and their use include:</p> <pre> list. /* a stem whose value is "LIST." */ list. = animals list.3 = 'cows' say list.1 list.new list.3 </pre> <p>The output is:</p> <pre> ANIMALS ANIMALS cows </pre> <p>since only the compound symbol list.3 has been assigned a value different from the value assigned to the stem. The value assigned to the stem is uppercase, because the value of the variable ANIMALS is undefined, and the default value of an undefined variable is the variable's name in uppercase.</p>

What is an Expression?

A CM REXX clause can contain one or more expressions. An expression consists of one or more terms and zero or more operators. The operators specify the operations to be performed on the terms.

The terms in an expression can be any of the following:

- Function call
- Literal string
- Operator
- Sub-expression
- Symbol

The following table lists the types of expressions and their meanings.

Table 2.4 ~ Expressions and their Meanings

Expression	Explanation
function calls	Function calls are of the form: <pre>function_name([expression] [, [expression]] ...)</pre> where <code>function_name</code> can be a symbol or a string.
literal strings	Literal strings are treated as constants.
operators	Operators can be grouped into four categories: Arithmetic Comparative Concatenation Logical
sub-expression	A sub-expression is any expression enclosed in parentheses.
symbols	Symbols are translated to uppercase and can be treated as constants or variables. Symbols that do not begin with a digit can be the name of a variable, in which case the value of that variable is used in the expression.

The following table lists the types of operators and their meanings.

Table 2.5 ~ Operators and their Meanings

Operator	Explanation
arithmetic operators	<p>Arithmetic operators are used to perform operations on numbers. CM REXX supports the following arithmetic operators:</p> <p>+ addition</p> <p>- subtraction</p> <p>* multiplication</p> <p>/ division</p> <p>% integer division (returns integer portion of result)</p> <p>/ remainder (not modulo - can be negative)</p> <p>** exponentiation (raise to a whole number power)</p>
comparative operators	<p>Comparative operators compare two terms and return the value '1' if the result is true or the value '0' if the result is false. There are two types of comparative operators: normal and strict.</p> <p>Two terms must be absolutely identical to be strictly equal. In other words, there must be the same number of leading or trailing blanks in both terms, no padding is performed before the comparison is made, and the comparison is based on the internal character representation of the platform where the program is executed.</p> <p>For strict less-than and greater-than comparisons, the collating sequence of the internal character representation is used. Thus, these results can be platform-dependent. Further, for strict comparisons, if string1 is shorter than string2 and is also a leading sub-string of string2, string1 is considered strictly less than string2.</p>
concatenation operators	<p>Concatenation operators combine two strings to form a single string. Concatenation can be indicated in any of the following ways:</p> <p> concatenate with no intervening blanks</p> <p>blank concatenate with one blank between strings</p> <p>abuttal concatenate with no intervening blanks</p> <p>It is important to remember that concatenation is implied when two adjacent terms are not separated by some other operator.</p>
logical operators	<p>Logical operators take one or two logical values as their operands and return a logical result - 1 (true) or 0 (false). CM REXX supports the following logical operators:</p> <p>& and; returns 1 if both terms are true</p> <p> or; returns 1 if either term is true</p> <p>&& exclusive or; returns 1 if either (but not both) terms are true</p> <p>\ or ^ not; 1 becomes 0 or 0 becomes 1</p>

Comparative Operators

CM REXX supports the following comparative operators:

Normal Comparative Operators

=	Equal
^=, \=, /=, <>, ><	not equal
>	greater than
<	less than
>=, ^<, \<, /<,	greater than or equal (not less than)
<=, ^>, \>, />,	less than or equal (not greater than)

Strict Comparative Operators

==	strictly equal (identical)
^==, \==, /==,	strictly not equal
>>	strictly greater than
<<	strictly less than
>>=, ^<<, \<<, /<<,	strictly greater than or equal
<<=, ^>>, \>>, />>,	strictly less than or equal

What is a Function?

A function is a program or subroutine that accepts zero or more arguments and returns a single value. A function call in CM REXX is an expression of the form:

```
function_name([expression] [, [expression]] ... )
```

A function call can be used in any expression wherever any other term would be valid. The argument expressions can also be function calls. There cannot be intervening blanks between `function_name` and the opening parenthesis. The presence of such blanks would cause the expression to be interpreted as two unrelated symbols or expressions. CM REXX supports three types of functions:

- Built-in
- Internal
- External

The following table lists the types of functions and their meanings.

Table 2.6 ~ Functions and their Meanings

Function	Explanation
built-in functions	Built-in functions are part of the language and are always available. These functions are documented in <i>Chapter 5: Built-in Functions</i> .
internal functions	Internal functions are subroutines contained within the program and identified by a label. Internal functions are always available to the program that includes them. An internal function must return control to the main program.
external functions	External functions are stand-alone routines that can be called by a CM REXX program. They must be written in CM REXX. Functions are available to any program. CM REXX locates the external function either in the current working directory, or in a directory on the current PATH. This is discussed in detail in <i>Chapter 3: Operations</i> .

What are Special Variables?

RC, RESULT, and SIGL are special variables whose values can be set automatically during execution of a CM REXX program.

The following table lists the types of special variables and their meanings.

Table 2.7 ~ Special Variables and their Meanings

Special Variable	Explanation
RC	Set to the return code from a command.
RESULT	Set to the value returned by a called subroutine. If no value is specified on the RETURN statement in the subroutine, RESULT is dropped.
SIGL	Set to the line number of the last instruction that caused a jump to a label. This could result from a CALL or SIGNAL instruction, an internal function call, or a trapped condition.

What are Condition Traps?

While the flow of execution in a program is normally controlled by the instructions in the program, CM REXX recognizes certain conditions that can alter the flow. *Condition traps* can be set in a program so that execution flow is automatically altered whenever one of these conditions is encountered. The CALL and SIGNAL instructions allow you to enable or disable condition traps and to specify the action to be taken if a condition is raised when the trap is enabled.

The following table lists the types of conditions that can be trapped and their meaning.

Table 2.8 ~ Conditions Traps and their Meanings

Condition Trap	Explanation
ERROR	Indicates an error condition during execution of a command or that the specified host command environment was not found.
FAILURE	Indicates that execution of a command failed or that the specified host command environment was not found.
HALT	Indicates detection of an external interrupt or termination signal.
LOSTDIGITS	Indicates that the result of a numeric operation has to be rounded to fit within the current numeric digits setting. The LOSTDIGITS condition can only be trapped with SIGNAL ON. It cannot be trapped with CALL ON.
NOTREADY	Indicates an error during an I/O operation.
NOVALUE	Indicates that a symbol referenced in an expression or in a PARSE, PROCEDURE, or DROP instruction has not been assigned a value.
SYNTAX	Indicates a syntax error during program execution. The SYNTAX condition can only be trapped with SIGNAL ON. It cannot be trapped with CALL ON.

What is an Input/Output Operation?

Input and output operations in CM REXX are implemented according to the I/O model defined by Cowlshaw in *The REXX Language: A Practical Approach to Programming*, (1990: Prentice Hall). This includes both character input and output streams and the external data queue. All of the following instructions and built-in functions for performing I/O, as defined by Cowlshaw, are included in CM REXX.

The following table lists the types of input/output operations and their meanings.

Table 2.9 ~ Input/Output Operations and their Meanings

Input/Output Operation	Explanation
CHARIN	Read characters from an input stream.
CHAROUT	Write characters to an output stream. Optionally, if the output stream is a file and no output string is specified, perform a close operation on the file.
CHARS	Return the number of characters remaining in an input stream.
LINEIN	Read one line from an input stream.
LINEOUT	Write one line to an output stream. Optionally, if the output stream is a file and no output string is specified, perform a close operation on the file.
LINES	Return the number of lines remaining in the input stream.
PARSE LINEIN	Read one line from the default input stream.
PARSE PULL	Read one line from the external data queue or, if the queue is empty, from the default input stream.
PULL	Same as PARSE PULL except that the data is automatically converted to uppercase.
PUSH	Write one line to the top of the external data queue.
QUEUE	Write one line to the end of the external data queue.
QUEUED	Return the number of lines remaining on the external data queue.
SAY	Write one line to the default output stream.
STREAM	Return a string describing the state of the specified input or output stream or perform operations on the stream.

Note

Transient I/O streams include the standard input, the standard output, and pipes, including named pipes. Persistent I/O streams are disk files. The default input stream is the standard input (STDIN). The default output stream is the standard output (STDOUT). Using CM REXX I/O functions with pipes allows you to write filter programs for use with other commands or programs.

What is Parsing?

One of the strengths of the REXX language is its extensive and flexible string manipulation capability. Besides the built-in functions that perform string operations, CM REXX includes the PARSE instruction that provides a generalized and powerful mechanism for assigning portions of a string to variables.

The general form of the PARSE instruction is:

```
PARSE [UPPER] keyword [expression] template
```

where: **template** is defined by the programmer and describes the way in which the string is to be separated and assigned to variables.

Note

A detailed syntax diagram and description of the PARSE instruction can be found in *Chapter 4: Instructions*, which also includes extensive examples of the power and flexibility of PARSE.

3 Operations

This chapter discusses the CM REXX Executable program (RADREXXW.EXE), as well as the CM REXX Interpreter and Panel Manager program, RADPNLWR.

The CM REXX Executable

The CM REXX executable executes CM REXX programs. To invoke a CM REXX command, use the following format:

```
RADREXXW ProgramName Arguments
```

Note that arguments are passed as a string.

The RADPNLWR Executable

The CM executable RADPNLWR serves two functions:

- RADPNLWR is the CM Panel Manager that displays and processes the responses to CM dialogs.
- RADPNLWR is the CM REXX Interpreter that executes CM REXX programs and methods.

Note

The terms *dialogs*, *dialog boxes*, and *panels* are used interchangeably in this manual.

Invoking RADPNLWR

The command syntax options for invoking RADPNLWR are given below.

```
RADPNLWR <panel-object-name in current IDMLIB directory>
```

```
RADPNLWR <REXX-program-name with fully qualified path>
```

```
RADPNLWR
```

You can invoke RADPNLWR with one parameter, or with zero parameters. The parameter can be either the name of a panel object, or the fully qualified path and name of a REXX program to execute.

When invoked with a parameter consisting of a panel object name, RADPNLWR locates the designated panel object in the current IDMLIB directory, and uses it to display the dialog it defines. The panel object name must be eight characters or less. The current IDMLIB directory is the directory identified by the IDMLIB setting in the NVD.INI file.

When invoked with a parameter consisting of a REXX program name, RADPNLWR launches the REXX program.

When invoked with zero parameters, RADPNLWR refers to the ZMASTER object located in the current IDMLIB directory to determine what action to take. If the ZPANEL variable contains the name of a panel object in the current IDMLIB directory, the dialog defined by the panel object is displayed. If there is no panel object specified in ZMASTER.ZPANEL, RADPNLWR exits.

The following table summarizes the variables in ZMASTER that RADPNLWR refers to or sets:

Table 3.1 ~ ZMASTER Variables

ZMASTER Variable	Usage
ZPANEL	The name of the panel object for the dialog to be displayed. RADPNLWR looks for this object in the current IDMLIB location. The object name can have a maximum of eight characters.
ZPCONT	A REXX program executed by RADPNLWR can set this variable to 1 to indicate that RADPNLWR will continue execution after the REXX program exits. When control is returned to RADPNLWR from the REXX program, RADPNLWR will display the panel identified by the ZPANEL variable. If the value of ZPCONT is 0 when RADPNLWR regains control from a REXX program it has just executed, RADPNLWR exits.
ZPHEAPNO	REXX programs executed by RADPNLWR can set this variable to the heap number in a multi-heap object associated with a list box or drop-down list control that identifies the initial value for the control when RADPNLWR displays the panel. This value will be displayed as the default value of the drop-down list, or the initially highlighted selection in a list box. Note: This value is only useful if there is only one control in the dialog whose data source is a multi-heap object, because the ZPHEAPNO value will be used to identify the default for all controls in a dialog whose data source is a multi-heap object.
ZPREXEC	The fully qualified path and name of the REXX program to execute when the current dialog terminates.

Table 3.1 ~ ZMASTER Variables

ZMASTER Variable	Usage
ZPSEL	<p>Contains the heap number in a multi-heap object that is associated with the control that was last manipulated by the user (whose data source is a multi-heap object, e.g., a drop-down list or list box), containing the value last selected by the user. For example, the data source of a list box is a multi-heap object. When the user selects one of the items in the list box, RADPNLWR places the heap number (in the data source object) that contains the value that the user selected, into the ZPSEL variable.</p> <p>Note: This value is only reliable if there is only one control in the dialog whose data source is a multi-heap object, because only the most recent user selection is recorded in ZPSEL, and there is no way to tell which control was the most recently selected.</p>

RADPNLWR Log Files

Each execution of RADPNLWR generates the following ASCII log files in the IDMLOG location on the client, which defaults to `C:\Program Files\Hewlett-Packard\CM\Log` when the CM Agent is installed. The IDMLOG location is specified in the IDMLOG setting in NVD.INI.

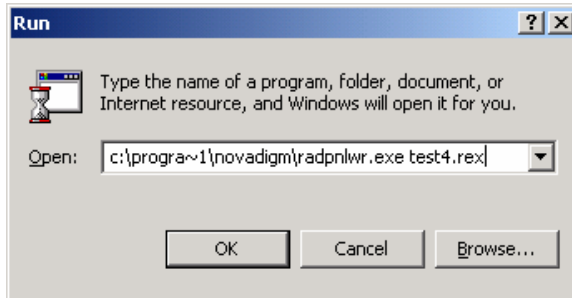
- **NEWPANEL.LOG**
This log audits the startup of RADPNLWR to the point where a panel is displayed or a REXX program is launched.
- **PNLREXX.LOG**
This log audits the REXX program interpreted by RADPNLWR.
- **<panel-name>.log**
This log audits a panel displayed by RADPNLWR. There is one log for each panel displayed. For example, if RADPNLWR displays a panel named PINSCOMP.EDM, the log file produced would be named PINSCOMP.LOG.

New log files are written each time you run a REXX method. Copy the logs to an alternate file if the contents of the log need to be retained for later use.

Executing a REXX Method from Windows

To execute a REXX method from the Windows Run dialog box

- 1 Launch the **Run** dialog box.



- 2 Type the full path of RADPNLWR followed by a space and the fully qualified name of the REXX program you want to execute. (Include the full path if the REXX program is not located in the same directory as RADPNLWR.) Click **OK**.

The CM REXX method you specified will execute.

Coding CM REXX Programs

A CM REXX program is contained in a text file with the .REX extension (for example, HWINFO.REX). You may write your REXX programs with the text editor of your choice.

CM REXX is designed to be portable across all platforms supported by CM. Ensure portability of your REXX program by following these two guidelines:

- The name of the program may be up to eight characters in length and has a .REX extension (for example: HWINFO.REX).
- The program does not contain platform-specific functions or host commands.

Including External Functions and Subroutines

CM REXX supports the use of functions or subroutines that are external to the program being executed. The following search order is used to locate external functions and subroutines:

- 1 Current working directory.
- 2 Directories specified in the PATH associated with the current command environment.

If your program includes an external function or subroutine call for which the file is not found in one of these locations, a message similar to the following appears:

```
Error 43 on line in filename: Routine not found
```

Executing Host Commands

There are a number of options for executing host commands in CM REXX. Which execution option you choose depends on the command to be executed and whether you need access to output from the commands for further processing.

- Command output is directed to STDOUT (Standard Output Stream). This is normally the display screen.
- If you require the output for later use, redirect STDOUT, and possibly STDERR (Standard Error Stream), to a file.
- Execute a host command directly by including the command as a clause in the program. The command may or may not be enclosed in quotes; however, we strongly recommend that you always enclose host commands in quotes. Quotes ensure that a host command is treated as such and they eliminate any risk of a host command being mistaken for a program variable with the same name as the host command, or any of its operands. Quotes also ensure the case-sensitivity of the host command.
- Use the ADDRESS instruction to specify the name of the host command environment that is to process the command. The default host command environment is the native operating system (EDMWIN by default, for CM).

Table 3.2 ~ Host Command Environments

Host Command	Environment
CMD	Windows 95, Windows 98, Windows NT 4.0
EDMWIN	Windows 95, Windows 98, Windows NT 4.0

Note

Command output is directed to the standard output (STDOUT), normally the display screen. You must redirect the standard output to a file if the output is required for later use.

4 Instructions

REXX instructions consist of one or more clauses that are identified by keywords, and are recognized only after meeting specific conditions. Selected instructions that are provided in CM REXX are explained in this chapter.

Overview of REXX Instructions

A REXX instruction is one or more clauses that can be specified to:

- Control the program flow,
- Manipulate data, or
- Affect the external environment.

An instruction is identified by a keyword and is recognized only when the following conditions are met:

- The keyword is the first token in the clause.
- The second token does not begin with an equals sign (=) (which implies assignment) or with a colon (:) (which indicates a label).

Instruction keywords are reserved when used in the context described above. Certain sub-keywords (such as WHILE or WHEN) are reserved within the context of particular instructions (such as DO or SELECT). Although instruction keywords and sub-keywords are not reserved outside this context, it is good programming practice not to use them as labels or as variables.

Instruction keywords and sub-keywords are not case-sensitive. Further, adjacent blanks have no effect other than to separate the keyword from surrounding tokens.

Quick Reference

The following instructions are provided in CM REXX and explained in this chapter:

Table 4.1 ~ Quick Reference to Instructions

Instruction	Description
ADDRESS	Specifies the external environment for the execution of host commands.
ARG	Retrieves the argument strings of a program or an internal routine and puts them into variables.
CALL	Invokes a routine or controls the trapping of certain conditions.
DO	Groups instructions together. Such an instruction group can be executed zero or more times depending on a conditional value and/or a repetitor.
DROP	Restores one or more variables to the un-initialized state. In the un-initialized state, the value of a variable is equal to the name of the variable in uppercase.
EXIT	Unconditionally leave a program. As an option, it can also return a result to the caller.
IF	Conditionally executes an instruction or an instruction group, or selects between alternative instructions or instruction groups.
INTERPRET	Executes dynamically created instructions.
LEAVE	Modifies the flow of control within a repetitive DO loop.
NOP	Controls the precision and format of numbers used in arithmetic operations.
NUMERIC	Controls the precision and format of numbers used in arithmetic operations.
PARSE	Assigns data to variables according to the REXX parsing rules and the specified template.
PROCEDURE	In an internal routine, protects the caller's variables from modification during execution of the routine. Also ensures that the subroutine's variables are in their un-initialized state each time the routine is called.
PULL	Simply a short form of PARSE UPPER PULL [template]. Reads a line from the CM REXX program stack. If the program stack is empty, PULL reads from the default character input stream (STDIN).
PUSH	Places a string at the top of the CM REXX program stack. Data are stacked in LIFO (last-in-first-out) order.
QUEUE	Places a string at the bottom of the CM REXX program stack. Data is stacked in FIFO (first-in-first-out) order.
RETURN	Return control from a REXX program or internal routine to its caller. Optionally, can also return a value.
SAY	Writes a line to the default character output stream.
SELECT	Conditionally execute one of several alternative instructions.
SIGNAL	Causes an abnormal change in the flow of control or controls the trapping of certain conditions.
TRACE	Traces execution flow in a program and is used primarily for debugging.
UPPER	Converts one or more variables to uppercase.

ADDRESS

Syntax

```
ADDRESS [environment [expr1]] [WITH redirect]
        [[VALUE] expr2]
```

Description

The ADDRESS instruction specifies the external environment for the execution of host commands.

Parameters

Parameter	Explanation
environment	Name of host command environment for subsequent host commands. Normally, the default host command environment is the native operating system, though this cannot be the case for applications that embed CM REXX as a macro language. The following additional host command environments are normally supported: UNIX, sh, csh, ksh, command, CMD, and DOS.
unix	UNIX
sh	The Unix Bourne shell; used for commands that are available only in the Bourne shell or for command syntax specific to the Bourne shell; this is the default shell used by the default host command environment (Unix).
csh	The Unix C shell; used for commands that are available only in the C shell or for command syntax specific to the C shell.
ksh	The UNIX Korn shell; used for commands that are available only in the Korn shell or for command syntax specific to the Korn shell.
command	A special host command environment that bypasses normal shell expansions; used for commands with operands that would normally be expanded by the shell, such as "*"; no shell is used; the command is executed directly; because no shell is invoked, piping (), redirection (>, >>, <, etc.), filename expansions (using *, ?, [], etc.), and back grounding (&) are unavailable.
CMD	The NetWare, OS/2, Windows NT, and Windows 95 shells. Note: Currently, CM supports the standard 'out of the box' environments, including COMMAND, CMD and the UNIX variants.
expr1	Host command to be executed. This can be a literal string or an expression that evaluates to a host command. When expr1 is specified, ADDRESS sends a single command to the specified environment. If expr1 is omitted, ADDRESS causes a change to the default host command environment, which persists until it is explicitly changed again or until the program exits, whichever comes first. When a new host command environment is specified, this becomes the primary host command environment. CM REXX retains the previous environment name as the alternate environment. Repeated execution of ADDRESS without operands has the effect of a toggle between the primary and alternate environments.
ADDRESS [VALUE] expr2	Equivalent to ADDRESS environment. expr2 is an expression that evaluates to the name of a host command environment. If expr2 does not begin with a symbol or a literal string (if it starts with a special character), you may omit the sub-keyword VALUE.

Parameter	Explanation
-----------	-------------

WITH <i>redirect</i>	<i>redirect</i> represents the keyword syntax that supports I/O redirection. This extended format is only available with Address CMD or UNIX (sh, ksh, csh), and not COMMAND. In the case of Windows, the redirection only works if the command executing is a "console" command that writes output to standard output. Most Windows commands do not write to "standard output." For example the command "xcopy" writes to "standard output" but Notepad does not.
----------------------	--

Syntax is as follows:

INPUT	PULL
	STEM stem_name
	STREAM file
	NORMAL

OUTPUT	[REPLACE] PUSH
	[APPEND] QUEUE
	STEM stem_name
	STREAM file
	NORMAL

ERROR	[REPLACE] PUSH
	[APPEND] QUEUE
	STEM stem_name
	STREAM file
	NORMAL

INPUT specifies redirection of standard input for the command. **OUTPUT** specifies redirection of standard output. **ERROR** specifies redirection of standard error. These keywords may be used individually or in any combination. When used in combination, the instruction has the form:

address UNIX *cmd* with input *ikey* output *okey* error *ekey*

where *cmd* is the command to be executed and *ikey*, *okey*, and *ekey* are additional keywords for input, output, and error, respectively.

REPLACE indicates that command standard output or standard error should replace existing data in the target specified. This is the default. **APPEND** indicates that command standard output or standard error should be appended to existing data in the target specified.

The remaining keywords indicate the source (for input) or target (for output and error) of I/O redirection.

PULL causes command input to be taken from the REXX program stack. **PUSH** and **QUEUE** redirect command output or error to the REXX program stack in the same manner as the **PUSH** and **QUEUE** instructions. These keywords are REXX extensions to the ANSI standard and should not be used if portability to other platforms is a consideration.

STEM specifies that the source of command input or the target of output or error is a

Parameter	Explanation
-----------	-------------

stem in the current program.

stem_name is the name of the stem to be used. It must be specified in the form stem., the trailing "." being required to distinguish it from an ordinary variable.

For **INPUT**, you must set stem_name.0 to the number of elements in the stem. stem_name.1 through stem_name.n contain the data to be redirected. For **OUTPUT** or **ERROR**, stem_name.0 is set automatically to the number of elements created in the stem. stem_name.1 through stem_name.n contain the data returned from the command.

STREAM specifies that the source of command input or the target of output or error is a file stream. **file** specifies the name of the file. It is recommended that **file** be enclosed in quotes (UNIX filenames are case sensitive and may also contain characters that would cause them to appear to REXX as an expression).

NORMAL resets the source of command input or the target of output or error back to the terminal. When **NORMAL** is specified, it must be the only keyword following **INPUT**, **OUTPUT**, or **ERROR**.

Usage Notes

Applications that embed CM REXX as a macro language can define additional host command environments and/or set a different default.

The current setting of **ADDRESS** is accessible through the **ADDRESS** built-in function, described in detail in *Chapter 5: Built-In Functions*.

Any host command sent to the default host command environment, or to one of the automatically recognized environments, creates a new process to execute the command. When the command completes, the created process terminates. If the command changes an attribute that is unique for each process (such as current working directory), the change is associated with the created process only, and has no effect on the process in which CM REXX is running.

Example 1

The following program fragment captures the output of the MS-DOS "dir" command in a file for later use.

```
address CMD 'dir > filelist'
```

Example 2

The following program fragment executes a C shell command to capture the session command history in a file for later use.

```
cmd_list = '/tmp/cmd.history'  
address csh 'history >' cmd_list
```

Example 3

The following program fragment alternates between two host command environments to execute commands that are specific to those environments.

```
cmd_list = '/tmp/cmd.history'  
home_file_list = '/tmp/home.list'  
here_file_list = '/tmp/here.list'  
sales_file_list = '/tmp/sales.list'  
address UNIX  
'ls -l >' here_file_list  
address csh
```

Example 4

In the following line, ~ is C-shell short hand for \$HOME.

```
'ls -l ~/reports >' home_file_list  
address /* resets environment name to UNIX */  
'ls -l > /home/sales/reports'  
address /* resets environment name to CSH */  
'history >' cmd_list
```

Example 5

The following program fragment captures the output of the UNIX "ls -l" command in a file for later use.

```
address UNIX 'ls -l' with output stream 'files'
```

Example 6

The following program scans a Windows directory for *.txt files and places the output of the dir command in the stem variable Text.

```
/*-----*/  
/*  L i s t D i r          */  
/*-----*/  
Trace Off  
Cmd = "Dir /b *.txt"  
Text.0 = 0  
Address CMD Cmd With Output Replace Stem Text.  
Do tt = 1 to Text.0  
  Say Text.tt  
End tt  
Exit 0
```

ARG

Syntax

ARG [template]

Description

The ARG instruction retrieves the argument strings of a program or an internal routine and puts them into variables.

Parameters

Parameter	Explanation
template	The parsing template that defines how the argument strings are assigned to variables. For details on parsing templates, refer to the PARSE instruction in this chapter. If template is omitted, the ARG instruction has no effect.

Usage Notes

The ARG instruction is simply a short form of PARSE UPPER [ARG template]. Thus, characters in the argument strings are translated to uppercase and then parsed into variables according to normal parsing rules (refer to the PARSE instruction in this chapter for details). Use PARSE ARG to preserve the case of the argument strings.

As with the PARSE instruction, ARG can be used repeatedly with different templates to separate the argument strings in different ways.

The argument strings and information about the argument strings are also accessible from the ARG built-in function, described in *Chapter Five: Built-In Functions*.

Example 1

The following program, named "bday", accepts a single argument for use in an output string.

```
arg who
say 'Happy birthday, ' who'!'
```

If the user types "bday Susan"
the output is "Happy birthday, SUSAN!"

If the user types "bday Jean Luc"
the output is "Happy birthday, JEAN LUC!"

Example 2

The following program fragment accepts a maximum of two arguments for processing; the third and subsequent arguments are discarded.

```
arg order_number part_number .
if order_number = '' then
  call display_order_list
if part_number = '' then
  call display_parts_list
```

Example 3

The following program fragment illustrates repeated use of ARG to separate the argument strings in different ways.

```
today = date(s)
say today
call breakup today
exit
breakup:
arg thisdate
arg year +4 month +2 day
arg +2 yr +2 +1 mo +1 +1 dy
say thisdate
say year month day
say yr mo dy
return
```

The output is:

```
19940303
19940303
1994 03 03
94 3 3
```

CALL

Syntax

```
CALL name { [expr] [, [expr]] ... }  
ON { condition [NAME trapname] }  
OFF { condition }
```

Description

The CALL instruction invokes a routine or controls the trapping of certain conditions.

Parameters

Parameter	Explanation
name	Names the subroutine to be invoked. It can refer to any of these types of routines: Internal routine Any subroutine or function contained within the current program and identified by a label. Built-in function One of the CM REXX built-in functions described in <i>Chapter Five: Built-In Functions</i> or one of the CM-specific functions described in <i>Chapter Six: Using Extensions</i> . External routine An external program written in REXX or a function written in a language other than REXX that has been added to the CM REXX interpreter, or that is part of an application that embeds CM REXX as a macro language.
expr	Any valid REXX expression. The expressions are evaluated from left to right with the results passed to the called routine as arguments.
ON, OFF	Sub-keywords of CALL that control the trapping of certain conditions. ON enables a condition trap. OFF disables a condition trap. Using CALL in this manner is similar to the use of SIGNAL.
condition	Simple symbols which are taken as constants.
NAME trapname	

Usage Notes

name must be either a symbol or a literal string. If it is a literal string, it can refer only to a built-in function or an external routine, since the search for internal routines is bypassed.

If the routine returns a value, it is assigned to the special variable RESULT. If the routine does not return a value, RESULT is dropped.

If **name** is an internal routine, all variables are available to both the subroutine and the caller. Use the PROCEDURE instruction, described in this chapter, to protect variables in the caller from undesired or unexpected modification by the called routine. The EXPOSE option of the PROCEDURE

instruction allows you to make selected variables from the caller available to the subroutine.

If **name** is an internal routine, the special variable SIGL is set to the line number of the CALL instruction when control is passed to the subroutine. If the routine uses the PROCEDURE instruction, you must EXPOSE SIGL if the line number of the CALL instruction is to be available for debugging purposes while in the subroutine.

An internal routine can call other internal routines or external routines. Eventually, a subroutine must exit, or return control to its caller using a RETURN instruction.

The following conditions can be controlled using the CALL instruction:

Condition	Explanation
ERROR	Indicates an error condition during execution of a command, or that the specified host command environment was not found.
FAILURE	Indicates that execution of a command failed, or that the specified host command environment was not found.
HALT	Indicates detection of an external interrupt or termination signal.
NOTREADY	Indicates an error during an I/O operation.

The following state information is saved when making a call to an internal subroutine, and is restored when control is returned to the caller:

State Information	Explanation
Status of DO loops and other structures	Executing a SIGNAL in the subroutine does not deactivate DO loops in the caller.
ADDRESS settings	Both the primary and alternate ADDRESS of the caller are unaffected by ADDRESS commands in the subroutine.
CONDITION traps	Use of CALL, or SIGNAL ON or OFF, in the subroutine does not change the settings in the caller.
CONDITION information	This is the information accessed by the CONDITION built-in function.
NUMERIC settings	Settings of precision, format, or fuzz factor in the subroutine do not affect the caller.
TRACE settings	All TRACE settings, including the interactive TRACE state, are restored when control is returned to the caller.

State Information	Explanation
-------------------	-------------

Elapsed time clocks	The subroutine can inherit an elapsed time clock from the caller and may reset it during execution without affecting the caller's clock; thus, an elapsed time clock started by the subroutine is not available to the caller.
---------------------	--

Using `CALL` to control condition traps differs from using `SIGNAL` in the following ways:

- **condition** is the name of the condition to be detected. If a condition trap is enabled, when that condition occurs, control is passed to one of the following:
 - to the label specified by **trapname**, if NAME **trapname** is specified, or
 - to the label that matches **condition**, if NAME **trapname** is not specified.
- `CALL` cannot be used with the `NOVALUE` or `SYNTAX` conditions.
- State information is preserved across the `CALL` so the trap routine can return to the caller, which can resume execution; with `SIGNAL`, program execution terminates when the trap routine completes.

Example 1

The following program fragment illustrates calling an internal subroutine which returns a value.

```
if date('w') = 'Friday' then call week_report
if result = 0 then say 'Report Generated'
  else say 'Error' result 'from report program'
exit
week_report:
status = 0
: /* Some processing, during which status gets a non-zero value */
: /* if something goes wrong */
return status
```

Example 2

The following program fragment illustrates nested calls of internal and external routines.

```
parse arg first second .
call sub1 first
call sub2 second
exit
sub1:
arg what_to_do
:
:
call sub3
if result > 0 then call extern1
return
sub2:
parse arg a '*' b .
:
:
return b
sub3:
:
:
return
```

Example 3

The following program fragment uses CALL to control condition traps.

```
call on error
call on halt name interrupt
address edwin 'holycow'
:
i = 1
do 100000
  i = i + 5
  say i
end
exit
error:
say 'Error condition detected at line' sig1
return
interrupt:
say 'Ctl-C detected; exiting at your request'
exit
```

Because the EDMWIN environment does not have a command named "holycow" (and assuming there is no program in your PATH named "holycow"), this program detects the ERROR condition, displays the message, and resumes execution following the ADDRESS instruction. If the user decides to press CTL-C (an interrupt signal) during the long DO loop, the HALT condition is detected, messages are printed, and the program terminates.

Example 4

This program illustrates the use of CALL and SIGNAL together to implement a multi-way call. The program might be named "doit".

```
parse arg what .

say 'starting in main'
who_to_call = 'aaa'
call multi who_to_call, what

say 'back in main'
exit
multi: procedure
say 'now entering multi'
if arg(2) = '' then signal value arg(1)
  else do
    say 'still in multi, arg is' arg(2)
    return
  end

say 'better not see this line'
return
aaa:
say 'now in aaa'
return
```

If the program is executed by typing `doit`, then the output is:

```
starting in main
now entering multi
now in aaa
back in main
```

If the program is executed by typing `doit go`, then the output is:

```
starting in main
now entering multi
still in multi, arg is go
back in main
```

DO

Syntax

```
DO [repetitor] [conditional]
   [instr_list]
END [symbol]
```

Description

The DO instruction is used to group instructions together. Such an instruction group can be executed zero or more times depending on a conditional value and/or a repetitor.

Usage Notes

A DO instruction group consists of the DO instruction followed by one or more instruction clauses, and then the keyword END. The END keyword *must* begin a new clause. **instr_list** represents the instruction clauses included in the group. Any CM REXX instruction can appear in the group, including the DO instruction.

conditional can be any of the following, as explained in the following table:

- WHILE expr1
- UNTIL expr1

Parameter	Explanation
expr1	expr1 is any expression that evaluates to 0 or 1. expr1 is evaluated for each pass through the loop using the current values for all variables. The instruction group is repeated WHILE expr1 evaluates to 1 or UNTIL expr1 evaluates to 1.
WHILE	A WHILE condition is evaluated at the beginning of the loop. Thus, if the condition is already satisfied at the start of the first iteration, the instruction group is never executed.
UNTIL	An UNTIL condition is evaluated at the end of the loop but before the control value, if any, is incremented.

The WHILE and UNTIL keywords are reserved within the context of a DO instruction. This means that they cannot be used in any of the expressions.

Execution of a DO loop can also be modified by the execution of a LEAVE or ITERATE instruction.

Repetitor may be any of the following, as explained in the following table:

- `exprn`
- `name= exprn [TO exprn] [BY exprn] [FOR exprn]`
- `FOREVER`

Parameter	Explanation
<code>exprn</code>	exprn is any expression that evaluates to a number. It is rounded before use according to the current setting of <code>NUMERIC DIGITS</code> . When used alone or with the <code>FOR</code> keyword, exprn must evaluate to a non-negative whole number.
<code>name</code>	name is a control variable. It can be any valid symbol. name is assigned an initial value at the beginning of the loop and is stepped <code>BY</code> a specified increment <code>TO</code> a maximum value or <code>FOR</code> a designated number of iterations. The value of the control variable can be altered within the loop, but this is not normally considered to be good programming practice. Also, if the control value is a compound symbol such as "I.J", altering "J" within the loop changes the control variable and can have an unexpected and undesirable effect on the result. Again, this is not normally considered to be good programming practice.
<code>TO, BY, and FOR</code>	<code>TO</code> , <code>BY</code> , and <code>FOR</code> can be used in any combination and in any order. They are evaluated in the order in which they appear in the <code>DO</code> instruction clause. The default value for <code>BY exprn</code> is 1. The expressions associated with <code>TO</code> , <code>BY</code> , and <code>FOR</code> are evaluated only once—when the <code>DO</code> instruction is first executed. The <code>TO</code> condition and the <code>FOR</code> count are checked at the beginning of each iteration of the loop. If the <code>TO</code> condition is already satisfied at the start of the first iteration, the instruction group is never executed. The <code>TO</code> , <code>BY</code> , and <code>FOR</code> keywords are reserved within the context of a <code>DO</code> instruction. This means that they cannot be used in any of the expressions that appear in conjunction with the specification of a control variable.
<code>FOREVER</code>	The <code>FOREVER</code> keyword indicates that the instruction group should be repeated until an instruction (such as <code>LEAVE</code> or <code>RETURN</code>) is executed to exit the loop.

repetitor and **conditional** can be used separately or in combination to control the number of times an instruction group is executed.

Example 1

The following program fragment illustrates the simplest form of the DO loop. If the user types Q, the program prints a message and exits; otherwise, processing proceeds.

```
say 'Enter menu selection or Q to quit'
pull reply
if reply = 'Q' then do
  say 'Exiting at your request'
  exit
end
else call do_selection reply
```

Example 2

The following program fragment illustrates a simple repetitive DO loop.

```
say 'Enter number of rows to process'
pull reply
if datatype(reply, 'W') then do reply
  line = linein('datafile')
  call mangle_it line
end
```

Example 3

The following program fragment illustrates the use of the WHILE conditional to force continued prompting for user input until something valid is entered. It also illustrates the use of DO loops within DO loops.

```
list = 'REXX C FORTRAN LISP PL/I'
thislang = ''
do while thislang = ''
  say 'What language for this program?'
  pull thislang
  if wordpos(thislang, list) = 0 then do
    say ''
    say 'Invalid selection:' thislang
    say 'Must be one of the following:' list
    thislang = ''
  end
  say ''
end
end
```

Example 4

The following program fragment illustrates the use of `DO FOREVER`. It repeatedly displays a menu for the user to select processing options until the user chooses the `QUIT` option.

```
do forever
  'clear'
  say ''
  say ' 1 Enter sales data'
  say ' 2 Consolidate by region'
  say ' 3 Consolidate by product line'
  say ' 4 Consolidate by salesman'
  say ' 5 Statistical analyses'
  say ' 6 Monthly report'
  say ' Q Quit'
  say ''
  say 'Select processing option'
  pull option
  if option = 'Q' then leave
  interpret 'call process.'option
end
exit
```

Example 5

The following program illustrates nested `DO` loops. It finds all primes between 1 and `n`, where `n` is the calling argument. If `n` is not specified, the default is 5000; the calls to `time('e')` make this program suitable for use as a benchmark.

```
call time 'e'
arg n
if n = '' then n = 5000
  /* Calculate all non-primes in the range and mark non-primes */
  /* in an array. */
do i = 2 to n%2
  do j = 2 to n%i
    k = i * j
    a.k = 0
  end
end
  /* Look through the array and display all the primes found. */
  /* */
do i = 1 to n
  if a.i \= 0 then say i
```

```
end  
say time('e')
```


DROP

Syntax

DROP varlist

Description

The DROP instruction restores one or more variables to the un-initialized state. In the un-initialized state, the value of a variable is equal to the name of the variable in uppercase.

Parameters

Parameter	Explanation
varlist	varlist specifies the variables to be dropped. varlist is one or more symbols separated by blanks. The symbols must be valid variable names. If a symbol is enclosed in parentheses, it is a variable reference; and its value is treated as a subsidiary variable list. The subsidiary list cannot include a variable reference, that is, it must be a list of symbols representing valid variables, separated by blanks. varlist can include the same variable more than once. It can also contain variables that have never been assigned a value.

Usage Notes

Variables are dropped from left to right, with variables in subsidiary lists dropped as soon as the variable reference is found. If a subroutine drops a variable that has been exposed from the caller, then the caller's variable is dropped. If a variable in **varlist** is a stem, then all variables that begin with that stem are dropped.

Example 1

```
x = 10
drop x
say x
```

The output is:

```
X
```

Example 2

```
x.a = 'cow'
x.b = 'pig'
drop x.
say x.b
```

The output is:

```
X.B
```

Example 3

```
list = 'a b x.'  
a = 10; b = 12; c = 14  
y. = 'unknown animal'; y.12 = 'pony'  
drop (list) c  
say y.b
```

The output is:

```
unknown animal
```

Example 4

The following program fragment illustrates the relationship between the value returned by the `SYMBOL` function and `DROPPED` variables.

```
x = 100  
say symbol('x')  
drop x  
say symbol('x')
```

The output is

```
VAR  
LIT
```

Example 5

The following program fragment illustrates using `DROP` and `SYMBOL` together instead of setting a flag to test for successful processing.

```
drop testvar  
do i = 1 to lines('in_file')  
  line = linein('in_file')  
  if word(line, 5) \= 'temp' then  
    testvar = word(line, 5)  
  
end  
if symbol('testvar') \= 'LIT' then  
  say 'Good data'  
  else say 'All temps'
```

EXIT

Syntax

EXIT [expression]

Description

The EXIT instruction is used to unconditionally leave a program. As an option, it can also return a result to the caller.

Parameters

Parameter	Explanation
expression	expression is any valid CM REXX expression. Its value is returned to the caller as a character string.

Usage Notes

When the EXIT instruction is executed, the program terminates immediately. If an external subroutine is executing, EXIT and RETURN have the same effect of returning control to the caller.

It is not absolutely necessary to include an EXIT instruction at the end of your program. EXIT is implied when there are no more instructions to execute. If, however, a program contains internal subroutines, it is important to include an EXIT instruction at the end of the main program. In the absence of such an EXIT, the program would fall through into the first internal subroutine.

Example 1

```
say 'Hello world'  
exit
```

This is identical to the one-line program:

```
say 'Hello world'
```

Example 2

The following program fragment illustrates returning a value on the exit instruction.

```
exitrc = 0  
do i = 1 to 3  
  interpret 'call report.'i  
  if result \= 0 then exitrc = 4
```

```
end
exit exitrc
```

Example 3

The following program fragment illustrates the use of `exit` to terminate a program when an unexpected condition occurs. It generates a report that can only be run on the last day of the month, so if the user is running this program on any other day, it terminates automatically. It also illustrates the use of `exit` to terminate the main program to avoid falling through into the first internal routine.

```
months = 'January February March April May',
         'June July August September October',
         'November December'
days='31 leap() 31 30 31 30 31 31 30 31 30 31'
this_month = wordpos(date('m'), months)
if left(date(), 2) \= word(days, this_month)
  then exit
call setup
call do_report
exit
leap:
```

The function to calculate the number of days in February is:

```
:
:
return howmany
```

IF

Syntax

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

Description

The IF instruction is used to conditionally execute an instruction or an instruction group, or to select between alternative instructions or instruction groups.

Parameters

Parameter	Explanation
expression	expression must evaluate to 0 or 1.
instruction	instruction can be an assignment, a command, or an instruction, including IF and SELECT constructs and DO groups.
THEN	The keyword THEN followed by an instruction is required whenever the IF instruction is used. If the value of expression is 1, then the instruction following THEN is executed. If instruction is DO, then an instruction group is executed. If the value of expression is 0, the THEN instruction is bypassed. It is not necessary for the keyword THEN to begin a new clause.
ELSE	The keyword ELSE indicates alternative processing to occur when the value of expression is 0. The keyword ELSE must begin a new clause in the program. If it appears on the same line as the THEN instruction, a semicolon must be present to terminate the THEN instruction.

Usage Notes

Optional semicolons in the syntax diagram indicate that the following component can appear on the same line as the preceding component (with or without the presence of a semicolon) or can appear on a new line in the program without changing the behavior of the IF instruction.

Use the NOP instruction to indicate that nothing is to be executed following a THEN or ELSE. A null clause is not an instruction in CM REXX, so putting an extra semicolon after THEN or ELSE results in *Error 1: Incomplete DO/SELECT/IF* or *Error 8: Unexpected THEN or ELSE*.

Example 1

The simplest form of IF:

```
rc = linein('data.file')
if rc \= 0 then say 'Error reading data.file'
```

Example 2

The following program fragment still uses the simplest form of IF but uses a function that evaluates to 0 or 1 as the conditional expression.

```
val = 'abc'
if datatype(val, 'l') then
    upper_val = translate(val)
```

Example 3

The following program illustrates alternative processing using ELSE.

```
say 'Enter menu selection (1, 2, or 3)'
pull answer
if datatype(answer, 'W') then call mysub
    else call error1
```

Example 4

The following program fragment extends the previous example to illustrate the use of a more complex conditional expression.

```
say 'Enter menu selection (1-8)'
pull answer
if \datatype(answer, 'w') | answer < 1 | ,
    answer > 8 then call error1
    else call mysub
```

Example 5

The following program fragment illustrates execution of a DO loop within an IF instruction.

```
list = 'REXX C FORTRAN LISP PL/I'
say 'What language for this program?'
pull thislang
if wordpos(thislang, list) = 0 then do
  say ''
  say 'Invalid selection:' thislang
  say 'Must be one of the following:' list
end
```

INTERPRET

Syntax `INTERPRET expression`

Description The INTERPRET instruction executes dynamically created instructions.

Parameters

Parameter	Explanation
expression	expression is any valid expression that evaluates to one or more CM REXX instructions. It is executed just as if it were a line inserted into the program.

Usage Notes

For instructions such as DO, IF, or SELECT, **expression** must include the complete instruction construct. If **expression** evaluates to a DO instruction which includes a LEAVE or ITERATE instruction, the complete DO-END construct must still be present.

Label clauses are not permitted in the expression to be interpreted.

Example 1

```
say 'Enter region for this report'  
pull reply  
do_prog = 'call report.'reply  
interpret do_prog
```

If the user enters East, the variable `do_prog` evaluates to `call report.east`. The INTERPRET instruction executes the CALL instruction.

Example 2

The following program fragment illustrates a similar use of INTERPRET without the intermediate variable; it calls a different subroutine for each day of the week.

```
today = date('w')  
interpret 'call report_'today
```


ITERATE

Syntax

ITERATE [name]

Description

The ITERATE instruction modifies the flow of control within a repetitive DO loop.

Parameters

Parameter	Explanation
name	name is the name of the control variable for the loop to be iterated. name must refer to the control variable for a currently active loop. Except for case, name must exactly match the symbol specifying the control variable on the DO instruction. Substitution for compound variables does not occur in this case. If name is omitted, then the innermost active loop is iterated.

Usage Notes

When an ITERATE instruction is encountered, processing of the DO instruction list stops, and control is returned to the DO clause in the same manner as if the END keyword had been encountered.

If more than one active loop uses the same control variable, then the innermost loop is iterated. All active loops inside the loop selected for iteration are terminated.

Example

The following program fragment outputs all the odd numbers between 1 and 10.

```
do i = 1 to 10
  if i//2 = 0 then iterate
  say i
end
```

The output is:

```
1
3
5
7
9
```

LEAVE

Syntax

LEAVE [name]

Description

The LEAVE instruction causes an immediate exit from one or more repetitive DO loops.

Parameters

Parameter	Explanation
name	name is the name of the control variable for the loop to be terminated. name must refer to the control variable for a currently active loop. Except for case, name must exactly match the symbol specifying the control variable on the DO instruction. Substitution for compound variables does not occur in this case. Control passes to the instruction immediately following the END keyword which matches the selected DO. If name is omitted, the innermost active loop is terminated.

Usage Notes

Execution of the DO instruction list terminates and control passes to the instruction immediately following the END keyword as if the END had been encountered and termination conditions had been satisfied normally. If there is a control variable for the loop, it retains the value it had at the time the LEAVE instruction was executed.

If more than one active loop uses the same control variable, then the innermost loop is terminated. All active loops inside the loop selected for termination are also terminated.

Example

The following program fragment illustrates the use of LEAVE to end a DO FOREVER loop.

```
do forever
  say ' 1  Enter sales data'
  say ' 2  Consolidate by region'
  say ' 3  Consolidate by product line'
  say ' Q  Quit'
  say 'Select processing option'
  pull option
  if option = 'Q' then leave
  interpret 'call process.'option
end
```

NOP

Syntax

NOP

Description

The NOP instruction is a dummy instruction. Because the NOP instruction has no effect, it is useful within IF or SELECT instructions.

Example

The following program fragment uses NOP in a SELECT instruction where an OTHERWISE clause is required, but no OTHERWISE processing is desired.

```
parse arg startup_option rest
select
  when startup_option = 1 then
    call lookup_rest
  when startup_option = 2 then
    call gen_report rest
  when startup_option = 3 then
    call newdata rest
  otherwise nop
end
```

NUMERIC

Syntax

```
NUMERIC { DIGITS [expr1]
          { FORM [SCIENTIFIC]
              [ENGINEERING]
              [VALUE] expr2
          }
          FUZZ [expr3] }
```

Description

The NUMERIC instruction controls the precision and format of numbers used in arithmetic operations.

Parameters

Parameter	Explanation
DIGITS	DIGITS controls the precision for arithmetic operations and for the evaluation of arithmetic functions.
expr1	expr1 specifies the number of significant digits in the result of arithmetic operations or functions. expr1 must evaluate to a positive whole number that is greater than the current setting of NUMERIC FUZZ. If necessary, it is rounded according to the current setting of NUMERIC DIGITS before it is used. If expr1 is omitted, the default value is 9. The current maximum value for expr1 in CM REXX is 10.
FORM	Controls the format used for exponential notation. The format must be one of the following: SCIENTIFIC Only one, non-zero digit appears before the decimal point. ENGINEERING The exponent (power of ten) is always expressed as a multiple of three. The number of digits before the decimal point is adjusted as necessary to meet this criterion. [VALUE]expr2 expr2 must evaluate to either SCIENTIFIC or ENGINEERING. The form is set to the value of expr2 .
FUZZ	Controls the number of digits, at full precision, that are ignored for numeric comparisons.
expr3	expr3 specifies the number of digits to ignore. expr3 must evaluate to a non-negative whole number that is less than the current setting of NUMERIC DIGITS. If necessary, it is rounded according to the current setting of NUMERIC DIGITS before it is used. If expr3 is omitted, the default value is 0.

Usage Notes

It should be noted that small values of NUMERIC DIGITS can produce unexpected or undesirable results in some cases since the setting affects all computations. For example, the execution of a DO loop can be altered by unexpected rounding of the repetitor expression or the value of a control variable.

The current setting of NUMERIC DIGITS is accessible using the DIGITS built-in function described in *Chapter Five: Built-In Functions*.

The NUMERIC FORM setting can also be specified by evaluating an expression that follows the sub-keyword VALUE. **expr2** must evaluate to either SCIENTIFIC or ENGINEERING. The VALUE sub-keyword can be omitted if **expr2** does not begin with a literal string or a symbol.

The current setting of NUMERIC FORM is accessible using the FORM built-in function described in *Chapter Five: Built-In Functions*.

NUMERIC FUZZ effectively reduces the precision used for numeric comparisons to the value:

```
NUMERIC DIGITS - NUMERIC FUZZ
```

The current setting of NUMERIC FUZZ is accessible using the FUZZ built-in function described in *Chapter Five: Built-In Functions*.

Example 1

The following program fragment illustrates the results of various settings of NUMERIC DIGITS.

```
x = 123456789
do i = digits() by -2 for 3
  numeric digits i
  say 'Digits:' digits() ' - ' format(x)
end
```

The output is:

```
Digits: 9 - 123456789
Digits: 7 - 1.234568E+8
Digits: 5 - 1.2346E+8
```

Example 2

The following program fragment illustrates the effect of `NUMERIC FORM ENGINEERING` on the output of the previous example.

```
numeric form engineering
x = 123456789
do i = digits() by -2 for 3
  numeric digits i
  say 'Digits:' digits() ' - ' format(x)
end
```

The output is:

```
Digits: 9 - 123456789
Digits: 7 - 123.4568E+6
Digits: 5 - 123.46E+6
```

Example 3

The following program fragment illustrates the effect of `NUMERIC FUZZ`.

```
numeric digits 6
x = 123456; y = 123455; z = 123451
if x = y then say 'True'; else say 'False'
numeric fuzz 1
if x = y then say 'True'; else say 'False'
if x = z then say 'True'; else say 'False'
```

The output is:

```
False
True
False
```

PARSE

Syntax

```
PARSE { [UPPER] ARG  
        LINEIN  
        PULL  
        SOURCE  
        VALUE [expr] WITH  
        VAR name  
        VERSION } [template]
```

Description

The PARSE instruction assigns data to variables according to the REXX parsing rules and the specified template.

Parameters

Parameter	Explanation
template	<p>template is a list of symbols separated by blanks or patterns. The symbols are the names of variables to which data are assigned. If template is omitted, variables are not set but data are prepared for parsing in one of the following ways:</p> <ul style="list-style-type: none">• for LINEIN or PULL A line is removed from a character stream or the CM REXX program stack.• for VALUE expr is evaluated.• for VAR If the variable does not have a value, the NOVALUE condition is raised. <p>A detailed discussion of parsing templates is presented below.</p>
ARG	Indicates that the data to be parsed is the argument strings passed to the program, subroutine, or function.
LINEIN	Indicates that the data to be parsed is the next line from the default character input stream. PARSE LINEIN is simply a short form of: <pre>PARSE VALUE LINEIN() WITH [template]</pre>
PULL	Indicates that the data to be parsed is one of the following: If data is available on the CM REXX program stack, the next string on the stack is parsed. If no data is available on the program stack, data is taken from the default character input stream (STDIN). If no data is available on the default character input stream, the program pauses for input.

Parameter	Explanation
SOURCE	<p>Indicates that the data to be parsed is a special string that identifies and describes the source of the program being executed. The SOURCE string is fixed and contains the following tokens:</p> <ul style="list-style-type: none"> • The system where the program is running for CM REXX. This is the native operating system. • How the program was invoked. This is COMMAND, FUNCTION, or SUBROUTINE. • The full path name of the program. • The name of the program without the path—the default host command environment. Normally this is the native operating system, but it can be different in applications that embed CM REXX as a macro language. <p>For example, the following code in the program test5.rex, running on a Windows NT 4.0 system, located in the C:\Program Files\Hewlett-Packard\CM directory, and executed from a DOS box command line:</p> <pre>parse source x</pre> <p>results in <code>x</code> being set to:</p> <pre>NT COMMAND C:\PROGRA~1\Hewlett-Packard\CM\TEST5.REX TEST5.REX EDMWIN</pre>
VALUE	<p>Indicates that the data to be parsed is the result of evaluating expr. The keyword WITH is required to indicate the end of expr. WITH is therefore reserved in this context and cannot be included in expr.</p>
VAR	<p>Indicates that the data to be parsed is the value of the variable specified by name. name must be a symbol that is a valid variable name in the current program. The variable is not changed unless it also appears in the template.</p>

Parameter	Explanation
VERSION	Indicates that the data to be parsed is a special string describing this version of CM REXX. The VERSION string is fixed and contains the following tokens: <ul style="list-style-type: none"> language name The first four characters are "REXX" with the remainder of the token being implementation-dependent. language level This indicates the degree of compliance with the language level definitions in <i>The REXX Language</i> by Cowlishaw. Language level 4.00 indicates full compliance with the second edition (1990) of this reference. release date (three tokens) The release date of this implementation in the same format as the default for the DATE built-in function (dd Mmm yyyy). For example, the following code: <pre>parse version x</pre> results in x being set to: <pre>REXX:Open-REXX:285:Open-REXX:ASCII :SingleThread:StaticLink 4.00 13 Nov 1998</pre>

Parsing Templates

A parsing template is a symbolic pattern by which a string is broken up (parsed) and assigned to variables. A string can be split by words (delimited by blanks), by matching specific string patterns, or by explicit numeric position. Portions of the string can also be skipped or discarded. The template can include any combination of:

- **Symbols**
The variable names to which the data is assigned.
- **Patterns**
Character string for which a match is sought.
- **Positional patterns**
Absolute or relative column numbers within the string.
- **Placeholder symbols**
The ".", indicating that data are to be discarded.

Parsing by Words

The simplest form of parsing templates is comprised only of symbols. The string is separated into words with one word assigned to each variable. One possible exception is the last variable in the template, which can be assigned more than one word if the number of symbols in the template does not exactly match the number of words in the string.

Usage Notes

Leading and trailing blanks are removed from all tokens except the last. For the last token, one leading blank (the delimiter) is removed but all other leading and trailing blanks are retained.

Example 1

```
string = 'Hello world'  
parse var string first second
```

The result is:

```
first == 'Hello'  
second == 'world'
```

Example 2

```
string = 'Once upon a time in the west'  
parse var string first second rest
```

The result is:

```
first == 'Once'  
second == 'upon'  
rest == 'a time in the west'
```

Example 3

```
string = 'Long ago and far away '  
parse var string first second rest
```

The result is:

```
first == 'Long'  
second == 'ago'  
rest = ' and far away '
```

Parsing by Patterns

Another method of parsing involves matching a pattern string. This can be useful in parsing strings that contain delimiters other than blanks between words. The pattern is specified in the template as a literal string or as a variable that is set to a literal string. If the pattern is specified as a variable, the variable name must be enclosed in parentheses in the template to distinguish it from the symbols to which data is to be assigned. The string to be parsed is separated so that all characters preceding the pattern are placed into a variable.

Usage Notes

When pattern matching is used, only the pattern itself is discarded. If there are any blanks following the pattern, they become leading blanks on the next token.

Example 1

```
string = 'red, green, blue'  
parse var string color1 ',' color2 ',' color3
```

The result is:

```
color1 == 'red'  
color2 == ' green'  
color3 == ' blue'
```

Example 2

```
string = 'time and time again'  
parse upper var string a 'and' b
```

The result is:

```
a == 'TIME'  
b == ' TIME AGAIN'
```

Example 3

```
parse arg x ',' y
```

If the argument string passed to this program is "4,3", then

```
x == '4'  
y == '3'
```

Example 4

```
delim = 'or'  
string = 'You or me or them?'  
parse var string a (delim) b (delim) c
```

The result is:

```
a == 'You'  
b == ' me'  
c == ' them?'
```

Example 5

The following program fragment extends the idea of using a variable name as the pattern to show how to parse a series of strings that may include different delimiters.

```
str.0 = 3  
str.1 = 'Numbers : 1414 : 2753 : 1816'  
str.2 = 'Names - Tom - Dick - Harry'  
str.3 = 'Cars # Ford # BMW # Toyota'  
do i = 1 to str.0  
  parse var str.i what x rest  
  parse var rest a (x) b (x) c  
  say what ':' a b c  
end
```

The output is:

```
Numbers: 1414 2753 1816  
Names: Tom Dick Harry  
Cars: Ford BMW Toyota
```

Parsing by Position

When parsing by position, the template includes column numbers where the next token begins. These can be absolute or relative column numbers. Using relative column numbers permits re-positioning of the starting point for the next token and even allows you to re-parse in a different manner data which has already been assigned to variables.

Usage Notes

The value of a positional pattern is specified in the template as a whole number or as a variable that is set to a whole number. If the positional

pattern is specified as a variable, the variable name must be enclosed in parentheses in the template to distinguish it from the symbols to which data is to be assigned.

A positional pattern that is not preceded by a sign, or that is preceded by an equals sign (=), is an absolute positional pattern. A positional pattern that is preceded by a plus or minus sign is a relative positional pattern.

When an absolute positional pattern appears in the template, the preceding variable receives all data up to, but not including, that absolute position. The next variable receives data beginning at the specified absolute position.

When a relative positional pattern appears in the template, the starting position for the next assignment is calculated by adding or subtracting the specified value from the last matched position.

Use "+0" as a relative positional pattern to assign data without moving the start point for the next assignment.

Example 1

The following program fragment gives the instruction to move to the 5th column and assign the rest of the string to variable "y".

```
x = 1234567890  
parse var x 5 y
```

The result is:

```
y = '567890'
```

Example 2

The following program fragment gives the instruction to assign the data up to column 3 to "y", then move forward 4 columns and assign the rest of the string to "z".

```
x = 1234567890
parse var x y 3 +4 z
```

The result is:

```
y = '12'
z = '7890'
```

Example 3

The following program fragment gives the instruction to assign the data up to column four to "a"; to assign the data in the next five columns to "b"; to move forward one column and assign the rest of the string to "c".

```
x = 'abcdefghijklmnop'
parse var x a 4 b +5 +1 c
```

The result is:

```
a == 'abc'
b == 'defgh'
c == 'ijklmnop'
```

Example 4

The following program fragment gives the instruction to assign the data up to column four to "a", to move back two columns and assign the rest of the string to "b"; move to column one and assign the next four columns to "c".

```
x = abcdefgh
parse var x a 4 -2 b 1 c +4
```

The result is:

```
a == 'ABC'
b == 'BCDEFGH'
c == 'ABCD'
```

Example 5

```
s.0 = 3
s.1 = 'A:1414:2753:1816'
s.2 = 'B-Tom-Dick-Harry'
s.3 = 'C#Ford#BMW#Toyota'
do i = 1 to s.0
  parse var s.i what 2 x +1 a (x) b (x) c
  say what': ' a b c
end
```

The output is:

```
A: 1414 2753 1816
B: Tom Dick Harry
C: Ford BMW Toyota
```

Example 6

The following program fragment gives the instruction to move to column three; assign the rest of the string to "a" but don't move the parsing position; assign the next three characters to "b"; move forward one column; assign the rest of the string to "c".

```
x = 1234567890
parse var x 3 a +0 b +3 +1 c
```

The result is:

```
a == '34567890'
b == '345'
c == '7890'
```

Parsing with Placeholders

Parsing templates can also include placeholder symbols. The placeholder symbol is the period ("."). If a period is encountered in a template, data that would normally be assigned to a variable at that point is discarded.

Example 1

The output of the following program fragment:

```
x = 'How are you'
parse var x a . b
say a b 'be?'
```

```
is:  
"How you be?"
```

Example 2

The output of the following program fragment:

```
x = 'one potato two potato three potato four'  
parse var x a . b . c . rest  
say a b c rest
```

```
is:  
one two three four
```

Putting it All Together

Parsing templates can include any combination of the elements discussed above. This makes PARSE an extremely powerful and flexible tool for manipulating data.

PROCEDURE

Syntax

```
PROCEDURE [EXPOSE varlist]
```

Description

The PROCEDURE instruction is used in an internal routine to protect the caller's variables from modification during execution of the routine. It also has the effect of ensuring that the subroutine's variables are in their un-initialized state each time the routine is called.

Parameters

Parameter	Explanation
PROCEDURE	If present, the PROCEDURE instruction must be the first instruction following the label. All variables used in the subroutine are then local to that routine. When a RETURN instruction is executed, all these local variables are dropped and the caller's variables are restored.
EXPOSE	The EXPOSE sub-keyword allows you to selectively expose variables from the caller's environment for manipulation by the subroutine.
varlist	varlist is the list of variables to be exposed. varlist is one or more symbols separated by blanks. The symbols must be valid variable names. If a symbol is enclosed in parentheses, it is a variable reference; and its value is treated as a subsidiary variable list. The subsidiary list cannot include a variable reference, that is, it must be a list of symbols, representing valid variables, separated by blanks. varlist can include the same variable more than once. It can also contain variables that have never been assigned a value.

Usage Notes

It is not necessary for an internal routine to include a PROCEDURE instruction. If it does not, then all the variables of the caller are visible to, and can be modified by, the subroutine. Using PROCEDURE protects the caller's variables from modification by the subroutine.

Variables are exposed from left to right. When a variable reference is encountered, the variable itself is exposed first, with variables in subsidiary lists exposed as soon as the variable reference is found. If a variable in **varlist** is a stem, then all variables that begin with that stem are exposed.

Consideration should be given to the order in which variables are exposed. If a variable is to be used to expose a compound variable, then it must be exposed before the compound variable.

Example 1

The following program fragment illustrates the effect of not using `PROCEDURE` in an internal subroutine.

```
x = 10; y = 20; z = 30
call blotz
say y
exit
blotz:
say y
return
```

The output is:

```
20
20
```

Example 2

The following program fragment illustrates the effect of `PROCEDURE` alone.

```
x = 10; y = 20; z = 30
call blotz
say y
exit
blotz:
procedure
say y
return
```

The output is:

```
Y
20
```

Example 3

The following program fragments illustrate the effect of EXPOSing a variable and how modifications to the variable affect its value on return to the caller.

For the following:

```
x = 10; y = 20; z = 30
call blotz
say y
exit
blotz:
procedure expose y
say y
return
```

The output is:

```
20
20
```

For the following:

```
x = 10; y = 20; z = 30
call blotz
say y
exit
blotz:
procedure expose y
say y
drop y
return
```

The output is:

```
20
Y
```

For the following:

```
x = 10; y = 20; z = 30
call blotz
say y
exit blotz:
procedure expose y
say y
y = x
return
```

The output is:

```
20
X
```

The variable `x` was not exposed so `y` was assigned the value of the uninitialized symbol `x`.

Example 4

The following program fragment illustrates the use of variable references and the exposure of compound variables.

```
a = 1; b = 2; c = 3
x = 10; y = 20; z = 30
p. = 'unknown value'
p.1 = 100; p.2 = 200; p.3 = 300

blotz_list = 'a b c'
call blotz
say p.b
exit
blotz:
procedure expose (blotz_list) p.b
p.b
b = 4
return
```

The output is:

```
200
unknown value
```

PULL

Syntax

PULL [template]

Description

The PULL instruction reads a line from the CM REXX program stack. If the program stack is empty, PULL reads from the default character input stream (STDIN). The PULL instruction is simply a short form of PARSE UPPER PULL [template].

Parameters

Parameter	Explanation
template	template is the parsing template that defines how the data are assigned to variables. For details on parsing templates, refer to the PARSE instruction on page 79. If template is omitted, the data read by PULL are simply discarded. This is functionally equivalent to using PULL, where the template is comprised solely of the placeholder symbol.

Usage Notes

The data read are translated to uppercase and then parsed into variables according to normal parsing rules (refer to the PARSE instruction in this chapter for details). Use PARSE PULL to preserve the case of the data.

The number of lines currently available in the program stack is accessible with the QUEUED built-in function described in *Chapter Five: Built-In Functions*.

Example 1

The following program fragment processes all data currently available on the CM REXX program stack.

```
do j = 1 while queued() > 0
  pull order.j . amount.j .
end
```

Example 2

The following program fragment assumes that no data are on the program stack and that PULL will read from STDIN, normally the terminal.

```
say 'Type a menu option or "Q" to quit'
pull reply
if reply = 'Q' then exit
```

The test is valid regardless of the case in which the user types `q` since `PULL` converts to uppercase.

PUSH

Syntax

PUSH [expression]

Description

The PUSH instruction places a string at the top of the CM REXX program stack. Data are stacked in LIFO (last-in-first-out) order.

Parameters

Parameter	Explanation
expression	expression is evaluated and the result placed on the program stack. If expression is omitted, a null string is placed on the stack.

Usage Notes

Use the QUEUE instruction, described in this chapter, to place data at the bottom of the program stack.

The number of lines currently available in the program stack is accessible with the QUEUED built-in function described in *Chapter Five: Built-In Functions*.

Example 1

The following example places `not nice` at the top of the program stack.

```
and = 'not'  
shove = 'nice'  
push and shove
```

Example 2

The following program fragment illustrates the use of PUSH to place something on the stack for use by a subroutine.

```
parse arg input  
push input  
if datatype(input, 'num') then call numeric  
  else call char  
:  
exit  
numeric: procedure  
  parse pull value  
:  
return
```

```
char: procedure
parse pull string
:
return
```


QUEUE

Syntax

QUEUE [expression]

Description

The QUEUE instruction places a string at the bottom of the CM REXX program stack. Data is stacked in FIFO (first-in-first-out) order.

Parameters

Parameter	Explanation
expression	expression is evaluated and the result placed on the program stack. If expression is omitted, a null string is placed on the stack.

Usage Notes

Use the PUSH instruction, described in this chapter, to place data at the top of the program stack.

The number of lines currently available in the program stack is accessible with the QUEUED built-in function described in *Chapter Five: Built-In Functions*.

Example 1

The following example places how much longer? at the bottom of the program stack.

```
for = 'how much'  
entry = 'longer?'  
queue for entry
```

Example 2

The following program fragment illustrates use of the stack to remove a block of lines from a file in place—no intermediate file.

```
pull start_line block_size
do start_line - 1
  queue linein('data.txt')
end
do block_size
  tossit = linein('data.txt')
end
do until lines('data.txt') = 0
  queue linein('data.txt')
end
pull first
call lineout 'data.txt', first, 1
do queued()
  pull next
  call lineout 'data.txt', next
end
call lineout 'data.txt'
```

RETURN

Syntax

RETURN [expression]

Description

The RETURN instruction is used to return control from a REXX program or internal routine to its caller. It can also, optionally, return a value.

Parameters

Parameter	Explanation
expression	expression is the value to be returned to the caller. expression can evaluate to any character string, including the null string.

Usage Notes

If the program is external, the effect of RETURN is identical to that of the EXIT instruction.

If the program was invoked by the CALL instruction, it is being executed as a subroutine. In this case, the return value is optional. When control returns to the caller, the special variable RESULT is set to the value of **expression**. If **expression** is omitted, the special variable RESULT is dropped.

If the program was invoked as a function, it must return a value. This value (the result of the function) is used in the original expression at the point where the function was invoked.

Example 1

The following program fragment illustrates the simplest use of RETURN in an internal routine invoked as a subroutine.

```
say 'Please select a processing option (1-8)'  
pull reply  
interpret 'call option.'reply  
:  
:  
exit  
option.1:  
procedure expose (list1)  
:  
:  
return  
option.2:  
:  
:
```

Example 2

The following program fragment illustrates returning a value from a subroutine.

```
say 'Please select a processing option (1-8)'  
pull reply  
if reply \= 'Q' then do  
  interpret 'call option.'reply  
  if result \= 0 then signal disaster  
end  
exit  
option.1:  
procedure expose (list1)  
status = 0  
  : /* If something goes wrong in here, an */  
  : /* appropriate message is displayed & */  
  : /* status is set to a non-zero value. */  
return status  
:  
:  
disaster:  
say 'Unrecoverable error in option:' reply  
say 'Processing terminated'  
exit
```

Example 3

The following program fragment illustrates the use of RETURN in an internal routine invoked as a function.

```
months = 'January February March April May',  
        'June July August September October',  
        'November December'  
days='31 leap() 31 30 31 30 31 31 30 31 30 31'  
:  
:  
exit  
leap:
```

The leap() function calculates the number of days in February...

```
:  
:  
return how many
```

SAY

Syntax SAY [expression]

Description The SAY instruction writes a line to the default character output stream.

Parameters

Parameter	Explanation
expression	expression is evaluated and the result is written to the default output stream. If expression is omitted, the result is a null string.

Usage Notes

The default character output stream is the standard output (STDOUT), and is normally the terminal unless the standard output has been redirected.

The SAY instruction is equivalent to:

```
CALL LINEOUT , [expression]
```

In the case of SAY, however, the special variable RESULT is not set.

To view terminal output on the screen in Windows environments, you must use the WinMessageBox function. See this function for more information.

Example 1

The following program fragment writes the string `Hello world` to the standard output, normally the terminal.

```
say 'Hello world'
```

Example 2

The output of the following program fragment:

```
say 'Enter amount of sale'  
pull amount  
say 'Commission is:' amount * .06
```

is 6% of the sale amount entered.

Example 3

```
retcode = linein('data.txt')
if retcode \= 0 then
  say 'Error reading "data.txt"'
```

If the read operation fails, the message is displayed.

SELECT

Syntax

```
SELECT
when_list
  [OTHERWISE [;] instr_list]
END
```

Description

The SELECT instruction is used to conditionally execute one of several alternative instructions.

Parameters

Parameter	Explanation
SELECT	A SELECT instruction consists of the SELECT instruction followed by one or more WHEN clauses, optionally followed by an OTHERWISE clause, and terminated by the keyword END. The END keyword must begin a new clause.
when_list	when_list defines the conditions under which each alternative is selected.
OTHERWISE	The keyword OTHERWISE indicates alternative processing to occur when none of the WHEN expressions evaluates to 1. instr_list is one or more instructions to be executed if the OTHERWISE path is chosen. If instr_list is omitted, this is equivalent to using the NOP instruction.

Usage Notes

when_list is made up of one or more constructs in the following syntax:

```
WHEN expression [;] THEN [;] instruction
```

Parameter	Explanation
expression	expression must evaluate to 0 or 1.
instruction	instruction can be an assignment, a command, or an instruction, including the DO, IF, or SELECT instruction.
THEN	The keyword THEN followed by an instruction is required whenever the WHEN keyword is used. If the value of expression is 1, then the instruction following THEN is executed. If instruction is DO, then an instruction group is executed. If the value of expression is 0, then instruction is bypassed and the next WHEN expression is evaluated. It is not necessary for the keyword THEN to begin a new clause.

Optional semicolons in the syntax diagrams indicate that the following component can appear on the same line as the preceding component (with or without the presence of a semicolon), or can appear on a new line in the program without changing the behavior of the SELECT instruction.

If you are certain that one of the WHEN alternatives will be executed, the OTHERWISE clause can be omitted; however, this is generally not considered good programming practice. If none of the WHEN expressions evaluates to 1, absence of an OTHERWISE clause results in *Error 7: WHEN or OTHERWISE expected*. If present, the keyword OTHERWISE *must* begin a new clause in the program.

Use the NOP instruction to indicate that nothing is to be executed following a THEN or OTHERWISE. A null clause is not an instruction in CM REXX, so putting an extra semicolon after the THEN results in an error.

Example 1

The following program fragment illustrates the use of NOP with SELECT. If a line begins with a comment character (#) followed by a space, no action is taken.

```
do while lines('parms.txt') \= 0
  dowhat = word(linein('parms.txt'), 1)
  select
    when dowhat = 'Monthly' then call report
    when dowhat = '#' then nop
    when dowhat = 'Weekly' then call add_data
    otherwise interpret 'call' dowhat
  end
end
```

Example 2

The following program fragment illustrates the use of SELECT to choose among alternative processing options.

```
parse arg startup_option rest
select
  when startup_option = 1 then
    call lookup rest
  when startup_option = 2 then
    call gen_report rest
  when startup_option = 3 then
    call newdata rest
  otherwise call edit
end
```

SIGNAL

Syntax

```
SIGNAL { label  
        [VALUE] expression  
        ON condition [NAME trapname]  
        OFF condition  
      }
```

Description

The SIGNAL instruction causes an abnormal change in the flow of control or controls the trapping of certain conditions.

Parameters

Parameter	Explanation
label	label is the label name to which control is passed. It must be a symbol (which is treated literally) or a literal string. label must be a valid label name in the current program.

Usage Notes

As an alternative, the label name can be derived from the expression following the keyword VALUE. **expression** must evaluate to a valid label name in the current program. The keyword VALUE can be omitted if **expression** does not begin with a symbol or a literal string.

When control passes to the specified label, all active DO, IF, SELECT, and INTERPRET instructions are immediately terminated and cannot be reactivated. The line number of the SIGNAL instruction is assigned to the special variable SIGL.

The ON and OFF sub-keywords of SIGNAL control the trapping of certain conditions. ON enables a condition trap. OFF disables a condition trap. Using SIGNAL in this manner is similar to the use of CALL except that control is not returned to the program executing the SIGNAL.

condition is the name of the condition to be detected. If a condition trap is enabled, when that condition occurs, control is passed to one of the following:

- to the label specified by **trapname**, if NAME **trapname** is specified.
- to the label that matches **condition**, if NAME **trapname** is not specified.

Both **condition** and **trapname** are single symbols which are taken as constants.

The following conditions can be controlled using the SIGNAL instruction:

Condition	Explanation
ERROR	Indicates an error condition during execution of a command or that the specified host command environment was not found.
FAILURE	Indicates that execution of a command failed or that the specified host command environment was not found.
HALT	Indicates detection of an external interrupt or termination signal.
NOTREADY	Indicates an error during an I/O operation.
NOVALUE	Indicates that a symbol referenced in an expression or in a PARSE, PROCEDURE, or DROP instruction has not been assigned a value.
SYNTAX	Indicates a syntax error during program execution.

Using SIGNAL to control condition traps differs from using CALL in the following ways:

- All conditions can be trapped with SIGNAL; CALL cannot be used with the NOVALUE and SYNTAX conditions.
- SIGNAL does not return control to the program that executed the SIGNAL. With CALL, state information is preserved across the CALL so the trap routine can return to the caller, which can resume execution.

Example

The following program fragment illustrates the use of `SIGNAL` to set up traps for all conditions.

```
signal on error
signal on failure
signal on halt name interrupt
signal on notready
signal on novalue name uhoh
say 'Enter host command environment'
parse pull hce
say 'Enter command to run'
parse pull cmd
say 'Enter filename to read'
parse pull file
line = linein(file)
address hce 'more /home/'userid()'/login'
""cmd""
i = 1
do 100000
  i = i + 5
  say i
end
a = b
exit
error:
say 'Error detected at line' sigl; exit
failure:
say condition('c') 'detected at line' sigl;exit
interrupt:
say 'Ctl-C detected'; exit
notready:
say 'File' file 'not found'; exit
uhoh:
say 'Oops, no value in line' sigl; exit
```

- If the user names a non-existent host environment, the failure exit is taken.
- If the execution of the user's command failed in any way, the error exit is taken.
- If the user names a file that does not exist or for which read permission has not been granted, the notready exit is taken.
- If the user presses CTL-C during the long DO loop, the halt exit is taken.

- If the program ever gets to the line that reads `a = b`, the novalue exit is taken.

TRACE

Syntax

```
TRACE [option]
      [VALUE] expression
```

Description

The TRACE instruction traces execution flow in a program and is used primarily for debugging.

Parameters

Parameter	Explanation
option	option specifies the level of tracing to occur. Alternatively, the level can be taken from the value of expression . The keyword VALUE can be omitted if expression does not begin with a symbol or a literal string. If no trace level is specified or if option or expression evaluate to a null string, the default is N.

Usage Notes

option (or the value of **expression**) can be one of the following:

Trace Option	Explanation
A (All)	Trace all clauses before execution.
C (Commands)	Trace all commands before execution; if the command results in error or failure, show the return code as well.
E (Error)	Trace (after execution) any command that results in error; show the return code as well.
F (Failure)	Trace (after execution) any command that results in failure; show the return code as well; this is identical to TRACE N.
I (Intermediates)	Trace all clauses before execution; show intermediate results of expressions as well as substituted names; show final results of expressions; show values assigned as the result of ARG, PARSE, or PULL instructions.
L (Labels)	Trace only labels; this is particularly useful for observing the flow to and from internal routines.
N (Normal)	Trace only commands that result in failure. Show the return code. This is the default trace level.
O (Off)	Nothing is traced; interactive tracing is disabled.
R (Results)	Trace all clauses before execution; show the final results of expressions; show values assigned as the result of ARG, PARSE, or PULL instructions.

Trace output is automatically formatted according to its logical depth of nesting within the program. If TRACE R or TRACE I is specified, results are

indented an additional two spaces and are enclosed in double quotes so that leading and trailing blanks can be easily identified. The first clause traced on any line is preceded by its line number.

All trace output lines have a three-character prefix to indicate the type of data. The following prefixes are used for all trace settings:

Line Prefix	Explanation
.	Source of the clause (the data that is actually in the program).
+++	Trace message; this could include error or failure return codes, prompts at interactive trace startup, a syntax error during interactive trace, or a traceback from a syntax error during execution.
>>>	Result of an expression, the value assigned to a variable during parsing, or the return value from a subroutine or function call.
>.>	Value assigned to a placeholder during parsing.

The following additional prefixes are used when TRACE I is in effect:

Line Prefix	Explanation
>V>	Contents of a variable.
>L>	Literal (constant symbol, un-initialized variable, or literal string).
>F>	Result of a function call.
>P>	Result of a prefix operation.
>O>	Result of an operation on two terms.
>C>	Compound variable; traced after substitution and before use.

Example 1

The following program fragment includes various kinds of REXX clauses; output is shown from specifying each of the trace options as a calling argument; the program is named "traceit.rex". The file infile.txt has one line with the number 123 starting in column 1.

```
trace value arg(1)
file = 'infile.txt'
line = linein(file)
x = word(line, 1)
if datatype(x) = 'NUM' then do
  y = x + 456 / 100
  say y
end
call Subrtn
if result > 4 then address edmwins 'copy
  infile.txt outfile.txt'
exit

Subrtn:
say now in subroutine
return 4
```

Example 2

Logged output from: RADPNLWR traceit.rex a

```
Termout  EDM000010 99.202 14:09:15 4 *- file = 'infile.txt'
EDM000010 99.202 14:09:15 5 *- line = linein(file)
EDM000010 99.202 14:09:15 6 *- x = word(line, 1)
EDM000010 99.202 14:09:15 7 *- if datatype(x) = 'NUM'
EDM000010 99.202 14:09:15 7 *-          then
EDM000010 99.202 14:09:15 7 *-          do
EDM000010 99.202 14:09:15 8 *-      y = x + 456 / 100
EDM000010 99.202 14:09:15 9 *-      say y
EDM000010 99.202 14:09:15 127.56
EDM000010 99.202 14:09:15 10 *-      end
EDM000010 99.202 14:09:15 11 *- call Subrtn
EDM000010 99.202 14:09:15 15 *- Subrtn:
EDM000010 99.202 14:09:15 16 *- say now in subroutine
EDM000010 99.202 14:09:15 NOW IN SUBROUTINE
EDM000010 99.202 14:09:15 17 *- return 4
EDM000010 99.202 14:09:15 12 *- if result > 4
EDM000010 99.202 14:09:15 13 *- exit
```


Example 3

Logged output from: RADPNLWR traceit.rex c

```
EDM000010 99.202 14:25:08 REXX Environment set up
  completed successfully
EDM000010 99.202 14:25:08 127.56
EDM000010 99.202 14:25:08 NOW IN SUBROUTINE
EDM000010 99.202 14:25:08 REXX host cmd env cleanup
  completed successfully
```

Example 4

Logged output from: radpnlwr traceit.rex e

(No errors occurred)

```
EDM000010 99.202 14:27:50 REXX Environment set up
  completed successfully
EDM000010 99.202 14:27:50 127.56
EDM000010 99.202 14:27:50 NOW IN SUBROUTINE
EDM000010 99.202 14:27:50 REXX host cmd env cleanup
  completed successfully
```

Example 5

Logged output from: radpnlwr traceit.rex f

(No failure occurred)

```
EDM000010 99.202 14:29:07 REXX Environment set up
  completed successfully
EDM000010 99.202 14:29:07 127.56
EDM000010 99.202 14:29:07 NOW IN SUBROUTINE
EDM000010 99.202 14:29:07 REXX host cmd env cleanup
  completed successfully
```

Example 6

Logged output from: radpnlwr traceit.rex l

```
EDM000010 99.202 14:30:25 REXX Environment set up completed successfully
EDM000010 99.202 14:30:25 127.56
EDM000010 99.202 14:30:25 15 *-* Subrtn:
EDM000010 99.202 14:30:25 NOW IN SUBROUTINE
EDM000010 99.202 14:30:25 REXX host cmd env cleanup
  completed successfully
```

Example 7

Logged output from: radpnlwr traceit.rex n

(No failure occurred)

```
EDM000010 99.202 14:31:34 REXX Environment set up
  completed successfully
EDM000010 99.202 14:31:34 127.56
EDM000010 99.202 14:31:34 NOW IN SUBROUTINE
EDM000010 99.202 14:31:34 REXX host cmd env cleanup
  completed successfully
```

Example 8

Logged output from: radpnlwr traceit.rex o

```
EDM000010 99.202 14:32:36 REXX Environment set up
  completed successfully
EDM000010 99.202 14:32:36 127.56
EDM000010 99.202 14:32:36 NOW IN SUBROUTINE
EDM000010 99.202 14:32:36 REXX host cmd env cleanup
  completed successfully
```

Example 9

Logged output from: radpnlwr traceit.rex i

```
EDM000010 99.202 14:33:51 REXX Environment set up
  completed successfully
EDM000010 99.202 14:33:51 4 *-* file = 'infile.txt'
EDM000010 99.202 14:33:51 >L>      "infile.txt"
EDM000010 99.202 14:33:51 >>>      "infile.txt"
EDM000010 99.202 14:33:51 5 *-* line = linein(file)
EDM000010 99.202 14:33:51 >V>      "infile.txt"
EDM000010 99.202 14:33:51 >F>      "123"
EDM000010 99.202 14:33:51 >>>      "123"
EDM000010 99.202 14:33:51 6 *-* x = word(line, 1)
EDM000010 99.202 14:33:51 >V>      "123"
EDM000010 99.202 14:33:51 >L>      "1"
EDM000010 99.202 14:33:51 >F>      "123"
EDM000010 99.202 14:33:51 >>>      "123"
EDM000010 99.202 14:33:51 7 *-* if datatype(x) = 'NUM'
EDM000010 99.202 14:33:51 >V>      "123"
EDM000010 99.202 14:33:51 >F>      "NUM"
EDM000010 99.202 14:33:51 >L>      "NUM"
EDM000010 99.202 14:33:51 >O>      "1"
```

```

EDM000010 99.202 14:33:51 >>>          "1"
EDM000010 99.202 14:33:51 7 *-*          then
EDM000010 99.202 14:33:51 7 *-*          do
EDM000010 99.202 14:33:51 8 *-*          y = x + 456 / 100
EDM000010 99.202 14:33:51 >V>          "123"
EDM000010 99.202 14:33:51 >L>          "456"
EDM000010 99.202 14:33:51 >L>          "100"
EDM000010 99.202 14:33:51 >O>          "4.56"
EDM000010 99.202 14:33:51 >O>          "127.56"
EDM000010 99.202 14:33:51 >>>          "127.56"
EDM000010 99.202 14:33:51 9 *-*          say y
EDM000010 99.202 14:33:51 >V>          "127.56"
EDM000010 99.202 14:33:51 >>>          "127.56"
EDM000010 99.202 14:33:51 127.56
EDM000010 99.202 14:33:51 10 *-*         end
EDM000010 99.202 14:33:51 11 *-*         call Subrtn
EDM000010 99.202 14:33:51 15 *-*         Subrtn:
EDM000010 99.202 14:33:51 16 *-*         say now in subroutine
EDM000010 99.202 14:33:51 >L>          "NOW"
EDM000010 99.202 14:33:51 >L>          "IN"
EDM000010 99.202 14:33:51 >O>          "NOW IN"
EDM000010 99.202 14:33:51 >L>          "SUBROUTINE"
EDM000010 99.202 14:33:51 >O>          "NOW IN SUBROUTINE"
EDM000010 99.202 14:33:51 >>>          "NOW IN SUBROUTINE"
EDM000010 99.202 14:33:51 NOW IN SUBROUTINE
EDM000010 99.202 14:33:51 17 *-*         return 4
EDM000010 99.202 14:33:51 >L>          "4"
EDM000010 99.202 14:33:51 >>>          "4"
EDM000010 99.202 14:33:51 12 *-*         if result > 4
EDM000010 99.202 14:33:51 >V>          "4"
EDM000010 99.202 14:33:51 >L>          "4"
EDM000010 99.202 14:33:51 >O>          "0"
EDM000010 99.202 14:33:51 >>>          "0"
EDM000010 99.202 14:33:51 13 *-*         exit
EDM000010 99.202 14:33:51 REXX host cmd env cleanup
      completed successfully

```

Example 10

Logged output from: radpnlwr traceit.rex r

```
EDM000010 99.202 14:38:14 REXX Environment set up
  completed successfully
EDM000010 99.202 14:38:14 4 *-* file = 'infile.txt'
EDM000010 99.202 14:38:14 >>>    "infile.txt"
EDM000010 99.202 14:38:14 5 *-* line = linein(file)
EDM000010 99.202 14:38:14 >>>    "123"
EDM000010 99.202 14:38:14 6 *-* x = word(line, 1)
EDM000010 99.202 14:38:14 >>>    "123"
EDM000010 99.202 14:38:14 7 *-* if datatype(x) = 'NUM'
EDM000010 99.202 14:38:14 >>>        "1"
EDM000010 99.202 14:38:14 7 *-*          then
EDM000010 99.202 14:38:14 7 *-*          do
EDM000010 99.202 14:38:14 8 *-*          y = x + 456 / 100
EDM000010 99.202 14:38:14 >>>          "127.56"
EDM000010 99.202 14:38:14 9 *-*          say y
EDM000010 99.202 14:38:14 >>>          "127.56"
EDM000010 99.202 14:38:14 127.56
EDM000010 99.202 14:38:14 10 *-*        end
EDM000010 99.202 14:38:14 11 *-* call Subrtn
EDM000010 99.202 14:38:14 15 *-* Subrtn:
EDM000010 99.202 14:38:14 16 *-* say now in subroutine
EDM000010 99.202 14:38:14 >>>    "NOW IN SUBROUTINE"
EDM000010 99.202 14:38:14          NOW IN SUBROUTINE
EDM000010 99.202 14:38:14 17 *-*          return 4
EDM000010 99.202 14:38:14 >>>          "4"
EDM000010 99.202 14:38:14 12 *-* if result > 4
EDM000010 99.202 14:38:14 >>>          "0"
EDM000010 99.202 14:38:14 13 *-*          exit
EDM000010 99.202 14:38:14 REXX host cmd env cleanup
  completed successfully
```

UPPER

Syntax `UPPER var_list`

Description The UPPER instruction converts one or more variables to uppercase.

Parameters

Parameter	Explanation
<code>var_list</code>	var_list is the list of variables to be converted to uppercase. var_list must be a list of symbols separated by blanks. Variable references (symbols enclosed in parentheses) are not permitted.

Example 1

For the following example:

```
a = 'Hello world'  
upper a say a
```

The output is:

```
HELLO WORLD
```

Example 2

For the following example:

```
a = 'c3po'  
b = 'r2d2'  
upper a b say a 'and' b
```

The output is:

```
C3PO and R2D2
```

5 Built-In Functions

This chapter explores the powerful set of built-in functions found in CM REXX. These functions can be called by any program.

Built-In Functions Overview

Typically, a function is invoked as a term in an expression. The general form of a function call is:

```
function_name([expression][, [expression]] ... )
```

A function returns a single result that is substituted in the expression just as the value of a variable is used. A function call can be used in any expression wherever any other term would be valid. The argument expressions can also be function calls. There cannot be intervening blanks between the `function_name` and the opening parenthesis. The presence of such blanks will cause the expression to be interpreted as two unrelated symbols or expressions.

You can also invoke a function using the `CALL` instruction. In this case, the proper syntax is:

```
CALL function_name [expression] [, [expression]]...
```

If you `CALL` a built-in function, the value that it returns is assigned to the special variable `RESULT`.

The following built-in functions are available in CM REXX and will be explained in this chapter:

ABBREV	FIND *	SOURCELINE
ABS	FORM	SPACE
ADDRESS	FORMAT	STREAM
ARG	FUZZ	STRIP
B2X	GETCWD	SUBSTR
BITAND	GETENV	SUBWORD
BITOR	INDEX *	SYMBOL
BITXOR	INSERT	TIME
C2D	JUSTIFY *	TRACE
C2X	LASTPOS	TRANSLATE
CENTER	LEFT	TRUNC
CHARIN	LENGTH	UPPER
CHAROUT	LINEIN	USERID
CHARS	LINEOUT	VALUE
CHDIR	LINES	VERIFY
COMPARE	LOWER	WORD
CONDITION	MAX	WORDINDEX
COPIES	MIN	WORDLENGTH
CUSERID	OVERLAY	WORDPOS
D2C	POS	WORDS
D2X	POPEN	X2B
DATATYPE	PUTENV	X2C
DATE	QUEUED	X2D
DELSTR	RANDOM	XRANGE
DELWORD	REVERSE	
DIGITS	RIGHT	
ERRORTXT	SIGN	

* Functions provided for compatibility with IBM

General Rules for Built-In Functions

We strongly recommend that you follow these general rules when invoking built-in functions, unless otherwise noted in the description of a particular function.

- The parentheses in a function call are required—even when no arguments are specified.
- The opening parenthesis must immediately follow the function name with no intervening blanks. This is required to distinguish a function call from a reference to a simple symbol or an instruction keyword.
- Any argument identified as a string can be specified as a null string.
- Any argument identified as a number is rounded, if necessary, according to the current setting of `NUMERIC DIGITS`, before it is used in the function.
- Any argument identified as a length must be specified as a non-negative integer.
- Any argument identified as a pad must be exactly one character in length.
- Optional arguments can be omitted from the right of the function call; taking out the comma is optional.
- Any function name or function argument can be specified in upper-, lower-, or mixed case.
- Functions with arguments that are one of a specified set of characters should have the argument enclosed in quotes. Without the quotes, the argument is interpreted as an un-initialized symbol. As long as the symbol remains un-initialized, the function behaves as expected since the value of the un-initialized symbol is the symbol in uppercase (e.g., when un-initialized, the value of the symbol `f00` is `F00`).

If, however, an assignment statement sets the value of that symbol to something else, the function results in *Error 40: Incorrect call to routine*. See *Appendix A: Message Summary* for more information.

ABBREV

Syntax

ABBREV(*information*, *info* [, *length*])

Description

The ABBREV function determines if one string is a valid abbreviation of a longer string. It returns 1 if the abbreviation is valid and 0 if the abbreviation is invalid.

Parameters

Parameter	Explanation
<i>information</i>	The unabbreviated string.
<i>info</i>	The abbreviated string. When info is the null string, it matches any value of information as long as length is omitted or specified as 0.
<i>length</i>	Specifies the minimum length of info . If length is omitted, the default is the length of info .

Usage Notes

If **info** is exactly equal to the leading characters of **information** and if the length of **info** is greater than or equal to **length**, then **info** is a valid abbreviation of **information**, and the function returns 1.

If either of the above conditions is not met, then the abbreviation is invalid and the function returns 0.

Example 1

The output of:

```
valid = abbrev('month', 'mo')
```

is:

```
valid = 1
```

Example 2

The output of:

```
valid = abbrev('month', 'mo', 2)
```

is:

```
valid = 1
```

Example 3

The output of:

```
valid = abbrev('month', 'mo', 3)
is:
valid = 0
```

Example 4

The output of:

```
valid = abbrev('month', m)
is:
valid = 0
```

The value of the symbol `m`, when not specifically assigned a value, is `M`.

Example 5

The value of:

```
valid = abbrev('month', '')
is:
valid = 1
```

The null string matches any value of information.

Example 6

The output of:

```
month = 'January'
mo = 'Jan'
if abbrev(month, mo) then say 'valid'
else say 'invalid'
is:
'valid'
```

ABS

Syntax `ABS (number)`

Description The ABS function returns the absolute value of a number.

Parameters

Parameter	Explanation
number	Any valid number. The result is formatted according to the current NUMERIC settings.

Example 1

The output of the following program fragment:

```
value = abs(-98.6)
```

is:

```
value = 98.6
```

Example 2

The output of the following program fragment:

```
numeric digits 4  
number = abs(-123456.7890)  
say number
```

is:

```
1.235E+5
```

ADDRESS

Syntax ADDRESS ()

Description The ADDRESS function returns the name of the current host command environment.

Usage Notes

The host command environment can be changed using the ADDRESS instruction. For more information see *Chapter 4: Instructions*.

Example 1

The output of the following:

```
env = address()  
  
is:  
  
env = EDMWIN
```

Example 2

The following program fragment sets the default host command environment to `cmd` before executing a DOS command.

```
address cmd  
'dir > filelist'  
say address()
```

The output is:

```
CMD
```

ARG

Syntax

ARG([n [, option]])

Description

The ARG function returns the argument string or information about the argument string. This is useful for verifying arguments passed to a subroutine or function before using them in the subroutine or function.

Parameters

Parameter	Explanation
n	Indicates the argument number to be returned and must be a positive integer. When only n is specified, ARG returns the nth argument string.
option	Can be either E or O . Used in conjunction with n . E xists. If the nth argument exists, ARG returns 1; otherwise, it returns 0. O mitted. If the nth argument is omitted, ARG returns 1; otherwise, it returns 0. When both arguments are specified, ARG tests for the existence of the nth argument string.

Usage Notes

With no parameters specified, ARG returns the number of arguments passed to the subroutine or function.

Example 1

In the following example where no argument is specified:

```
call subr
:
subr:
arglist = arg()      /* arglist = 0 */
arg1 = arg(1)        /* arg1 = '' */
arg1_exist = arg(1, 'e')
```

The output is `arg1_exist = 0`; i.e., the first argument does not exist.

Example 2

In the following example:

```
call subr a,,b
:
subr:
arglist = arg()      /* arglist = 3 */
arg1 = arg(1)        /* arg1 = "A" */
arg2_omitted = arg(2,'o')
```

The output is `arg2_omitted = 1`; i.e., the second argument is omitted.

BITAND

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

Description The BITAND function returns the results of a logical AND of two strings.

Parameters

Parameter	Explanation
string1 string2	The two strings upon which the AND operation is performed. If the strings are of unequal length, the length of the result is that of the longer of the two strings. If string2 is omitted, the default is the null string.
pad	A character specified to pad the shorter string if string1 and string2 are of unequal length. Pad characters are added on the right of the shorter string before the AND is performed. If pad is omitted, the AND operation terminates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
anded = bitand('52'x, '43'x)
      is:
anded = '42'x
```

Example 2

The output of the following program fragment:

```
anded = bitand('52'x, '4343'x)
      is:
anded = '4243'x
```

BITOR

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

Description The BITOR function returns the logical inclusive OR of two strings.

Parameters

Parameter	Explanation
string1 string2	The two strings on which the OR operation is performed. If the strings are of unequal length, the length of the result is that of the longer of the two strings. If string2 is omitted, the default is the null string.
pad	A character specified to pad the shorter string if string1 and string2 are of unequal length. Pad characters are added on the right of the shorter string before the OR is performed. If pad is omitted, the OR operation terminates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
ord = bitor('52'x, '43'x')
      is:
ord = '53'x'
```

Example 2

The output of the following program fragment:

```
ord = bitor('52x', '4343'x)
      is:
ord = '5343'x'
```

BITXOR

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

Description The BITXOR function returns the logical exclusive OR of two strings.

Parameters

Parameter	Explanation
string1 string2	The two strings on which the exclusive OR operation is performed. If the strings are of unequal length, the length of the result is that of the longer of the two strings. If string2 is omitted, the default is the null string.
pad	A character specified to pad the shorter string if string1 and string2 are of unequal length. Pad characters are added on the right of the shorter string before the exclusive OR is performed. If pad is omitted, the exclusive OR operation terminates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
xord = bitxor('52'x, '43'x)
is:
xord = '11'x
```

Example 2

The output of the following program fragment:

```
xord = bitxor('52'x, '4343'x)
is:
xord = '1143'x
```

B2X

Syntax

`B2X(string)`

Description

The B2X function converts a binary string to a hexadecimal string.

Parameters

Parameter	Explanation
string	<p>The character representation of the binary data to be converted. It can be any length and can contain embedded blanks at four-digit boundaries. If string does not contain an even multiple of four digits, zeros are added on the left to make an even multiple. string is not a binary string literal—i.e., it is <i>not</i> specified in the form '1010'b.</p> <p>The value returned is a character representation of the equivalent hexadecimal string. It does not contain embedded blanks.</p> <p>The results of B2X() can be used as the input for the functions X2D() or X2C() to convert binary strings into other representations.</p>

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
hexval = b2x('0110 0001')  
  
is:  
  
hexval = '61'
```

Example 2

The output of the following program fragment:

```
charval = x2c(b2x('01100001'))  
  
is:  
  
charval = 'a'
```

CENTER

Syntax `CENTER(string, length [, pad])`
 `CENTRE(string, length [, pad])`

Description The CENTER function centers a string within a specified number of character positions. The alternative spelling CENTRE is also supported.

Parameters

Parameter	Explanation
string	The character string to be centered.
length	Specifies the total number of character positions within which string is to be centered. If string is longer than length , it is truncated to fit, as necessary, at both ends.
pad	The character that occupies character positions at either end of string . If pad is omitted, the default is blank.

Usage Note

If an odd number of characters must be truncated or padded, the excess is added or dropped on the right side of **string**.

Example 1

The output of the following program fragment:

```
greeting = center('Hello!',10)
is:
greeting = " Hello! "
```

Example 2

The output of the following program fragment:

```
news = center('Headline', 12, '*')
is:
news = "***Headline**"
```

Example 3

The output of the following program fragment:

```
quote = 'To be or not to be?'  
line_length = 18  
sayit = center(quote, line_length)  
say sayit
```

is:

```
"To be or not to be"
```

CHARIN

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

Description The CHARIN function returns a string of characters from a character input stream.

Parameters

Parameter	Explanation
name	The name of the character input stream. This can be a persistent stream such as a disk file or a transient stream such as STDIN or a pipe (including a named pipe). If name is omitted, the default is STDIN.
start	Specifies an explicit read position. It must be a positive integer and within the bounds of the input stream specified. If start is omitted, the default is the current read position. start cannot be specified for a transient input stream.
length	Specifies the number of characters to be read. If length is omitted, the default is 1. If length is specified as 0, then the function resets the read position to the value of start and returns a null string. If there are fewer characters in the stream than length , the program can wait for additional characters to become available. If it is not possible for additional characters to become available, the function returns fewer than the specified number of characters and raises the NOTREADY condition. The built-in function STREAM can be used to determine the state of a character stream.

Usage Notes

When reading disk files, use CHARIN to read less than a full line or files in which the lines do not have normal line-end terminators. For files that have normal line-end terminators, you may want to use the built-in function LINEIN to read an entire line.

When the input stream is a disk file, use of an I/O function such as CHARIN can leave the file in an open state. Thus, it may be necessary to close the file using CHAROUT, LINEOUT, or STREAM before performing subsequent read or write operations to the file.

Example 1

This example returns 5 characters from the current read position and assigns that value to the variable emp_number.

```
emp_number = charin('personnel.txt',,5)
```


Example 2

The following program fragment displays a prompt to the user. It then pauses until data is available on STDIN (in this case, characters typed at the keyboard). `CHARIN` returns a single character and assigns that value to the variable `num`. A host command then prints a file whose name is a concatenation of `report` and the character entered on the keyboard.

```
say 'Enter report number'  
num = charin()  
address cmd 'print report.'num
```

CHAROUT

Syntax

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

Description

The CHAROUT function writes a string to a character output stream and returns the number of characters remaining in the string after the write is performed.

Parameters

Parameter	Explanation
name	The name of the character output stream. This can be a persistent stream, such as a disk file, or a transient stream such as STDOUT or a pipe (including a named pipe). If name is omitted, the default is STDOUT.
string	The character string to be written.
start	The character position within the output stream at which writing of output characters begins.

Usage Notes

If **name** is a persistent stream (usually a disk file), then **string** can be omitted. In this case, one of the following actions is taken:

- If **start** is specified, CHAROUT resets the write position to the start value and the function returns 0.
- If **start** is also omitted, CHAROUT closes the output stream and the function returns 0.

Start specifies an explicit write position. It must be a positive integer and within the bounds of the output stream specified. If **start** is omitted, the default is the current write position. **Start** *cannot* be specified for a transient output stream.

The program waits until the write operation is complete. If it is not possible to write all the characters to the output stream, the function returns the number of characters not written and raises the NOTREADY condition.

When the output stream is a disk file, use of an I/O function such as CHAROUT can leave the file in an open state. Thus, it may be necessary to close the file using CHAROUT, LINEOUT, or STREAM before performing subsequent read or write operations to the file.

Example 1

The following program fragment writes the string specified by the variable `emp_number` to the file `personnel.txt`; `rc` is normally 0.

```
emp_number = 'DEV003'  
rc = charout('personnel.txt', emp_number)
```

Example 2

The following program fragment writes the string specified by the variable `emp_number` to the file `personnel.txt` beginning at the 75th character position. Note the use of `CALL` to invoke the function.

```
emp_number = 'DEV003'  
call charout 'personnel.txt', emp_number, 75
```

Example 3

The following program fragment writes `Hello world` to `STDOUT`, usually the terminal. `out_rc` is normally 0.

```
out_rc = charout(, 'Hello world')
```

Example 4

The following program fragment writes the string `Hello world` followed by a new-line character to `STDOUT`, usually the terminal. This produces the same output as say `'Hello world'`.

```
call charout , 'Hello world' || '0a'x
```

Example 5

The following program fragment:

```
Call charout 'foo.txt'
```

closes the file `foo.txt`.

CHARS

Syntax CHARS ([name])

Description The CHARS function returns the number of characters remaining in a character input stream.

Parameters

Parameter	Explanation
name	The name of the character input stream. This can be a persistent stream, such as a disk file, or a transient stream such as STDIN or a pipe (including a named pipe). If name is omitted, the default is STDIN.

Usage Notes

When the input stream is a transient stream, CHARS returns 1 if there is any data available in the stream, and 0 if there is no data available in the stream.

When the input stream is a disk file, use of an I/O function such as CHARS can leave the file in an open state. Thus, it may be necessary to close the file using CHAROUT, LINEOUT, or STREAM before performing subsequent read or write operations to the file.

Example 1

In the following example `count` is set to the number of characters in the disk file named `myfile`.

```
count = chars('myfile')
```

Example 2

The following program fragment tests for the existence of a file. If the file exists (the value of the CHARS function is greater than zero), the file is deleted before proceeding.

```
if chars('myfile') > 0 then  
  address CMD 'erase myfile'
```

CHDIR

Syntax CHDIR([directory])

Description The CHDIR function changes the current working directory for the process in which the CM REXX program is running.

Parameters

Parameter	Explanation
directory	Specifies the path to which the current working directory is to be set. directory can be any valid directory path on your system. The value you specify for directory can be any character string that would validly effect a directory change if typed in at the command prompt. If directory is omitted, the default is the path specified by the HOME environment variable.

Usage Notes

CHDIR returns 0 if the current working directory is successfully changed. Otherwise, it returns non-zero.

To effect a directory change for operations within the current program, you *must* use CHDIR. If you use the host command CD, that command is executed in a different process from your CM REXX program and has no effect on the current working directory for the program.

Example

In the following example, the current directory was `c:\progra~1\Hewlett-Packard\CM` when the program was started.

```
olddir = getcwd()
cd_rc = chdir('lib')
newdir = getcwd()
say olddir
say newdir
```

The output is:

```
c:\progra~1\Hewlett-Packard\CM
c:\progra~1\Hewlett-Packard\CM\lib
```

COMPARE

Syntax COMPARE(string1, string2 [, pad])

Description The COMPARE function determines if two strings are identical.

Parameters

Parameter	Explanation
string1 string2	The two strings to be compared. If the strings are of unequal length, the shorter string is padded before the comparison is performed.
pad	Specifies the character to be appended to the shorter of the two strings. If pad is omitted, the default is blank.

Usage Notes

The COMPARE function returns 0 if the strings are identical. If the strings are not identical, the function returns the number of the left-most character position at which a discrepancy was detected.

Example 1

In the following example `comp_rc` is 0. The first string is padded with blanks to make it equal in length to the second string; this also makes it identical to the second string.

```
comp_rc = compare('a', 'a ')
```

Example 2

In the following `comp_rc` is 1; the first argument (the symbol `q`) has the value `Q` since it has not been assigned a value; `Q` and `q` are not identical.

```
comp_rc = compare(q, 'q')
```

Example 3

In the following example `c` is 6. **Pad** is omitted so the value of `a` is padded with blanks, making the string effectively "alpha ". The first discrepancy is in position 6, where `a` has a blank and `b` has a `b`.

```
a = 'alpha'  
b = 'alphabet'  
c = compare(a, b)
```

CONDITION

Syntax `CONDITION([option])`

Description The `CONDITION` function returns information about the current trapped condition.

Parameters

Parameter	Explanation
option	Specifies the type of information to be returned. Can be any string beginning with one of the characters shown below. If option is omitted, the default value is I. If option is specified, it must be one of the following: C, D, I, or S. C (condition name) The name of the current trapped condition. D (description) The descriptive string associated with the current trapped condition. If no descriptive string is available, this option returns a null string. I (instruction) The instruction executed when the condition was trapped. This is either <code>CALL</code> or <code>SIGNAL</code> . S (state) The state of the current trapped condition. This is <code>ON</code> , <code>OFF</code> , or <code>DELAY</code> .

Usage Notes

The descriptive strings for each condition are as follows:

Condition	Explanation
<code>ERROR FAILURE</code>	The string that was passed to the external environment which resulted in the condition being raised.
<code>HALT</code>	Any string associated with the halt request by the external environment. This can be a null string.
<code>NOVALUE</code>	The derived name of the variable referenced, which raised the condition.
<code>NOTREADY</code>	The name of the stream being accessed when the condition was raised. If this is a default stream, then a null string is returned.
<code>SYNTAX</code>	Any string associated with the error by the interpreter. This can be a null string.

Example

The following program fragment illustrates the use of the `CONDITION` function to implement a generic condition trap.

```
signal on novalue name trapit
signal on syntax name trapit
signal on notready name trapit
signal on halt name trapit
signal on error name trapit
signal on failure name trapit
:
:
exit
trapit:
say condition('c') 'raised at line:' sigl
select
  when condition('c') = 'NOVALUE' then
    str = 'Bad variable is:'
  when condition('c') = 'ERROR' then
    str = 'Bad command is:'
  when condition('c') = 'FAILURE' then
    str = 'Bad command is:'
  otherwise
    str = 'Condition string (may be null):'
  end
say ''
say str condition('d')
exit
```

COPIES

Syntax `COPIES(string, n)`

Description The COPIES function returns a string composed of a specific number of concatenated copies of an original string.

Parameters

Parameter	Explanation
string	The original string to be copied.
n	Specifies the number of copies of string to concatenate. n must be a positive number or zero.

Example 1

The output of the following program fragment:

```
newstring = copies('ho', 3)
is:
newstring = 'hohoho'
```

Example 2

The output of the following program fragment:

```
str = '616263'x
newstring = copies(str, 2)
say newstring
is:
abcabc
```

Example 3

The output of the following program fragment:

```
do i = 0 to 3
  say copies('ho', i)
end
is:
```

```
ho
hoho
hohoho
```

The first line of output is a null string since `n` is 0.

Example 4

The following program fragment uses `COPIES` to provide leading zeroes so that each number is exactly six characters long.

```
num.0 = 37
:
:
do i = 1 to num.0
  num.i = copies('0', 6-length(num.i)) || num.i
end
```

CUSERID

Syntax

CUSERID()

Description

The CUSERID function returns the User ID of the user currently logged on to the computer. It is identical to the USERID built-in function.

Example

The following program fragment displays the User ID of the individual running the program.

```
say cuserid()
```

C2D

Syntax `C2D(string [, n])`

Description The C2D function converts a character string to the decimal value of its ASCII representation.

Parameters

Parameter	Explanation
string	The character string to be converted.
n	If n is specified, then string is interpreted as a signed number. If the left-most bit is zero, then the number is positive. Otherwise, the number is a twos-complement negative number. If n is 0, the function returns 0. If n is omitted, the return value is positive.

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
decval = c2d('abc')  
  
is:  
  
decval = '979899'
```

Example 2

The output of the following program fragment:

```
hexval = d2x(c2d('abc'))  
  
is:  
  
hexval = '616263'
```

C2X

Syntax `C2X(string)`

Description The C2X function converts a character string to its hexadecimal representation.

Parameters

Parameters	Explanation
string	The string to be converted. The function returns the character representation of its hexadecimal value. If string is the null string, then C2X returns the null string.

Usage Note

C2X can be used in conjunction with X2B to convert character strings to their binary representation.

The examples below are in United States ASCII.

Example 1

The output of the following program fragment:

```
hexval = c2x('a')  
is:  
hexval = '61'
```

Example 2

The output of the following program fragment:

```
hexval = c2x('61'x)  
is:  
hexval = '61'
```

Example 3

The output of the following program fragment:

```
bval = x2b(c2x('a'))  
is:
```

```
bval = '01100001'
```

DATATYPE

Syntax

DATATYPE (string [,type])

Description

The DATATYPE function tests the data type of a string. It can be used to determine the data type or to determine if the data is of the desired type.

Parameters

Parameters	Explanation
string	The string for which the data type is to be tested.
type	If specified, is one of the valid data types.

Return Values

Return Value	Explanation
NUM	string is a number that can be added to zero without error
CHAR	string does not meet the criteria for NUM.
0 or 1	The function returns 1 if string matches the specified type; otherwise, it returns 0. type must be one of the following: A, B, L, M, N, S, U, W, X . A (alphanumeric) string contains only the characters a-z, A-Z, or 0-9. B (binary) string contains only binary digits (0 and 1), possibly with embedded blanks between groups of four digits. L (lowercase) string contains only the characters a-z. M (mixed case) string contains only the characters a-z or A-Z. N (number) string is a number; DATATYPE without the type argument would return NUM. S (symbol) string contains only those characters that are valid in a CM REXX symbol. U (uppercase) string contains only the characters A-Z. W (whole number) string is a valid whole number under the current setting of NUMERIC DIGITS. X (hexadecimal) string contains only valid hexadecimal digits (a-f, A-F, or 0-9), possibly with embedded blanks, or string is the null string.

Example 1

The output of the following fragment:

```
type = datatype('abc')
      is:
type = 'CHAR'
```

Example 2

The output of the following fragment:

```
val = 10
type = datatype(val)
      is:
type = 'NUM'
```

Example 3

The output of the following fragment:

```
string = 'April 15'
type = datatype(string, 'A')
      is:
type = 1
```

Example 4

The following program fragment tests the data type of a variable to determine if it is composed entirely of lowercase characters. If so, the string is converted to uppercase.

```
val = 'abc'
if datatype(val, 'L') = 1 then
  upper_val = translate(val)
```

Example 5

The following program fragment prompts for user input and then verifies that the user typed a valid whole number. The `DATATYPE` function is used as a logical symbol since its value will be either 0 or 1. If the user input is a whole number, `DATATYPE` returns 1 (true).

```
say 'Enter menu selection (1, 2, or 3)'  
pull answer  
if datatype(answer, 'W') then call mysub  
else call error1
```

Example 6

The following program fragment extends the previous example to validate not only the type of user input but also that it is within the valid range.

```
say 'Enter menu selection (1-8)'  
pull answer  
if \datatype(answer, 'w') | answer < 1 | ,  
   answer > 8 then call error1
```

DATE

Syntax

```
DATE ([out_option [, date_string, in_option]])
```

Description

The DATE function returns the current date or converts dates from one format to another. Date format conversion occurs when DATE is coded with the date_string and in_option arguments; it permits arithmetic operations to be performed on dates of any format.

Parameters

Parameter	Explanation
out_option	Specifies the format in which the date is returned. Valid values for out_option are: B, C, D, E, J, M, N, O, S, U, and W . B (base) The number of complete days since the base date of 1 January 0001. Complete days include the base date but do not include the current day. The date format returned is dddd. C (century) The number of days in the current century. The count of days includes 1 January of the century year (such as 1900) and the current day. The date format returned is dddd. D (days) The number of days in the current year. The count includes the current day. The date format returned is ddd. E (European) The current date in the standard European format of dd/mm/yy. J (Julian) The current date in the format yyddd. yy is the last two digits of the current year. ddd is the number of days, including today, in the current year. M (month) The full English name of the current month, beginning with a capital letter. N (normal) The current date in the format dd Mmm yyyy. This is the same format as the default returned when out_option is omitted. O (ordered) The current date in the format yy/mm/dd. S (standard) The current date in the format yyyyymmdd. U (USA) The current date in the standard United States format of mm/dd/yy. W (weekday) The full English name for the current day of the week, beginning with a capital letter.
date_string	Specifies the date to be converted. It may be a literal string, a variable reference, or an expression that evaluates to a date. It must be in one of the valid out-option date formats, except Weekday or Month. These are: B, C, D, E, J, N, O, S, and U .
in_option	Specifies the format of date_string and must be one of the date format options for out_option, other than Weekday or Month. Thus, valid values for in_option are: B, C, D, E, J, N, O, S, and U .

Usage Note

If **out_option** is omitted, the format returned is: `dd mmm yyyy`
where:

dd is the current day of the month, without leading zeroes.

mmm is the first three characters of the English name of the current month.

yyyy is the four-digit representation of the current year.

Example 1

The output of the following program fragment:

```
today = date()
```

could be:

```
today = '4 Jul 1994'
```

Example 2

The output of the following program fragment:

```
thisdate = date('U')
```

could be:

```
thisdate = '07/04/94'
```

Example 3

The output of the following program fragment:

```
sdate = date('s')
```

could be:

```
sdate = '19940704'
```

Dates in this format are suitable for sorting and other ordering operations.

Example 4

The following program fragment converts a data in "normal" REXX format to a format suitable for sorting:

```
newdate = date('s',_ '04 Jul 1998', 'n')
```

The converted date format in newdate is "19980704".

Example 5

The following program fragment adds 90 days to the current date:

```
today = date()
plus90=date('u', date('b', today, 'n')+90, 'b')
```

If today is 04/30/98, plus 90 is "07/29/98".

Example 6

The following program fragment runs a quarterly report only if the current month is one of those included in the list of reporting months.

```
report_months = 'March June September December'
if wordpos(date('M'), report_months) \= 0 then
  call quarterly_report
else say 'Not a reporting month'
```

Example 7

The following program fragment calls a different subroutine for each day of the week. When run on Monday, it calls `report_Monday` and so forth.

```
today = date('w')
interpret 'call report_'today
```

Example 8

The following program fragment is a slightly different approach to the previous example. In this case, the subroutines do not have names that can easily be related to any date format. This example takes advantage of the fact that `date('b')//7` returns a numeric value for the day of the week (Monday = 0).

```
sub.0 = 'start_week'  
sub.1 = 'two_days'  
sub.2 = 'hump_day'  
sub.4 = 'four_days'  
sub.5 = 'tgif'  
sub.6 = 'weekend'  
sub.7 = 'weekend'  
daynum = date('b')//7  
interpret 'call' sub.daynum
```

DELSTR

Syntax DELSTR (string, n [, length])

Description The DELSTR function deletes one or more characters within a string.

Parameters

Parameter	Explanation
string	The string from which characters are to be deleted.
n	Specifies the character position within string where deletion begins. n must be a positive number. If n is greater than the length of string, then string remains unchanged.
length	Specifies the number of characters to be deleted. length must be non-negative. If length is omitted, all remaining characters in the string, beginning at position n , are deleted.

Example 1

The output of the following program fragment:

```
str = delstr('string', 4)
is:
str = 'str'
```

Example 2

The output of the following program fragment:

```
airborne = 'paratroops'
infantry = delstr(airborne, 1, 4)
is:
infantry = 'troops'
```

Example 3

The following program fragment reads lines of an input file of addresses, parses for the zip code, and puts all zip codes into the five-digit form rather than the "zip plus four" form. Any zip codes longer than five digits (as in 60018-6300) have the sixth and all subsequent characters deleted; any zip codes in the five-digit form remain unchanged.

```
do i = 1 to lines('addrfile')
  parse value linein('addrfile') with +95 zip .
  5digit_zip.i = delstr(zip, 6)
end
```


DELWORD

Syntax `DELWORD (string, n [, length])`

Description The DELWORD function deletes one or more blank-delimited words in a string.

Parameters

Parameter	Explanation
string	Is the string from which words are to be deleted.
n	Specifies the number of the first word to be deleted. n must be a positive number. If n is greater than the number of words in string , then string remains unchanged.
length	Specifies the number of words to be deleted. length must be non-negative. If length is omitted, all remaining words in the string, beginning with word n , are deleted.

Example 1

The output of the following program fragment:

```
s = delword('how now brown cow', 2)
is:
s = 'how'
```

Example 2

The output of the following program fragment:

```
s = delword('hi there world', 2, 1)
is:
s = hi world'
```

Example 3

In the following program fragment:

```
parse var var1 first . . rest
newvar = first rest
newvar2 = delword(var1, 2, 2)
```

- When `var1='Raining cats and dogs'`, then both `newvar` and `newvar2` have the value `'Raining dogs'`.

- **When** var1='Raining cats and dogs', **then** newvar='Raining dogs' **but** newvar2='Raining dogs'.

DIGITS

Syntax DIGITS ()

Description The DIGITS function returns the current setting of NUMERIC DIGITS.

Usage Notes

The description of the NUMERIC instruction in the previous chapter, *Chapter 4: Instructions*, contains information on using NUMERIC DIGITS to control the precision of arithmetic operations and the evaluation of arithmetic functions.

Example 1

In the following example $x = 9$ if the default for NUMERIC DIGITS is in effect.

```
x = digits()
```

Example 2

The following program fragment tests the current setting of NUMERIC DIGITS and resets it if necessary before evaluating the FORMAT function. If precision is not tested and reset, the FORMAT function would raise *Error 40: Incorrect call to routine*. By testing and, if necessary, resetting NUMERIC DIGITS, the FORMAT function can be evaluated and $x = '-1.2E+2'$ (assuming the default setting of NUMERIC FORM).

```
if digits() > 2 then numeric digits 2
x = format(-123,3)
```

D2C

Syntax

D2C(whole-number [, n])

Description

The D2C function converts the decimal representation of a number to its character representation.

Parameters

Parameter	Explanation
whole-number	The decimal representation of the number to be converted. It must be a whole number - that is, it must be a number that can be represented entirely in digits within the current setting of NUMERIC DIGITS. If n is omitted, whole-number must be non-negative.
n	The length of the result in characters. It must be non-negative. If n is specified, the result is sign-extended to the specified length. If the result will not fit in n characters, it is truncated on the left.

Example 1

The output of the following program fragment:

```
charval = d2c(97)
is:
charval = 'a'
```

Example 2

The output of the following program fragment:

```
charval = d2c(979899)
is:
charval = 'abc'
```

D2X

Syntax

D2X(whole-number [, n])

Description

The D2X function converts the decimal representation of a number to its hexadecimal representation.

Parameters

Parameter	Explanation
whole-number	The decimal representation of the number to be converted. It must be a whole number – that is, it must be a number that can be represented entirely in digits within the current setting of NUMERIC DIGITS. If n is omitted, whole-number must be non-negative.
n	The length of the result in characters. It must be non-negative. If n is specified, the result is sign-extended to the specified length. If the result will not fit in n characters, it is truncated on the left.

Example 1

The output of the following program fragment:

```
hexval = d2x(97)
      is:
hexval = '61'
```

Example 2

The output of the following program fragment:

```
bval = x2b(d2x(97))
      is:
bval = '01100001'
```

ERRORTEXT

Syntax

ERRORTEXT (n)

Description

The ERRORTEXT function returns the message text associated with the specified CM REXX error number.

Parameters

Parameter	Explanation
n	Is a number in the range 0-99. If n is not a currently defined CM REXX error, then ERRORTEXT returns a null string. If n is not within the valid range, then ERRORTEXT results in Error 40: Incorrect call to routine.

Example 1

The output of the following program fragment:

```
msg = errortext(11)
      is:
msg = 'Control stack full'
```

Example 2

The following program fragment illustrates the use of the special variable `rc` to retrieve the appropriate message text when a processing error occurs. When the `SYNTAX` condition is raised, the value of `rc` is the number of the error that raised the condition.

```
signal on syntax
a = 10
b = max(a, x)
say b
syntax:
say errortext(rc)
say 'detected at line' sigl
exit
```

The output is:

```
Bad arithmetic conversion detected at line 3
```

Note

The processing error occurs because the variable `x` used in the MAX function, is uninitialized and therefore has the value `X`. Arguments of MAX must be numeric.

FIND

Syntax

`FIND(string1, string2)`

Description

The FIND function searches a string of blank-delimited words for the first occurrence of another string of blank-delimited words.

Parameters

Parameter	Explanation
string1	The string to be searched.
string2	The search string.

Usage Notes

FIND returns the number of the first word in **string1** where a match is found. If no match is found, FIND returns 0.

For purposes of comparison, multiple blanks between words in either **string1** or **string2** are treated as a single blank.

FIND is included in CM REXX for compatibility with the VM and TSO/E implementations of REXX. It may not be available in other implementations and is not included in the standard language definition. Use WORDPOS to ensure portability of an application across all implementations of REXX.

Example 1

The output of the following program fragment:

```
x = find("How now brown cow", "brown cow")
is:
x = 3
```

Example 2

The output of the following program fragment:

```
y = find("Once upon a time", "a time")
is:
y = 3
```


Example 3

The following program fragment uses `FIND` to verify user response to a prompt; if the answer provided by the user does not match one of the words in the list, `FIND` returns 0.

```
list = 'REXX C FORTRAN LISP PL/I'  
say 'What language for this program?'  
pull lang  
if find(list, lang) = 0 then  
    say 'Language not available'
```

FORM

Syntax

FORM()

Description

The FORM function returns the current setting of NUMERIC FORM.

Usage Notes

The description of the NUMERIC instruction in the previous chapter, *Chapter 4: Instructions*, contains information on using NUMERIC FORM to control the precision and format of numbers used in the results of arithmetic operations, and the evaluation of arithmetic functions.

Example 1

In the following program fragment:

```
expform = form()
```

expform = 'SCIENTIFIC' if the default setting of NUMERIC FORM is in effect.

Example 2

The following program fragment ensures that NUMERIC FORM is set correctly for this application before proceeding with other operations.

```
if form() \= 'ENGINEERING' then  
  numeric form engineering
```

FORMAT

Syntax `FORMAT(num [, [before] [, [after] [, [expp] [, expt]]])`

Description The FORMAT function rounds and formats a number.

Parameters

Parameter	Explanation
num	Is the number to be formatted. If no additional arguments are specified, FORMAT simply rounds the number.
before	The number of places to the left of the decimal point (the integer portion) of the result. before must be a positive integer. If before is omitted, the number of places to the left of the decimal point is exactly the number contained in the result. If before is greater than the number of places to the left of the decimal in the result, the result is padded on the left with blanks. If before is less than the number of places to the left of the decimal in the result, Error 40 results.
after	The number of places to the right of the decimal point (the decimal portion) of the result. after can be a positive integer or zero. If after is omitted, the number of places to the right of the decimal point is exactly the number contained in the result. If after is greater than the number of decimal places in the result, the result is padded with zeros. If after is less than the number of decimal places in the result, the result is rounded to fit. If after is specified as 0, then num is rounded to the nearest integer.
expp expt	Used to override the current settings of NUMERIC DIGITS and NUMERIC FORM in the result of FORMAT.
expp	Specifies the number of digits to be used in the exponent portion of the result. expp must be a positive integer or zero. If expp is greater than the number of digits required for the exponent, it is padded on the left with zeros. If expp is less than the number of digits required for the exponent, Error 40 results. If expp is specified as 0, no exponent is supplied in the result, and zeros are added as necessary to express the result without exponential notation. If expp is non-zero and the exponent of the result is zero, then the result is padded on the right with expp +2 blanks.
expt	The trigger point for exponential notation. expt must be a positive integer or zero. If the number of places to the left of the decimal point in the result is greater than expt , the result is expressed exponentially. If the number of places to the right of the decimal in the result is greater than 2* expt , the result is expressed exponentially. If expt is specified as 0, the result is always expressed exponentially unless the exponent of the result is 0.

Usage Notes

FORMAT first rounds the number using the standard REXX rules that would be applied if the operation `num + 0` were performed. It then formats the number. By default, the number is formatted according to the current

settings of `NUMERIC DIGITS` and `NUMERIC FORM`. The last two arguments of `FORMAT` allow you to override these defaults.

Example 1

The output of the following program fragment:

```
x = format(12,5)
is:
x = ' 12'
```

Example 2

The following program fragment outputs a right-justified column of numbers.

```
numlist = '10 456 2 1034'
do i = 1 to words(numlist)
  say format(word(numlist,i),4)
end
```

The output is:

```
10
456
 2
1034
```

Example 3

The following program fragment outputs a decimal-aligned column of numbers with exactly two decimal places in each number.

```
numlist = '10.567 456 .2 1034.6 45.25'
do i=1 to words(numlist)
  say format(word,numlist,i),4,2)
end
```

The output is:

```
10.57
456.00
 0.20
1034.60
 45.25
```

Example 4

The following program fragment illustrates the effect of the exponent trigger point on the formatted results.

```
numlist = '10 120 10.123 9.12345 123.12345'  
do i = 1 to words(numlist)  
  say format(word(numlist,i),,,,2)  
end
```

The output is:

```
10  
1.2E+2  
10.123  
9.12345  
1.2312345E+2
```

Example 5

The following program fragment illustrates use of the exponent trigger point to over-ride the current setting of `NUMERIC DIGITS`.

```
numeric digits 3  
numlist = '10 100 1000 10000 100000'  
do i = 1 to words(numlist)  
  say format(word(numlist,i))  
end  
say ''  
do j = 1 to words(numlist)  
  say format(word(numlist,j),,,,5)  
end
```

The output is:

```
10  
100  
1.00E+3  
1.00E+4  
1.00E+5  
  
10  
100  
1000  
10000  
1.00E+5
```

Example 6

The following program fragment illustrates use of the `expp` argument of `format()`.

```
numeric digits 3
list = 0 1 2 3
num = 12345
do i = 1 to words(list)
  say format(num,,,word(list,i))
end
```

The output is:

```
12300
1.23E+4
1.23E+04
1.23E+004
```

FUZZ

Syntax

FUZZ ()

Description

The FUZZ function returns the current setting of NUMERIC FUZZ.

Usage Notes

The description of the NUMERIC instruction in the previous chapter, *Chapter 4: Instructions*, contains information on using NUMERIC FUZZ to control how many digits are ignored in a numeric comparison.

Example

In the following program fragment:

```
expfuzz = fuzz ()
```

expfuzz = 0 if the default setting of NUMERIC FUZZ is in effect.

GETCWD

Syntax GETCWD()

Description The GETCWD function returns the full path name of the current working directory.

Example 1

In the following program fragment:

```
dir = getcwd()
```

if the current directory is c:\progra~1\Hewlett-Packard\CM, then dir =
'c:\progra~1\Hewlett-Packard\CM'

Example 2

The following program fragment creates an output file name within the current working directory:

```
dir = getcwd()  
outfile = dir'\output.txt'
```


GETENV

Syntax `GETENV(string)`

Description The GETENV function returns the current setting of an environment variable.

Parameters

Parameter	Explanation
<code>string</code>	The name of the environment variable for which the current setting is to be returned. If the environment variable specified by string is not set, GETENV returns a null string.

Usage Note

It is recommended that the string argument be enclosed in quotes. Without the quotes, **string** is an un-initialized symbol. As long as the symbol remains un-initialized, GETENV behaves as expected since the value of the un-initialized symbol is the symbol in uppercase. If, however, an assignment statement sets the value of that symbol to something else, the GETENV function would attempt to determine the setting of the environment variable specified by the value assigned to **string**.

Example

In the following program fragment:

```
home = getenv('HOME')
```

`home` = the current value of `HOME`. This is the same value that would result from typing the DOS command:

```
set HOME
```

INDEX

Syntax INDEX(string1, string2 [, start])

Description The INDEX function searches a string for the first occurrence of another string.

Parameters

Parameter	Explanation
string1	The string to be searched.
string2	The search string.
start	The character position in string1 where the search begins. start must be a positive integer. If start is greater than the length of string1 , INDEX returns 0.

Usage Notes

INDEX is included in CM REXX for compatibility with the VM and TSO/E implementations of REXX. It may not be available in other implementations and it is not included in the standard language definition. Use POS to ensure portability of an application across all implementations of REXX.

INDEX returns the position of the first character in **string1** where a match is found. If no match is found, INDEX returns 0.

Example 1

The output of the following program fragment:

```
where = index('abcdef', 'c')  
  
is:  
  
where = 3
```

Example 2

The output of the following program fragment:

```
where = index('abrakadabra', 'a', 5)  
  
is:  
  
where = 6
```

Example 3

The following program fragment uses `INDEX` to verify user response to a prompt. If the answer provided by the user does not match one of the characters in the list, `INDEX` returns 0.

```
options = abcxyz
say 'Select a processing option'
pull which_option
if index(options, which_option) = 0 then
  call bad_option
else call got_it_right
```

INSERT

Syntax

INSERT(string1, string2 [, [n] [, [length][, pad]])

Description

The INSERT function inserts one string into another string.

Parameters

Parameter	Explanation
string1	The string to be inserted.
string2	The inserted string.
n	The character position in string2 after which insertion begins. n must be a non-negative number. If n is specified as 0, string1 is inserted before the first character of string2 . If n is omitted, the default value is 0.
length	The number of characters to be inserted. length must be a non-negative number. If string1 is shorter than length , it is padded on the right to the value of length before insertion. If n is greater than the length of string2 , string1 is also padded on the left before insertion. If length is 0, none of the characters in string1 are inserted. If length is omitted, the default is the length of string1 .
pad	Character used to pad string1 before insertion. If pad is omitted, the default pad character is a blank.

Example

This program fragment illustrates various combinations of the arguments to INSERT.

```
ins = 'scotty '  
string = 'beam me up now'  
say insert(ins, string)  
say insert(ins, string, length(string)+1)  
say insert(ins, string, 11)  
say insert(ins, string, 20)  
say insert(ins, string, 20, 0, '!')
```

The output is:

```
scotty beam me up now  
beam me up now scotty  
beam me up scotty now  
beam me up now scotty  
beam me up now!!!!!!
```

JUSTIFY

Syntax `JUSTIFY(string, length [, pad])`

Description The JUSTIFY function adds pad characters between words in a string of blank-delimited words to justify both margins.

Parameters

Parameter	Explanation
string	String of blank-delimited words.
length	length is the length of the string returned by the function.
pad	Character used to pad string. If pad is omitted, the default pad character is a blank.

Usage Note

JUSTIFY is included in CM REXX for compatibility with the VM and TSO/E implementations of REXX. It may not be available in other implementations and is not included in the standard language definition. Use POS to ensure portability of an application across all implementations of REXX.

Example

The output of the following program fragment:

```
str = 'To be or not to be'
outstr = justify(str, 25)

is:

outstr = 'To  be  or not to be'
```

LASTPOS

Syntax

`LASTPOS(string1, string2 [, start])`

Description

The LASTPOS function finds the right-most occurrence of one string within another string. It scans string2 from right to left looking for string1.

Parameters

Parameter	Explanation
string1	Search string.
string2	String to be searched.
start	Character position within string2 where the backward search begins. start must be a positive integer. If start is greater than the length of string2 , it defaults to the length of string2 . If start is omitted, the default is the length of string2 .

Usage Notes

It returns the character position of the right-most occurrence of **string1** in **string2**. If **string1** is not found in **string2**, then LASTPOS returns 0.

Example 1

The output of the following program fragment:

```
x = lastpos('a', 'abrakadabra')
is:
x = 11
```

Example 2

The output of the following program fragment:

```
x = lastpos('a', 'abrakadabra', 7)
is:
x = 6
```

Example 3

In the following program fragment, `LASTPOS` returns 0 if there is only one entry in `product_list` (no blanks in the list) or non-zero if there is more than one entry in the list.

```
product_list = 'RMS RMP ROMS RCS'
if lastpos(' ', product_list) = 0
    then say 'Only one CM product installed'
    else say 'Several CM products installed'
```

The output is

```
Several CM products installed
```

LEFT

Syntax `LEFT(string, n [, pad])`

Description The LEFT function returns the left-most characters in a string.

Parameters

Parameter	Explanation
string	The original string.
n	The number of characters to be returned. n must be non-negative. If n is zero, the LEFT function returns a null string. If n is greater than the length of string, the value returned by LEFT is padded on the right to the length of n .
pad	Character used to pad the result. If pad is omitted, the default is a blank character.

Example 1

The output of the following program fragment:

```
x = left('abcdefg', 3)
is:
x = 'abc'
```

Example 2

The output of the following program fragment:

```
alphabet = left('abc', 26)
is:
alphabet = 'abc'
```

Example 3

The output of the following program fragment:

```
alphabet = left('abc', 6, '.')
The output is:
alphabet = 'abc...'
```


Example 4

The following program fragment processes an input file by selecting data only from those lines that do not begin with a comment character (#).

```
input = 'mydata.txt'
j = 1
do lines(input)
  line = linein(input)
  if left(line, 1) \= '#' then do
    parse var line num.j descr.j .

    j = j + 1
  end
end
```

Example 5

The following program fragment uses the LEFT and RIGHT functions to format output data.

```
line.1 = 'Jan East 1500 West 975 Total $ 2475'
line.2 = 'Feb East 24660 West 975 Total $34635'
line.3 = 'Mar East 800 West 8500 Total $ 9300'
:
:
do i = 1 to 12
  say left(line.i, 3) right(line.i, 6)
end
```

The output is:

```
Jan $ 2475
Feb $34635
Mar $ 9300
:
:
```

LENGTH

Syntax `LENGTH(string)`

Description The LENGTH function determines the number of characters in a string.

Parameters

Parameter	Explanation
string	The string for which the length is to be determined.

Example 1

The output of the following program fragment:

```
x = length('Hello')
```

is:

```
x = 5
```

Example 2

The following program fragment validates user input based on the number of characters in that input.

```
say 'Enter part number'  
pull reply  
if length(reply) \= 4 then do  
  say 'Invalid part number:' reply  
  say 'Part numbers have exactly 4 digits'  
end
```

LINEIN

Syntax

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

Description

The function reads a line from a character input stream. It can also be used to set the read position in a persistent input stream. Use `LINEIN` for input streams that have normal line-end terminators (usually CR/LF).

Parameters

Parameter	Explanation
name	The name of the character input stream. This can be a persistent stream such as a disk file or a transient stream such as STDIN or a pipe (including a named pipe). If name is omitted, the default is STDIN.
line	Specifies an explicit read position in a persistent input stream such as a disk file. It must be a positive integer and must be within the bounds of the input stream specified. If line is omitted, the default is the current read position. line cannot be specified for a transient input stream.
count	Specifies the number of lines to be read. count must be 0 or 1. If count is omitted, the default is 1. If count is specified as 0, then the read position is set to the beginning of line, and the function returns a null string.

Usage Notes

If a complete line is not available in the stream, the program can wait until the line is complete. If it is not possible for a line to be completed, the function returns all available characters and raises the NOTREADY condition. The built-in function `STREAM` can be used to determine the state of a character stream.

Use `LINEIN` to read complete lines that have normal line-end terminators. This means that it is important to know the kind of data contained in a file that you read using `LINEIN`. Trying to read a large file that lacks normal line-end terminators (such as a binary file) using `LINEIN` can result in unexpected and undesirable results. Use `CHARIN` to read less than a complete line, or to read lines that do not have normal line-end terminators.

Use of an I/O function such as `LINEIN` can leave a persistent input stream in an open state. Thus, it may be necessary to close it using `LINEOUT`, `CHAROUT`, or `STREAM` before performing subsequent read or write operations.

Example 1

The following example reads one line from the current read position and assigns that value to the variable `emp_record`.

```
emp_record = linein('personnel.txt')
```

Example 2

The following program fragment displays a prompt to the user. It then pauses until data is available on STDIN (in this case, characters typed at the keyboard); `LINEIN` returns everything that was typed at the keyboard before `ENTER` was pressed and assigns that value to the variable `num`; a host command then prints a file.

```
say 'Enter report number'  
num = linein()  
address cmd 'print report.'num
```

Example 3

The following program fragment processes all lines in an input file, one line at a time.

```
infile = 'foo.txt'  
do i = 1 while lines(infile) > 0  
  line.i = linein(infile)  
end
```

LINEOUT

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

Description The LINEOUT function writes a line to a character output stream and returns the number of lines remaining in the stream after the write has been attempted.

Parameters

Parameter	Explanation
name	The name of the character output stream. This can be a persistent stream such as a disk file or a transient stream such as STDOUT or a pipe (including a named pipe). If name is omitted, the default is STDOUT.
string	The character string to be written. If name is a persistent stream, then string can be omitted. In this case, one of the following actions is taken: If line is specified, LINEOUT resets the write position to the start value, and the function returns 0. If line is omitted, LINEOUT closes the output stream, and the function returns 0.
line	Specifies an explicit write position. It must be a positive integer and must be within the bounds of the output stream specified. If line is omitted, the default is the current write position. line may not be specified for a transient output stream.

Example 1

The following program fragment writes the string specified by the variable `emp_data` to the file `personnel.txt`. `rc` is normally 0.

```
emp_data = 'DEV003 Smith Joe Software Engineer'  
rc = lineout('personnel.txt', emp_data)  
if rc \= 0 then  
    say 'Error in writing to personnel file'
```

Example 2

The following program fragment:

```
out_rc = lineout(, 'Hello world')
```

writes "Hello world" to STDOUT, usually the terminal. `out_rc` is normally 0.

Example 3

The following program fragment writes the lines specified by the compound variables `emp.<n>` to the file `personnel.txt`. After the last line is written, it closes the file. Note the use of `CALL` to invoke the function.

```
outfile = 'personnel.txt'
emp.0 = 57
emp.1 = 'DEV003 Smith Joe Software Engineer'
emp.2 = 'DEV004 Jones Anne AI Specialist'
:
:
do i = 1 to emp.0
  call lineout outfile, emp.i
end
call lineout outfile
```

LINES

Syntax LINES ([name])

Description The LINES function returns the number of complete lines remaining in a character input stream.

Parameters

Parameter	Explanation
name	The name of the character input stream. This can be a persistent stream such as a disk file or a transient stream such as STDIN or a pipe (including a named pipe). If name is omitted, the default is STDIN.

Example 1

In the following program fragment:

```
count = lines('foo.txt')
```

count is set to the number of lines in the disk file named `foo.txt`.

Example 2

The following program fragment tests for the existence of a file. If the file exists (the value of the LINES function is greater than zero), the file is deleted before proceeding.

```
if lines('foo.txt') > 0 then  
  address cmd 'erase foo.txt'
```

Example 3

The following program named `anydata` gives different results depending on whether or not data is waiting.

```
if lines() then say 'Data available'  
else say 'No data'
```

When you run this program by typing:

```
anydata
```

the output is:

```
No data
```

When you run this program by typing:

```
echo 'Hello world' | anydata
```

the output is:

```
Data available
```


LOWER

Syntax `LOWER(string)`

Description The LOWER function converts characters in a string to lowercase.

Parameters

Parameter	Explanation
string	The string of characters to be converted. string can be upper-, lower-, or mixed-case.

Example 1

The output of the following program fragment:

```
low = lower('ABCD')
```

is:

```
low = 'abcd'
```

Example 2

The following program fragment converts user input to lowercase before validating the input.

```
say 'Enter authorization'  
parse pull reply  
if wordpos(lower(reply), auth_list) \= 0 then  
  call run_prog  
else say 'Sorry, not authorized'
```

Example 3

The following program is functionally equivalent to the previous example but ensures that `reply` is taken from the terminal (STDIN) rather than from data that might be on the program stack.

```
say 'Enter authorization'  
reply = lower(linein())  
if wordpos(reply, auth_list) \= 0 then  
  call run_prog  
else say 'Sorry, not authorized'
```

MAX

Syntax `MAX(number [, number] ...)`

Description The MAX function returns the largest number in a list of numbers.

Parameters

Parameter	Explanation
number	Any valid number.

Example

The output of the following program fragment:

```
x = max(10, 12, 9)
```

is:

```
x = 12
```

MIN

Syntax `MIN(number [, number] ...)`

Description The MIN function returns the smallest number in a list of numbers.

Parameters

Parameter	Explanation
number	Any valid number.

Example 1

The output of the following program fragment:

```
x = min(10, 12, 9)
```

is:

```
x = 9
```

Example 2

The following program fragment uses MIN to get the length of the shortest word in a string.

```
list = 'the a an'  
shortest = length(word(list, 1))  
do while list \= ''  
  parse var list next list  
  shortest = min(shortest, length(next))  
end  
say shortest
```

The output is:

```
1
```

OVERLAY

Syntax `OVERLAY(string1, string2 [, [n][, [length] [, pad]])`

Description The OVERLAY function overlays one string with characters from another string.

Parameters

Parameter	Explanation
string1	This is the overlay string, that is, the string that supplies characters for the overlay operation.
string2	The original string in which characters are to be replaced by characters from string1 .
n	The character position in string2 where the overlay begins. n must be a positive integer. If n is greater than the length of string2 , string1 is padded on the left before the overlay is performed. If n is omitted, the default value is 1.
length	Number of characters to overlay. length must be non-negative. If length is greater than the number of characters in string1 , string1 is padded on the right before the overlay is performed. If length is less than the number of characters in string1 , string1 is truncated from the right before the overlay is performed. If length is omitted, the default value is the length of string1 .
pad	Character to be used for padding string1 . If pad is omitted, the default is a blank character.

Example 1

The output of the following program fragment:

```
str = overlay('old', 'new data')
is:
str = 'old data'
```

Example 2

The output of the following program fragment:

```
str = overlay('old', 'Some new data', 6)
is:
str = 'Some old data'
```

Example 3

The output of the following program fragment:

```
str = overlay('change', 'New data', 12, 8, '*')
is:
str = 'New data***change**'
```

Example 4

The following program fragment takes a template reply message and uses `OVERLAY` to replace a placeholder string with the current date before mailing the message.

```
parse arg inquirer
auto_reply = 'template.txt'
mail_msg = 'msg.txt'
d = "Insert today's date here"
do lines(auto_reply)
  line = linein(auto_reply)
  if wordpos(d, line) \= 0 then
line=overlay(date(),line,pos(d,line),(length(d))
  call lineout mail_msg, line
end
call lineout mail_msg /*be sure file is closed */
```

POPEN

Syntax POPEN(*command* [, *option*])

Description The POPEN function executes a host command and places the results on the REXX program stack. It returns the completion code of the host command.

Parameters

Parameter	Explanation
<i>command</i>	Any host command that is valid in the Bourne shell.
<i>option</i>	Indicates whether command output should be placed on the stack in FIFO or LIFO order. 'P' specifies LIFO order; 'Q' specifies FIFO order. If option is omitted, the default value is 'Q'.

Usage Note

POPEN redirects STDOUT to the program stack. Use POPEN to:

- Capture the output of a host command for subsequent processing.
- Execute any host command that may write to STDOUT when you do not wish that output to appear on the terminal screen.

Example 1

The following program invokes the UNIX test command to check for existence of a file. If the file exists, **test** sets a completion code of 0 and therefore **state** = 0. If the file does not exist, **test** sets a completion code of 1 and therefore **state** = 1.

```
state = popen("test -f myfile")
```

Example 2

The following program fragment processes all files in the current directory with a date/time stamp matching the current month.

```
x = 5  
rc = popen("ls -l")  
if rc \= 0 then call error1  
do queued()  
parse pull nextfile
```

```
if word(nextfile, x) = left(date(m),3) then
call prog2
end
```

Note that the output of "ls" is system-dependent. This example is for SunOS.
Change value of "x" for other systems as needed.

POS

Syntax `POS(string1, string2 [, start])`

Description The POS function searches a string for the left-most occurrence of another string.

Parameters

Parameter	Explanation
<code>string1</code>	The search string.
<code>string2</code>	The string to be searched.
<code>start</code>	The character position in string2 where the search begins. start must be a positive integer. If start is greater than the length of string2 , POS returns 0.

Usage Note

POS returns the position of the left-most character in **string2** where a match is found. If no match is found, POS returns 0.

Example 1

The output of the following program fragment:

```
where = pos('c', 'abcdef')  
  
is:  
  
where = 3
```

Example 2

The output of the following program fragment:

```
where = pos('a', 'abrakadabra', 5)  
  
is:  
  
where = 6
```


Example 3

The following program fragment uses `POS` to verify user response to a prompt; if the answer provided by the user does not match one of the characters in the list, `POS` returns 0.

```
options = abcxyz
say 'Select a processing option'
pull which_option
if pos(which_option, options) = 0 then
    call bad_option
else call value which_option
```

PUTENV

Syntax `PUTENV(string)`

Description The PUTENV function sets the value of an environment variable.

Parameters

Parameter	Explanation
string	A command to set the value of an environment variable. The command is of the form VARIABLE=value .

Usage Notes

Blanks are not permitted around the equal sign.

Use PUTENV to set or modify the value of an environment variable used by the process in which the CM REXX program is running. Environment variables set by PUTENV are not retained after the CM REXX program terminates.

Example

The following program fragment:

```
rc = putenv('MYVAR=FOO')
```

sets the MYVAR environment variable. If PUTENV executes successfully, the value of rc is 0. If an error occurs, the value of rc is non-zero.

QUEUED

Syntax

QUEUED ()

Description

The QUEUED function returns the number of lines remaining on the CM REXX external data queue.

Example

The following program processes every line remaining on the CM REXX external data queue, based on some pre-determined criterion.

```
do queued()
  pull nextone
  if word(nextone, 3) > checkit then call bigger
  else call smaller
end
```

RANDOM

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

Description The RANDOM function returns a quasi-random, non-negative whole number.

Parameters

Parameter	Explanation
min	The lower value of the range. min must be non-negative. If min is omitted, the default is 0.
max	The upper value of the range. max must be non-negative. If max is omitted, the default is 999.
seed	An initial seed value that can be used to create a repeatable series of results. seed must be a whole number. If seed is omitted, the default is an arbitrary value, which can be time-dependent.

Usage Note

The magnitude of the range specified cannot exceed 100000. Specifically, the following must be true:

```
max - min <= 100000
```

Example 1

The output of the following program fragment:

```
x = random()
```

could be:

```
x = 983
```

Example 2

The output of the following program fragment:

```
x = random(9)
```

could be:

```
x = 2
```

Example 3

The following program fragment generates a random number for use as the extension on a temporary file required by the program.

```
ext = random()  
tmpfile = '\\tmp\\thisprog.' + ext
```

REVERSE

Syntax `REVERSE(string)`

Description The REVERSE function reverses the characters in a string.

Parameters

Parameter	Explanation
string	The original string in which the characters are to be reversed.

Example 1

The output of the following program fragment:

```
str = reverse('string')
```

is:

```
str = 'gnirts'
```

Example 2

The output of the following program fragment:

```
time = reverse('noon ')
```

is:

```
time = ' noon'
```

RIGHT

Syntax `RIGHT(string, n [, pad])`

Description The RIGHT function returns the right-most characters in a string.

Parameters

Parameter	Explanation
string	The original string.
n	The number of characters to be returned. n must be non-negative. If n is zero, the RIGHT function returns a null string. If n is greater than the length of string, the value returned by RIGHT is padded on the left to the length of n .
pad	The character used to pad the result. If pad is omitted, the default is a blank character.

Example 1

The output of the following program fragment:

```
x = right('abcdefg', 3)
is:
x = 'efg'
```

Example 2

The output of the following program fragment:

```
alphabet = right('xyz', 26)
is:
alphabet = '      xyz'
```

Example 3

The output of the following program fragment:

```
alphabet = right('xyz', 6, '.')
is:
alphabet = '...xyz'
```

Example 4

The following program fragment removes 6-character sequence numbers from the beginning of each line of a file.

```
input = 'foo.txt'
output = 'bar.txt'
do lines(input)
  line = linein(input)
  line = right(line, length(line)-6)
  call lineout output, line
end
call lineout output
```

Example 5

The following program fragment uses the `LEFT` and `RIGHT` functions to format output data.

```
line.1 = 'Jan East 1500 West 975 Total $ 2475'
line.2 = 'Feb East 24660 West 975 Total $34635'
line.3 = 'Mar East 800 West 8500 Total $ 9300'
:
:
do i = 1 to 12
  say left(line.i, 3) right(line.i, 6)
end
```

The output is:

```
Jan $ 2475
Feb $34635
Mar $ 9300
:
:
```


SIGN

Syntax `SIGN(number)`

Description The SIGN function returns a value that indicates the sign of a number.

Parameters

Parameter	Explanation
number	The number for which the sign is to be determined. If number is negative, then SIGN returns -1. If number is zero, then SIGN returns 0. If number is positive, then SIGN returns 1.

Example 1

The output of the following program fragment:

```
x = sign(10)
```

is:

```
x = 1
```

Example 2

The output of the following program fragment raises 2 to the power chosen by the user. It does not permit negative or zero exponents.

```
say 'Enter exponent'  
pull power  
if sign(power) > 0 then say 2**power  
else say power 'invalid here'
```

SOURCELINE

Syntax SOURCELINE ([n])

Description The SOURCELINE function returns either the number of lines in the current program or the contents of the specified line.

Parameters

Parameter	Explanation
n	A line number within the range of the current program. n must be positive and cannot exceed the line number of the last line in the program. When n is specified, SOURCELINE returns the contents of the n th line in the program. If n is omitted, SOURCELINE returns the line number of the last line in the program. If no source lines are available (as in the case of a compiled program), SOURCELINE returns 0.

Example 1

In the following program fragment:

```
prog_length = sourceline()
```

if the current program contains 50 lines, then

```
prog_length = 50
```

Example 2

The following program fragment illustrates the use of SOURCELINE to identify errors occurring during program execution.

```
call on error name uhoh
parse arg program_name
address CMD program_name
:
:
exit
uhoh:
parse value sourceline(sig1) with 'CMD' failed
say 'Host command failed'
interpret 'say' failed "'not found in PATH'"
return
```

SPACE

Syntax

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

Description

The SPACE function reformats a string of blank-delimited words such that the specified number of pad characters appears between each word.

Parameters

Parameter	Explanation
string	The string of blank-delimited words to be formatted.
n	The number of pad characters to appear between each word in the result. n must be non-negative. If n is specified as zero, all blanks in string are removed. If n is omitted, the default value is 1.
pad	Character used between each word in the result. If pad is omitted, the default pad character is a blank.

Example 1

The output of the following program fragment:

```
x = space('Good morning')
is:
x = 'Good morning'
```

Example 2

The following program fragment creates a header line for a report.

```
str = date time userid status
header = space(str, 6, '-')
```

The header line looks like:

```
DATE-----TIME-----USERID-----STATUS
```

Example 3

The following program uses `SPACE` in conjunction with `TRANSLATE` to remove characters from a string.

```
string = 'work group'
string = translate(string, 'o', ' ou')
string = space(string, 0)
string = translate(string, 'o', ' o')
string = space(string, 0)
say string
```

The output is:

```
wrkgrp
```

STREAM

Syntax

STREAM(name [, operation[, strmcmd]])

Description

The STREAM function is used to determine the state of a stream, or to perform an operation on a stream and return the result.

Parameters

Parameter	Explanation
name	The name of the stream of interest.
operation	<p>Describes the action to be carried out. If operation is omitted, the default value is S. If operation is specified, it must have one of the following values: C, D, or S.</p> <p>C (command) The command to execute on this stream as specified by the strmcmd argument.</p> <p>D (description) Descriptive string associated with the current state of the stream; the descriptive strings are available only when the state of the stream is READY. strmcmd must not be specified.</p> <p>S (state) The current state of the specified stream. strmcmd must not be specified; the value returned, if you specify S, is one of the following:</p> <p>ERROR An erroneous operation has been attempted on the stream.</p> <p>NOTREADY Normal input or output operations would raise the NOTREADY condition.</p> <p>READY The stream is ready for normal input or output operations.</p> <p>UNKNOWN The state of the stream cannot be determined.</p>

Parameter	Explanation
strmcmd	A command to be executed on the stream. strmcmd must be enclosed in quotes and must be one of the following:
open	Open the stream for input or output operations; the function returns the state of the stream.
close	Close the stream for input or output operations; the function returns the state of the stream.
delete	Remove the file; the function returns a null string.
query exists	Test for existence of the stream; the function returns the name of the stream, if it exists; otherwise it returns a null string.
query size	Determine the number of characters in the file; the function returns the number of characters.
query datetime	Retrieve the date/time stamp of the file; the function returns the information in the form mm-dd-yy hh:mm:ss.
seek offset	Position the file for the next input or output operation; offset must be a positive integer preceded by one of the following characters:
	= Offset is from the beginning of the file.
	< Offset is from the end of the file.
	+ Offset is forward from the current position.
	- Offset is backward from the current position.

Example

The following program fragment illustrates the use of the `STREAM` function:

```
strm = 'sales.txt'
state = stream(strm, 'c', 'query exists')
if state \= '' then
  if stream(strm, 'c', 'open') \= 'READY' then
    say 'error opening file' strm
  else
    :
    /* Process the file..      */
    :
    :
```

STRIP

Syntax

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

Description

The STRIP function removes leading, trailing, or both leading and trailing characters from a string.

Parameters

Parameter	Explanation
string	The string from which characters are to be removed.
option	Specifies whether leading, trailing, or both leading and trailing characters are to be removed. option can be any string beginning with the character L , T , or B , in any case. If the first character of option is L , only leading characters are removed. If the first character of option is T , only trailing characters are removed. If the first character of option is B , both leading and trailing characters are removed. If option begins with any other character, Error 40 results. If option is omitted, the default is B .
char	Character to be removed from string . If specified, char can be only one character. If char is omitted, the default is a blank.

Example 1

The output of the following program fragment:

```
x = strip('  Gypsy Rose  ')  
  
is:  
  
x = 'Gypsy Rose'
```

Example 2

The output of the following program fragment:

```
x = strip('000123', '1')  
  
is:  
  
x = '123'
```

Example 3

The output of the following program fragment:

```
x = strip('In retrospect....', 'Trail', '.')
is:
x = 'In retrospect'
```

Example 4

The following program fragment removes leading and trailing blanks from a value to be used as the tail in referencing a compound symbol.

```
pfile = 'params.txt'
do lines(pfile)
  parse value linein(pfile) with arg1 arg2 prog
  prog = strip(upper(prog))
  interpret 'call subr.'prog arg1',' arg2
end
```


SUBSTR

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

Description The SUBSTR function returns a sub-string of a string.

Parameters

Parameter	Explanation
string	The string from which the sub-string is to be extracted.
n	Character position within string where the sub-string begins. n must be positive. If n is greater than the length of string , then only pad characters are returned.
length	The length of the sub-string to be returned. length must be non-negative. If length is greater than the number of characters from n to the end of string , then the result is padded on the right. If length is specified as 0, then the null string is returned. If length is omitted, the result includes all characters from n to the end of string .
pad	The pad character to be used. If pad is omitted, the default is a blank character.

Example 1

The output of the following program fragment:

```
x = substr('CM REXX', 7)
is:
x = 'REXX'
```

Example 2

The output of the following program fragment:

```
herbs = 'parsley sage rosemary thyme'
herb2 = substr(herbs, 9, 4)
is:
herb2 = 'sage'
```

Example 3

The output of the following program fragment:

```
today = substr(date(u), 4, 2)
```

is:

```
today = '18'
```

on the 18th day of any month.

Example 4

The following program fragment extracts a sub-string from a series of numbers, and pads the short ones with zeroes.

```
numlist = '14 144 4114 41'  
do i = 1 to words(numlist)  
  x = substr(word(numlist, i), 2, 3, 0)  
  say x  
end
```

The output is:

```
400  
440  
114  
100
```

SUBWORD

Syntax `SUBWORD(string, n [, length])`

Description The SUBWORD function returns a sub-string from a string of blank-delimited words.

Parameters

Parameter	Explanation
string	The string from which the sub-string is to be extracted.
n	The number of the word within string where the sub-string begins. n must be positive. If n is greater than the number of words in string, then the null string is returned.
length	Number of words to be returned. length must be non-negative. If length is specified as 0, then the null string is returned. If length is omitted, the result includes all remaining words in string.

Example 1

The output of the following program fragment:

```
n = subword('over the rainbow', 3)
is:
n = 'rainbow'
```

Example 2

The output of the following program fragment:

```
days = 'Mon Tue Wed Thur Fri Sat Sun'
weekend = subword(days, 6)
is:
weekend = 'Sat Sun'
```

SYMBOL

Syntax SYMBOL (name)

Description The SYMBOL function returns the status of a symbol.

Parameters

Parameter	Explanation
name	Specifies the symbol name for which status is to be determined. name is, itself, a symbol - that is, normal conversion to uppercase and substitution of assigned values occurs before the SYMBOL function is evaluated. It is therefore recommended that name be enclosed in quotes to prevent substitution and ensure that the status returned is for the symbol intended.

Return Values

Return Value	Explanation
BAD	Indicates that name is not a valid REXX symbol.
VAR	Indicates that name is a variable (a symbol to which a value has been assigned).
LIT	Indicates that name is a literal; this could be either a constant symbol or a symbol to which no value has yet been assigned.

Example 1

The following program fragment illustrates the various results from the `SYMBOL` function.

```
a = 14
b = 3
c. = 0
c.3 = 'hello'
say symbol(a)
say symbol('a')
say symbol('c.1')
say symbol('c.b')
say symbol('d')
say symbol('%')
```

The output is:

```
LIT    /* after substitution, is symbol(14) */
VAR    /* no substitution          */
VAR
VAR
LIT    /* no value yet assigned      */
BAD    /* "%" not permitted as symbol name */
```

Example 2

The following program fragment illustrates using `SYMBOL` instead of setting a flag to test for successful processing.

```
drop testvar
do i = 1 to lines('in_file')
  line = linein('in_file')
  if word(line, 5) \= 'temp' then
    testvar = word(line, 5)
  end
if symbol('testvar') \= 'LIT' then
  say 'Good data'
  else say 'All temps'
```

TIME

Syntax

```
TIME([out_option [, time_string, in_option]])
```

Description

The TIME function returns the current time of day, or converts times from one format to another. The second and third arguments of TIME provide support for converting time formats. Time format conversion permits arithmetic operations to be performed on times of any format.

Parameters

Parameter	Explanation
out_option	Specifies the format in which the time is returned. If out_option is omitted, the format returned is: hh:mm:ss. The valid format values for out_option are: C, E, H, L, M, N, R, S .
C (civil)	The time in civil format - hh:mmxx . The value of hh (hours) is between 1 and 12, without leading zeros. The value of mm (minutes) reflects the current minute. The value of xx is either am or pm , to indicate the midnight-to-noon or noon-to-midnight period, respectively.
E (elapsed)	The number of seconds since the elapsed time clock was started or reset. The format is sssss , without leading zeros or blanks. The first execution of TIME(E) starts the elapsed time clock and returns a value of 0.
H (hours)	The number of complete hours since midnight. The format is hh , without leading zeros or blanks. In the case of the period from midnight to 1:00, the value returned is 0.
L (long)	Extended time. The format is hh:mm:ss.uuuuuu . Hours, minutes, and seconds conform to the rules for the normal format. uuuuuu represents fractional seconds, given in microseconds. Fractional seconds are not available in some implementations. In these cases, TIME(L) returns the same value as TIME(N).
M (minutes)	The number of complete minutes since midnight. The format is mmmm , without leading zeros or blanks. In the case of the period from 12:00 midnight to 12:01 A.M., the value returned is 0.
N (normal)	The time of day using the 24-hour clock. The format is hh:mm:ss . The value of hh is from 00 through 23, with leading zeros. The value of mm and of ss is from 00 through 59, with leading zeros. Fractional seconds are ignored. This is the default result of TIME when no option is specified.
R (reset)	The number of seconds since the elapsed time clock was started or reset. The format is sssss , without leading zeros or blanks. In addition to returning elapsed time, TIME(R) resets the elapsed time clock.
S (seconds)	The number of complete seconds since midnight. The format is sssss , without leading zeros or blanks. In the case of the period from 12:00 midnight to 12:00:01, the value returned is 0.
time_string	Specifies the time to be converted. Time_string must be in one of the time formats

Parameter	Explanation
	described above. It may be a literal string, a variable reference, or an expression that evaluates to a time.
in_option	Specifies the format of time_string and must be one of the time formats described above. It should not be (E) Elapsed or (R) Reset.

Example 1

The output of the following program fragment:

```
now = time()
```

could be:

```
now = '10:30:15'
```

Example 2

The output of the following program fragment:

```
cnow = time('c')
```

could be:

```
cnow = '10:30am'
```

Example 3

The following program fragment measures the elapsed time required to run specified programs.

```
do forever
  say 'Enter program name or "Q"'
  parse pull prog
  if upper(prog) = 'Q' then leave
  call time('r')
  address cmd prog
  prog_time = time('e')
  say 'Time to run' prog ':' prog_time
end
exit
```

Example 4

The output of the following program fragment:

```
now = time('c', '17:17:00', 'n')
```

could be:

```
now = '5:17 pm'
```

Example 5

If it is currently 4:40 pm, the output of the following program fragment:

```
plus45 = time('c', time('m') + 45, 'm')
```

could be:

```
plus45=5:25 pm)
```


TRACE

Syntax `TRACE([option])`

Description The TRACE function returns the current setting of TRACE. It can also be used to change the TRACE setting.

Parameters

Parameter	Explanation
option	One of the valid TRACE settings as described in <i>Chapter 4: Instructions</i> . Valid trace settings are A, C, E, F, I, L, N, O, R .

Example 1

The output of the following program fragment:

```
setting = trace()
```

could be:

```
setting = 'N'
```

Example 2

The following program fragment uses the TRACE function both to capture the initial TRACE setting and to change the setting prior to calling a subroutine; after the subroutine returns, the TRACE instruction restores the TRACE setting to its original value.

```
set1 = trace('o')  
call subr  
trace value set1
```

TRANSLATE

Syntax

```
TRANSLATE(string [, [out_tbl] [, [in_tbl] [, pad]])
```

Description

The TRANSLATE function translates the characters in a string according to the specified translation tables.

Parameters

Parameter	Explanation
string	The string to be translated. Each character in string is looked up in in_tbl . If the character is found in in_tbl , the position in in_tbl at which it was found is used as an index into out_tbl . The character at that position in out_tbl is substituted for the character that was looked up, in the result string returned by the TRANSLATE function. If the character is not found in in_tbl , the character that was looked up is appended to the result string returned by the TRANSLATE function. If neither in_tbl nor out_tbl are specified, the TRANSLATE function returns string in uppercase.
out_tbl	The set of characters used in the result string. The default value is the null string. out_tbl is padded with pad or truncated so that out_tbl and in_tbl are the same length.
in_tbl	The set of characters from the original string that are to be translated in the result. The default value is XRANGE('00'x, 'FF'x).
pad	The character used to pad out_tbl , if necessary. If pad is omitted, the default is a blank.

Example 1

The output of the following program fragment:

```
upper_str = translate('Hello')
```

is:

```
upper_str = 'HELLO'
```

This is a fully portable equivalent to the CM REXX UPPER function, which may not be available in other REXX implementations.

Example 2

The following program fragment converts a string to lowercase. This is a fully portable equivalent to the CM REXX `LOWER` function, which may not be available in other REXX implementations.

```
parse arg string
uppers = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lowers = 'abcdefghijklmnopqrstuvwxyz'
lstring = translate(string, lowers, uppers)
```

Example 3

This example shows how to use `TRANSLATE` to reorder the characters in an input string.

```
intab = 'abcdefgh'
pattern = 'ef/gh/abcd'
reorder = translate(pattern, '19940704',intab)
```

The output is:

```
reorder = '07/04/1994'
```

TRUNC

Syntax

`TRUNC (number [, n])`

Description

The TRUNC function returns the integer portion of a number and, optionally, a specified number of decimal places.

Parameters

Parameter	Explanation
number	The numeric value to be truncated.
n	The number of decimal positions in the result. n must be non-negative. If n is omitted, the default value is 0.

Example 1

The output of the following program fragment:

```
x = trunc(3.1416)
```

is:

```
x = 3
```

Example 2

The output of the following program fragment:

```
y = trunc(3.1416, 2)
```

is:

```
y = 3.14
```

Example 3

The output of the following program fragment:

```
z = trunc(3.14, 3)
```

is:

```
z = 3.140
```

UPPER

Syntax `UPPER(string)`

Description The UPPER function converts characters in a string to uppercase.

Parameters

Parameter	Explanation
string	The string of characters to be converted. string can be upper-, lower-, or mixed-case.

Example 1

The output of the following program fragment:

```
up = upper('abcd')
```

is:

```
up = 'ABCD'
```

Example 2

The output of the following program fragment:

```
up = upper('Hello world')
```

is:

```
up = 'HELLO WORLD'
```

Example 3

The following program fragment ensures that user input is in uppercase for validation while also insuring that `reply` is taken from the terminal (STDIN) rather than from data that might be on the program stack.

```
say 'Enter authorization'  
reply = upper(linein())  
if wordpos(reply, auth_list) \= 0 then  
  call run_prog  
else say 'Sorry, not authorized'
```

USERID

Syntax

USERID()

Description

The USERID function returns the userid of the user currently logged on to the computer. It is identical to the CUSERID built-in function.

Example

The output of the following program fragment displays the User ID of the individual running the program.

```
say userid()
```

VALUE

Syntax VALUE (name)

Description The VALUE function returns the value of a symbol.

Parameters

Parameters	Explanation
name	name specifies the symbol name for which status is to be determined. name is, itself, a symbol. Normal conversion to uppercase and substitution of assigned values occurs before the VALUE function is evaluated. It is therefore recommended that name be enclosed in quotes to prevent substitution and ensure that the status returned is for the symbol intended.

Example 1

The output of the following program:

```
x = 10
say value('x')

is:

10
```

Example 2

The output of the following program fragment:

```
x = 10
y = 'x'
say value(y)

is:

10
```

Example 3

This example results in *Error 31: Name starts with number or '.',* because the value of x (10) is substituted before the VALUE function is evaluated.

```
x = 10
say value(x)
```

Example 4

The output of the following program fragment:

```
x = qqg
qqg = 10
y.10 = 'hello'
y.x = 'goodbye'
say value('y.x')
say value(y.x)
say value('y.' || x)
```

is:

```
goodbye
GOODBYE
hello
```


VERIFY

Syntax `VERIFY(string, char_list [, [option]`
 `[, start]])`

Description The VERIFY function verifies whether or not a string is composed only of characters in a specified character list.

Parameters

Parameters	Explanation
string	Is the string to be verified.
char_list	Is the list of acceptable characters.
option	Controls whether the function verifies the presence or absence of characters in char_list . Can be any string beginning with the character M or N , in any case. If option is omitted, the default is N . M (match) The function returns the position of the first character in string that is present in char_list . N (nomatch) The function returns the position of the first character in string that is not present in char_list .
start	Is the character position in string where the verification begins. start must be a positive integer. If start is greater than the length of string, the function returns 0. If start is omitted, the default value is 1.

Usage Notes

With no additional arguments, the function returns the character position in **string** of the first character that is not present in **char_list**. If all characters in **string** are present in **char_list**, the function returns 0. If **string** is the null string, the function also returns 0.

Example 1

The output of the following program fragment:

```
x = verify('abc', 'abcdefg')
```

is:

```
x = 0
```

Example 2

The output of the following program fragment:

```
x = verify(abc, 'abcdefg')
```

is:

```
x = 1;
```

The value of the symbol `abc` is `ABC`, and none of these characters is in `abcdefg`.

Example 3

The following program fragment verifies that all date values in a file contain only numbers or slash before processing the file.

```
infile = 'orders.txt'
bad_data = 0
OK_chars = '1234567890/'
do lines(infile)
  parse value linein(infile) with order_date .
  bad_data = verify(order_date, OK_chars)
end
call lineout infile
if bad_data > 0 then do
  say 'Some orders have invalid dates'
  say 'These must be corrected to proceed'
  exit
end
else call run_orders
```

Example 4

The following program fragment verifies that employee numbers include a valid department designator in position ≥ 6 before proceeding.

```
infile = 'personnel.txt'
bad_data = 0
dept_letters = 'RDAFL'
do lines(infile)
  parse value linein(infile) with empno .
  if verify(empno, dept_letters, 'M', 6) = 0
    then bad_data = 1
  end
end
call lineout infile
if bad_data then do
  say 'Found some invalid employee numbers'
  exit
```

```
end  
else call do_payroll
```

WORD

Syntax `WORD(string, n)`

Description The WORD function returns a single word from a string of blank-delimited words.

Parameters

Parameters	Explanation
string	The string of blank-delimited words.
n	The number of the word to be returned. n must be a positive integer. If n is greater than the number of words in string, the function returns a null string.

Example 1

The output of the following program fragment:

```
x = word('Happy New Year', 2)
is:
x = 'New'
```

Example 2

The following program fragment determines the compiler to use based on user input.

```
say 'Enter language, program name, and userid'
pull reply /* gets user input in uppercase */
select
  when word(reply, 1) = 'REXX' then comp = 'rxc'
  when word(reply, 1) = 'C' then comp = 'cc'
  otherwise comp = 'unknown'
end
```

WORDINDEX

Syntax `WORDINDEX(string, n)`

Description The WORDINDEX function returns the character position, of the start of a specified word in a string of blank-delimited words.

Parameters

Parameters	Explanation
string	Is the string of blank-delimited words.
n	n must be a positive integer. If n is greater than the number of words in string , the function returns 0.

Example 1

The output of the following program fragment:

```
x = wordindex('Happy New Year', 2)
```

is:

```
x = 7
```

Example 2

The following program fragment uses WORDINDEX to set the right position for parsing lines of data that are not consistently formatted.

```
output = ''
line.0 = 3
line.1 = 'Benjamin Franklin'
line.2 = 'George Washington'
line.3 = 'Abe Lincoln'
do i = 1 to lines.0
  x = wordindex(line.i, 2) - 1
  parse var line.i +(x) last_name
  output = output last_name
end
say strip(output)
```

The output is:

```
Franklin Washington Lincoln
```

WORDLENGTH

Syntax WORDLENGTH(*string*, *n*)

Description The WORDLENGTH function returns the length of a specified word in a string of blank-delimited words.

Parameters

Parameters	Explanation
<i>string</i>	Is the string of blank-delimited words.
<i>n</i>	The number of the word whose length is to be returned. n must be a positive integer. If n is greater than the number of words in string , the function returns 0.

Example 1

The output of the following program fragment:

```
x = wordlength('Happy New Year', 2)

is:

x = 3
```

Example 2

The following program fragment uses WORDLENGTH to set the right position for verifying part numbers.

```
part.0 = 3
part.1 = 'Mouse 1046'
part.2 = 'Keyboard 90772'
part.3 = 'Monitor 806'
do i = 1 to part.0
  x = wordlength(part.i, 1) + 2
  if verify(part.i, '1234567890', , x) \= 0
    then say 'Bad part number for:' line.i
  end
```

WORDPOS

Syntax `WORDPOS(string1, string2 [, start])`

Description The WORDPOS function searches a string of blank-delimited words for the first occurrence of another string of blank delimited words.

Parameters

Parameters	Explanation
string1	string1 is the search string.
string2	string2 is the string to be searched.
start	The number of the word in string2 where the search begins. start must be a positive integer. If start is omitted, the default value is 1.

Usage Notes

Multiple blanks between words in both **string1** and **string2** are treated as a single blank for comparison purposes.

The function returns the word number of the first word in **string2** that matches **string1**. If **string1** is not found in **string2**, the function returns 0.

Example 1

The output of the following program fragment:

```
z = wordpos('time', 'time and time again')
is:
z = 1
```

Example 2

The output of the following program fragment:

```
z = wordpos(time, 'Time flies')
is:
z = 0
```

Example 3

The output of the following program fragment:

```
a = 'the best of times'  
b = 'It was the best of times'  
c = wordpos(a, b)
```

is:

```
c = 3
```

Example 4

The output of the following program fragment:

```
a = 'the best of times, the worst of times'  
b = 'times'  
say wordpos(b, a, 5)
```

is:

```
8
```

Example 5

The following program fragment uses WORDPOS to verify user input.

```
prod_list = 'RMS RMP ROMS RCS'  
say 'Name a CM product'  
parse pull answer  
if wordpos(answer, prod_list) = 0  
  then say "Sorry, that product's not from CM"
```


WORDS

Syntax `WORDS(string)`

Description The WORDS function returns the number of words in a string of blank-delimited words.

Parameters

Parameters	Explanation
string	The string of blank-delimited words.

Example 1

The output of the following program fragment:

```
x = words('Hip, hip, hooray')
```

is:

```
x = 3
```

Example 2

The following program fragment processes a file, discarding all blank lines.

```
file = 'foo.txt'  
do lines(file)  
  line = linein(file)  
  if words(line) \= 0 then call reports line  
end
```

XRANGE

Syntax `XRANGE([start] [, end])`

Description The XRANGE function returns a string of all the valid character encodings within a range.

Parameters

Parameters	Explanation
start	The beginning of the range. If start is omitted, the default value is '00'x.
end	The end of the range. If end is omitted, the default is 'ff'x.

Usage Note

If **start** is greater than **end**, then the result will automatically wrap from 'ff'x to '00'x.

Example 1

The output of the following program fragment:

```
x = xrange('m', 'r')
    is:
x = 'mnopqr'
```

Example 2

For the following program fragment:

```
y = xrange('fa'x, '04'x)
say y
```

the output is the character representation of the hexadecimal string:

```
'fafbfcfdfeff01020304'x
```

Example 3

The output of the following program fragment:

```
a = x2c(b2x('01100011'))
b = d2c(112)
say xrange(a, b)
```

is:

```
cdefghijklmnop
```

X2B

Syntax `X2B(string)`

Description The X2B function converts a string of hexadecimal characters to a string of binary characters.

Parameters

Parameters	Explanation
string	String of hexadecimal characters. This is <i>not</i> a hexadecimal string literal in the form 'nnnn'x. It is simply the hexadecimal digits themselves.

Usage Note

You can use X2B in combination with other conversion functions to convert various formats to their equivalent binary value.

Example 1

The output of the following program fragment:

```
x = x2b('63')  
  
is:  
  
x = '01100011'
```

Example 2

The output of the following program fragment:

```
y = x2b(c2x('a'))  
  
is:  
  
y = '01100001'
```

X2C

Syntax `X2C(string)`

Description The X2C function converts a string of hexadecimal characters to character format.

Parameters

Parameters	Explanation
string	A string of hexadecimal characters. This is <i>not</i> a hexadecimal string literal in the form 'nnnn'. It is simply the hexadecimal digits themselves. string can contain embedded blanks, which are ignored, between pairs of characters.

Usage Notes

If the length of **string** is not an even multiple of 2, it is automatically padded with a leading zero before the conversion is performed.

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

Example 1

The output of the following program fragment:

```
x = x2c('616263')
is:
x = 'abc'
```

Example 2

The output of the following program fragment:

```
say x2c('f')
is the character representation of '0f'x
```

Example 3

The output of the following program fragment:

```
z = x2c(d2x('112'))
```

is:

```
z = 'p'
```

X2D

Syntax X2D(string {, n})

Description The X2D function converts a string of hexadecimal characters to its decimal equivalent.

Parameters

Parameters	Explanation
string	A string of hexadecimal characters. This is <i>not</i> a hexadecimal string literal in the form 'nnnn'x. It is simply the hexadecimal digits themselves. string can contain embedded blanks, which are ignored, between pairs of characters. If string is null, the function returns 0.
n	n indicates that the string represents a signed number expressed in n characters. If necessary, string is padded on the left with zeroes or truncated on the left so that the length of string is n characters. If n is specified, the left-most bit determines the sign; if it is zero, the number is positive; otherwise it is a negative number in twos-complement form. If n is 0, the function returns 0.

Usage Note

The value returned by X2D is expressed as a whole number. If it cannot be expressed as a whole number within the current setting of NUMERIC DIGITS, the *Error 40: Incorrect call to routine*, results.

Example 1

The output of the following program fragment:

```
x = x2d('76')
is:
x = '112'
```

Example 2

The output of the following program fragment:

```
y = x2d(b2x('01100011'))
is:
y = 99
```

Example 3

The output of the following program fragment:

```
z = x2d(b2x('01100001'),1)
```

is:

```
z = 1
```

Example 4

The output of the following program fragment:

```
q = x2d('f063', 4)
```

is:

```
q = -3997
```

6 Using Extensions

This chapter teaches you how to use the REXX function extensions of CM when you customize CM processing for your CM environment.

CM Agent REXX Methods

CM Agent REXX Methods enable you to attach logic to CM objects in the form of methods. Methods are programs that apply to an object or a specific class of objects. Methods can perform virtually any type of operation against CM objects or any other CM-managed elements in the desktop environment. At a minimum, every object class includes a create method and a delete method.

Methods enable the base CM software to be enhanced, extended, and interfaced to external services. The CM Agent software includes a number of CM REXX methods and various other methods that serve as templates for specific functions.

Overview of CM REXX Extensions

CM provides function extensions that support access to CM data through compound symbols and stem variables. These CM REXX extensions act on CM objects that reside in the CM Agent's internal storage object pool, enabling access to CM variables, including Z-named variables. The extensions reference multi-heap object variables much like arrays of variables. Multiheap reference techniques are more efficient than explicit references to the CM variables, and, therefore, are strongly encouraged.

REXX, CM, Objects and Object Paths/Folders

Included in the CM (agent) extensions for REXX is the ability to read and write CM Objects. When processing objects, REXX internally maintains a list/queue of objects being processed. Objects are added to the queue via EDMGET(RADGET) or EDMBLD (REXX) functions. Objects are saved to disk via the EDMSET(RADSET) function and objects are deleted/removed from the REXX queue via the EDMFREE function. There is no CM extension to delete/erase the object file from disk, but the REXX "stream" function can be used to do this.

By default when an object is added to the REXX queue, the default path/directory is the (current) value of IDMLIB. The REXX queue of object names is unique by the object name (case neutral), regardless of the folder that the object resides in, thus if ZFOO is opened in IDMLIB we would have to close ZFOO (EDMFREE) before we can (re)open it in the C:\Bar directory.

The first time EDMGET(RADGET) is called for an object, it will always tied to read it from disk at the specified (or default) directory. If the object exists, the first heap is read and loaded into (REXX) storage. If the object does not exist on disk, then an empty object is allocated with the default heap size of 1024 bytes. If a larger heap size is need for an empty object, EDMBLD can be used to create it. The heap size is the sum all the lengths of the variables/attributes in the object.

Directory paths can be any valid path that exists. If a specified path does not exit, the "current directory" will be used in its place. The specified directory can be specified with or without a trailing slash. Internally this is checked and handled correctly when building the actual file name of the object file. Generally, once an object is added to the REXX queue, the initial directory (default or specified) can't be changed via EDMSET(RADSET). There is an exception. If the object in the REXX queue was built (EDMBLD) or never existed on disk, then, if a directory is specified via the call to EDMSET(RADSET), it will then be come the directory that the specified object will be save to.

In addition to the absolute path, relative paths can be specified. The relative paths are (NOTE: there are two underscores on each size of the names) `__lib__` for IDMLIB, `__adm__` for IDMADM, `__sys__` for IDMSYS, `__data__` for IDMDATA, `__log__` for IDMLOG and `__root__` for IDMROOT.

Using Extensions

This chapter explains how to use CM REXX function extensions when you customize CM processing at your site. CM Agent REXX Methods enables you to attach logic to CM objects in the form of methods. Methods are programs that apply to an object or a specific class of objects. Methods can perform virtually any type of operation against CM objects and/or any other CM-managed elements in the desktop environment. At a minimum, every object class includes a create method and a delete method. Methods enable the base CM software to be enhanced, extended, and interfaced to external services.

The CM Agent software includes a number of CM REXX methods and various other methods that serve as templates for specific functions. Overview of CM REXX Extensions CM provides function extensions that support access to CM data through compound symbols and stem variables. These CM REXX extensions act on CM objects that reside in the CM Agent's in storage object pool, enabling access to CM variables, including Z-named variables. The extensions reference multi-heap object variables much like arrays of variables. Multi heap reference techniques are more efficient than explicit references to the CM variables, and, therefore, are strongly encouraged.

Function Calls and Return Values

Function calls can be made in either of two ways:

- Use the CALL statement.
- Place the return value into a variable.

When you use the CALL statement, CM REXX sets the special variable RESULT to the value returned by the function. Unless otherwise noted, all CM REXX extensions return a value of 0 upon successful execution, and the value 8 if execution fails. The following example contains a CALL statement on the first line, and a return value on the second. Note the use of parentheses in the second example. Also note that REXX is a "case neutral" language. REXX variables function and instructions and be specified in mix case so EdmGet edmget EDMGET are all the same.

The following example contains a CALL statement on the first line, and a return value on the second. Note the use of parentheses in the second example.

Example

```
CALL EDMGET 'ZMASTER',0,'NOLOAD';  
rc=EDMGET('ZMASTER',0,'NOLOAD');
```

Identifying Variables

When an object is fetched/open with EDMGET(RADGET), in addition to returning an error/return code of 0 or 8, the function also creates REXX variables. So if we issue the call:

```
call EDMGET 'myobject'
```

EDMGET will create the REXX variable myobject and save to it the number of heap in the object. So if we issue:

```
call EDMGET 'myobject'

say myobject /* (might) output 10 */
```

A more programmatic way to do this would be like this:

```
object = 'myobject'

call edmget object

say value( object )
```

The REXX function VALUE will return the value of the specified variable name. Check the description of VALUE for more information on this function.

In addition to the REXX variable that contains the name of the object being processed, edmget also sets the variable <objectname>vars to the one more than the total number of variables in the object. So extending the above example we can issue:

```
object = 'myobject'

call edmget object

heaps = value( object )

vars = value( object || "vars" ) - 1

say heaps /* 10, maybe */

say vars /* 62, maybe */
```

Appending *n* to a CM object name (where *n* is an integer between 1 and the number of variables) returns the variable name. Note, however, that the suffix *n* does not return the variable value. For example:

```
do vv = 1 to vars
  attr_name = value( object || vv )
  say attr_name /* might show "ZOS" */
end vv
```

We need to know the attribute/variable name saved in the object to get its value. The attribute/variable values are saved in two forms, which are:

```
<objectname>.<attribute>
<objectname>.resolved.<attribute>
```

Where `<objectname>.<attribute>` is the exact data of the attribute in the object. So, if the value is `&(ZMASTER.ZFOO)`, then that is the value saved to REXX. In the second form, the value that would be saved to REXX for `<objectname>.resolved.<attribute>` would be the "value" of `&(ZMASTER.ZFOO)` which (might) be "BAR."

There is a form used for debugging which is:

```
<objectname>.ventry.<ordinal>
```

In this form, the `<ordinal>` is the numeric entry (starting from 1) of the attribute in the template. The value of `<objectname>.ventry.<ordinal>` is a string of seven "REXX words" which contain the attribute name(1), offset in the object(2), length(3) and flag bytes F1 F2 F3 and F4

REXX variables and CM object values

REXX variables in the form of `Name.a.b.c` are known as "compound variables," which contain a "stem" and a "tail." The "stem" name is the string of characters up to and including the first period. The characters after the first period are the "tail." The tail is actually composed tokens glued together with periods. So when REXX tries to read or write a compound variable like **item.red.green**, it breaks it down as follows:

It first looks at the tail,

```
red.green
```

Then checks to see if any of the "tokens" are REXX variables. In this case we have the tokens red and green. REXX will look up these variables to see if they have a value. If they have a value, that value will be replace the variable name in the tail. So if "red" was 24 and green was 48 the tail would look like this:

```
24.48
```

Then REXX would look up the value of the (REXX) variable "ITEM.24.48" If the (tail) token is not a valid REXX variable (see datatype('s') REXX function) or a value variable has no value, then the token is left as is, and converted to uppercase, so green would become GREEN if "green" is undefined and red would become RED and the REXX variable fetched would be:
"ITEM.RED.GREEN"

So when working with CM objects in REXX, care needs to be taken after EDMGET reads an object into REXX variables. For example:

```
call edmget 'zmaster'
```

There might be a variable in the zmaster object call zos, so see its value we would write:

```
say zmaster.zos /* maybe show "NT" */
```

But if we were to write:

```
zos = 'fred'
```

```
call edmget 'zmaster'
```

```
say zmaster.zos /* this will fetch the REXX variable  
zmaster.fred */
```

REXX first will look to see if the "tail" could be resolved as a REXX variable. In this case, it could because we assigned 'zos' the value of 'fred.'

To safeguard for this condition we could start all the REXX variable we use in our REXX code to start with "@" or "?" which we usually don't see in attribute name, or can use the REXX drop instruction to make sure that REXX will leave the stem as is:

```
zos = 'fred'  
  
call edmget 'zmaster'  
  
drop zos  
  
say zmaster.zos /* this will fetch the REXX variable  
zmaster.zos */
```

Example 1

In this example, rc1 is set to the number of variables in the ZMASTER object, and rc2 is set to the name of the first variable. Had the ZMASTER object not been found on the desktop, then rc1 would have been set to ZMASTERVERS. Had the first variable not been defined, or if it had no value, then rc2 would have been set to ZMASTER1.

```
CALL EDMGET 'ZMASTER','0','NOLOAD'  
rc1 = ZMASTERVERS - 1  
SAY 'There are' RC1 'variables in the ZMASTER object.'  
rc2 = ZMASTER1  
SAY 'The 1st variable is ' rc2
```


Example 2

A more sophisticated and practical example follows below.

This program returns how many variables there are in the first heap of the ZCLIENT object, the name of each variable, and the value of each variable.

```
object = 'zclient'
CALL EDMGET object, 0, 'NOLOAD'
number_variables = value ( object || 'vars' ) - 1
SAY 'There are' number_variables,
    'variables in the first heap of the' object,
    "object"
do n = 1 to number_variables
    variable_name = value( object || n)
    full_name = space( object variable_name, 1, '.' )
    variable_value = value( full_name )
    SAY 'The value of ZCLIENT.'variable_name,
        'is' variable_value
end n
exit 0
```

Example 3

This program will show the contents of the ZERRMSG variable in each heap of the ZERROR object in the PNLREXX.LOG file.

```
CALL EDMGET('ZERROR',0)
Nheaps = ZERROR
Nheaps = Nheaps -1
    /* Loop through all heaps in the object. */
do CurrHeap = 0 to Nheaps by 1
    CALL EDMGET 'ZERROR',CurrHeap
    say ZERROR.ZERRMSG
        end /* Loop through all heaps in the object. */
```

The CM REXX Extension List

The following sections document the CM REXX method extensions.

- EDMADD
- EDMATTR
- EDMBLD
- EDMCMD
- EDMDELHEAP
- EDMDELVAR
- EDMFREE
- EDMGET
- EDMLOC
- EDMRST
- EDMSET
- EDMSORT
- GET_CHILD_OBJ
- LOAD_CHILDREN
- NOWAIT
- NvdVerQueryValueString
FileInfo
- RADGET
- RADSET
- RXXCOMMANDKILL
- RXXCOMMANDSPAWN
- RXXCOMMANDWAIT
- RXXOSENDOFFLINESTRING
- RXXOSENVIROMENT
SEPARATOR
- RXXOSNAME
- RXXOSPATHSEPARATOR
- RXXSLEEP
- WinMessageBox
- WinExpandEnvironmentString
- WinGetVersion
-

EDMADD

Syntax

EDMADD(object_name)

Description

Calling EDMADD adds an empty heap to the end of the specified CM object in memory, and the newly added heap becomes the currently selected heap. The total heap-count (stored in a variable with the same name as the name of the object) is incremented. The newly added heap will not be stored in the object on disk until EDMSET is called.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long. The object name has to already exist in the REXX object queue. If the names does not exist, the call fails.

Example 1

This CM REXX method reads in the lines of an input file, (AUTOEXEC.BAT), and creates SAMPLE.EDM, a multi-heap object with one variable per heap. Each variable contains the value of a single line from that file.

```
/* Create a 'sample' object. */
CALL EDMBLD 'SAMPLE'

/* Define a file to read. */
infile1='C:\AUTOEXEC.BAT'

heapcount = 0
do while lines( infile1 ) > 0
  /* Loop through the input file. If it's */
  /* not the first heap then we need to add */
  /* a new heap to the object. */
  if heapcount > 0
    then CALL EDMADD 'SAMPLE'

  /* Read in a line of the input file. */
  /* Set the SAMPLE object variable. */
  SAMPLE.LINE1 = linein( infile1 )

  /* Save the current heap. */
  CALL EDMSET 'SAMPLE'
```

```
/* Increase the heap counter. */  
heapcount = heapcount +1  
end
```

EDMATTR

Syntax

EDMATTR(filename)

Description

The value returned from EDMATTR contains a string with the following file attribute information:

- File exists (or does not exist).
- File size (in bytes).
- Date file was last updated
- Time file was last updated (in 24 hour format).
- Time file was updated (in AM/PM format).

If the file does not exist, a string with the character value of 8 is returned.

Parameters

Parameter	Explanation
filename	The full name and path of the file you are querying.

Usage Note

We recommend using the REXX built-in function STREAM instead of EDMATTR. For more information on the STREAM function, see *Chapter 5: Built-In Functions*.

Example

The output of the following program fragment:

```
CALL EDMATTR 'C:\AUTOEXEC.BAT'
```

is:

```
0 126848 02-14-96 03:12:00 03:12a
```

In the above return value:

0 indicates the file exists.

126848 is the size of the file in bytes.

02-14-96 is the date the file was last updated.

03:12:00 is the time the file was last updated.

03:12a is the time the file was last updated.

EDMBLD

Syntax `Call EDMBLD object_name[,heap_size[,path]]`

Description Calling EDMBLD adds a new object to the REXX object queue.

Parameters

Parameter	Explanation
object_name	A valid CM object name. object_name can be up to eight characters long. The object name has to be a new object. If the object name already exists in the REXX object queue, the call fails.
heap_size	The size of each heap within the object to be built. The default is 1K (1024 bytes). The heap_size option can be between 1 to 6144 bytes. If the object already is present in REXX (via a EDM/RADGET), the function call will fail. Thus, EDMBLD creates a new entry in the REXX object queue. Each additional heap adds only heap_size bytes to the size of the object.
path	Is the location/directory that the object will be saved to. The default is the (current) value of IDMLIB. When this object/heap is save, on the first call to EDMSET, if a path is specified, then that path becomes the allocated path for this object.

Example

```
CALL EDMBLD 'MAINT', 256, "c:\myobjects"
```

EDMCMD

Syntax

```
EDMCMD('modifier command_line')
```

Description

Calling EDMCMD allows you to use CM Extended Batch command line modifiers to execute a platform-specific command.

Parameters

Parameter	Explanation
modifier	Enhances your control over the desktop during execution of the command. Modifiers can be grouped together but must be placed before command_line . CM REXX supports only the following modifiers: SHOW , HIDE , WAIT , NOWAIT , and FULLSCR .
SHOW	Specify SHOW to see the command or program as it executes in its default sized window.
HIDE	Specify HIDE to prohibit the display of a window during startup and execution of a program or command.
WAIT	Specify WAIT to force CM REXX to wait for the completion of the command before resuming execution of the REXX program.
NOWAIT	Specify NOWAIT to continue execution of the REXX program as soon as the command is launched, without waiting for completion of the command.
FULLSCR	Specify FULLSCR to display the command's execution in a full screen window, rather than its default window size.
command_line	A platform-specific, valid command with proper syntax and parameters.

Usage Note

CM REXX interprets the command line modifier and passes the command to the local operating system.

Example

```
CALL EDMCMD 'NOWAIT HIDE EDMDEMON'
```


EDMDELHEAP

Syntax EDMDELHEAP (object_name)

Description Calling EDMDELHEAP deletes the current heap from a CM object. The heap is immediately deleted from both the object in memory, and from the object as stored on disk.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.

Example

This example deletes heap 5 from the SAMPLE object.

```
CALL EDMGET 'sample', 5  
CALL EDMDELHEAP 'sample'
```

EDMDELVAR

Syntax EDMDELVAR(object_name,variable)

Description Calling EDMDELVAR deletes a specified variable from a CM object. The variable is deleted immediately from both the object in memory, and from the object as stored on disk.

Parameters

Parameter	Explanation
object_name	The name of a valid CM object. object_name can be up to eight characters long.
variable	The name of a variable contained in the object. variable can be up to eight characters long.

Example

This example deletes the VAR1 variable from the SAMPLE object.

```
CALL EDMGET 'SAMPLE'  
CALL EDMDELVAR 'SAMPLE', 'VAR1'
```

EDMFREE

Syntax

EDMFREE (object_name)

Description

Calling EDMFREE removes the specified CM object from the REXX object queue. In addition to the "REXX object queue" there is an internal object queue. For the most part the REXX and internal queue are process in parallel. There are certain cases where this is not true, namely when invoking REXX via radpnlwr.exe. In this case, radpnlwr is the owner of ZMASTER, so if EDMFREE is called with ZMASTER, it would be purged from the REXX object queue, and not the internal object queue. The term "managed" is used to describe this. (See nvdobjects function call) Object that REXX has full control over are "managed," otherwise they are unmanaged. This should not be a concern when running REXX via radrexw.exe or radrex on unix.

There is a finite number of objects that can be loaded in the internal object queue. For Unix it is 50, otherwise 20.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long. If the object is not in the REXX object queue, the call returns 8.

Example

```
CALL EDMFREE 'MYOBJECT'
```

EDMGET

Syntax

Call EDMGET *object_name*, [*heap_number* [, 'NOLOAD' [, *path*]]]

Description

Calling EDMGET reads the specified heap from a CM object into memory, making it the currently selected heap. If you specify a non-existent heap, EDMGET returns a value of 8.

NOTE: You can use this command on the CM Configuration Server OR the CM Agent. However, you must note that heap numbers on the RCS start at 1, while heap numbers on the client start at 0.

Parameters

Parameter	Explanation
<i>object_name</i>	A valid CM object. object_name can be up to eight characters long.
<i>heap_number</i>	Specifies the relative position of the heap or instance in the object. heap_number must be an integer between 0 and 65,536. The maximum value of heap_number is one less than the number of heaps in the object. Default is 0.
'NOLOAD'	If you include the NOLOAD option, EDMGET will NOT "reload" the object template off from disk. If the object being process can "change" (variables/attributes added and/or deleted) while the REXX program/script is running, it best not to specify this argument. If the template does not change then specifying NOLOAD will save a (disk) read of the object's template.
<i>Path</i>	Is the location/directory that the object will be read/written to. The default is the (current) value of IDMLIB. Once the path is established via a "GET", (for the most part), this directory will stay in effect until the object is purged from the REXX object queue via EDMFREE.

Example

This program will show the contents of the ZERRMSG variable in each heap of the ZERROR object written to the log file.

```
Object = "ZERROR"
Dir     = "c:\Temp\Objects"

CALL EDMGET Object, 0,, Dir /* NOLOAD was omitted as a null argument
*/

Nheaps = value( Object )
NVars  = value( Object || "vars" )
Nheaps = Nheaps - 1

/* Loop through all heaps in the object. */
for CurrHeap = 0 to Nheaps by 1
  CALL EDMGET Object, CurrHeap
```

```
errorvar = Object || ".ZERRMSG"  
say ZERROR.ZERRMSG  
end /* Loop through all heaps in the object. */
```

RCS: You can use this function on the CM Configuration Server. Note that only the first two arguments are supported. Also, note that heap numbers on the RCS start at 1, while heap numbers on the client start at 0. A heap number of 0 on the RCS means "the current heap."

EDMGETV

Syntax EDMGETV object_name, var_name, [heap_number [, 'NOLOAD' , path]]]

Description Calling EDMGETV reads and returns the specified variable from a CM object.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.
Var_name	Name of the variable to access.
Heap_number	Specifies the relative position of the heap or instance in the object. heap_number must be an integer between 0 and 65,536. The maximum value of heap_number is one less than the number of heaps in the object. The default is 0. If the specified heap is out of range is out of range, a REXX syntax error will be raised.
NOLOAD	If you include the NOLOAD option, EDMGET will NOT "reload" the object template off from disk. If the object being process can "change" (variables/attributes added and/or deleted) while the REXX program/script is running, it best not to specify this argument. If the template does not change then specifying NOLOAD will save a (disk) read of the object's template.
Path	The location/directory that the object will be read/written to. The default is the (current) value of IDMLIB. Once the path is established via a "GET", (for the most part), this directory will stay in effect until the object is purged from the REXX object queue via EDMFREE,

Example

```
Say EDMGETV( "ZMASTER", "ZOS" ) /* outputs (maybe) WINXP */
```

CM Configuration Server note - You can use this function on the CM Configuration Server. Note that only the first three arguments are supported. The CM Configuration Server supports a fourth argument which is a flag. If its value is 1, then if the value fetched is in the form of &(object.variable), the CM Configuration Server will try to find this value. If the fourth argument is missing or is 0, then the value is returned as-is. Also note that heap numbers on the CM Configuration Server start at 1, while heap numbers on the client start at 0. A heap number of 0 on the CM Configuration Server means the "current heap."

EDMLOC

Syntax EDMLOC (filename)

Description The value returned from EDMLOC specifies whether or not a file exists. A return value of 0 indicates the file exists. If the file does not exist, 8 is returned.

Parameters

Parameter	Explanation
filename	The full name and path of the file you are querying.

Usage Note

We recommend using the built-in function `STREAM` with the `QUERY EXISTS` option instead of `EDMLOC`. For more information on the `STREAM` function, see *Chapter 5: Built-In Functions*.

Example

```
CALL EDMLOC 'C:\autoexec.bat'
```

EDMRST

Syntax EDMRST(object_name)

Description Calling EDMRST resets the specified CM object to a single heap object in memory. A call to EDMSET must be made to save this change to disk.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.

Example

This example demonstrates that the values in the current heap will be saved in the only heap remaining after EDMSET is called.

```
CALL EDMGET 'MYOBJECT', 5
  /* Get sixth heap of object.          */
CALL EDMRST 'MYOBJECT'
  /* Reset object to single heap.       */
CALL EDMSET 'MYOBJECT'
  /* The single heap object's variables */
  /* have the values that were in the sixth */
  /* heap, originally.                   */
```


EDMSET

Syntax Call EDMSET object_name [,path]

Description Calling EDMSET saves the current heap for **object_name** to disk.

NOTE: You can use this command on the CM Configuration Server OR the CM Agent. However, you must note that heap numbers on the RCS start at 1, while heap numbers on the client start at 0.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.
Path	Is the location/directory that the object will be written to. The default is the (current) value of IDMLIB. Once the path is established via a GET/BLD, this directory will stay in effect until the object is purged from the REXX object queue via EDMFREE. EDMSET can override the path if the object did not exist on disk or was add to the REXX object queue via EDMBLD.

Examples

This CM REXX method reads in the lines of an input file, (AUTOEXEC.BAT), and creates SAMPLE.EDM, a multi-heap object with one variable per heap. Each variable contains the value of a single line from that file.

```
/* Create a 'sample' object. */
CALL EDMBLD 'SAMPLE'

/* Define a file to read. */
infile1='C:\AUTOEXEC.BAT'

heapcount = 0
do while lines( infile1 ) > 0
  /* Loop through the input file. If it's */
  /* not the first heap then we need to add */
  /* a new heap to the object. */
  if heapcount > 0
    then CALL EDMADD 'SAMPLE'

  /* Read in a line of the input file. */
  /* Set the SAMPLE object variable. */
  SAMPLE.LINE1 = linein( infile1 )

  /* Save the current heap. */
```

```
CALL EDMSET 'SAMPLE'  
  
/* Increase the heap counter. */  
heapcount = heapcount +1  
end
```

RCS: You can use this function on the CM Configuration Server. Note that on the RCS there is a second argument, the heap number. The specified heap number can be set from 1 to MAX+1, where MAX is the total number of heap in the (RCS) object. If MAX+1 is specified then that heap is created. **NOTE** that EDMBLD is not available on the RCS. Also, note that heap numbers on the RCS start at 1, while heap numbers on the client start at 0. A heap number of 0 on the RCS means "the current heap."

EDMSORT

Syntax `EDMSORT(object_name, variable)`

Description Calling EDMSORT will sort the heaps of a multi-heap CM object into alphabetic ascending order by the contents of a specific variable, and save the changes to disk.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.
variable	The name of a variable contained within the object. variable can be up to eight characters long, and must be specified in uppercase.

Example

```
CALL EDMSORT 'MYOBJECT', 'VAR1'
```

GET_CHILD_OBJ

Description

This function is equivalent to EDMGET, and is no longer supported. Use EDMGET instead. See LOAD_CHILDREN for information describing how to access and manipulate child and grandchild objects.

LOAD_CHILDREN

Syntax

```
call LOAD_CHILDREN 'object_name'
```

Description

Calling the LOAD_CHILDREN function provides visibility to child and grandchild objects of an object. Once visibility is established, the child and/or grandchild objects can be read into storage with EDMGET and written to disk by EDMSET.

Parameters

Parameter	Explanation
object_name	The name of the object whose children and grandchildren objects you need to access.

Usage

CM objects pertinent to a service are stored in the service's IDMLIB location and its sub-directories. The IDMLIB location is the directory identified by the IDMLIB setting in the [NOVAEDM] section of WIN.INI. This setting is dynamically changed by the CM Agent to a unique directory associated with the service being installed or otherwise manipulated. For example, if the subscriber is installing a service named HELLO, the IDMLIB directory for the HELLO service might be:

```
C:\Program Files\Hewlett-Packard\CM\Lib\username\ABC\SOFTWARE  
\ZSERVICE\HELLO
```

The components comprising the service are stored in a tree of sub-directories, for which the service's IDMLIB directory is the root. Child and grandchild objects of an object stored within this tree structure are stored in sub-directories of the directory in which the parent object is stored.

EDMGET and EDMSET normally only have the ability to access objects that are stored in the IDMLIB directory. The LOAD_CHILDREN function provides EDMGET and EDMSET the ability to access child and grandchild objects of parent objects located in the IDMLIB directory.

Parent objects contain information identifying their child objects. You can inspect an object using CM Admin Agent Explorer to determine which objects will be made accessible by calling LOAD_CHILDREN for that object. For example, the DMSYNC object for the HELLO service may appear as follows in CM Admin Agent Explorer:

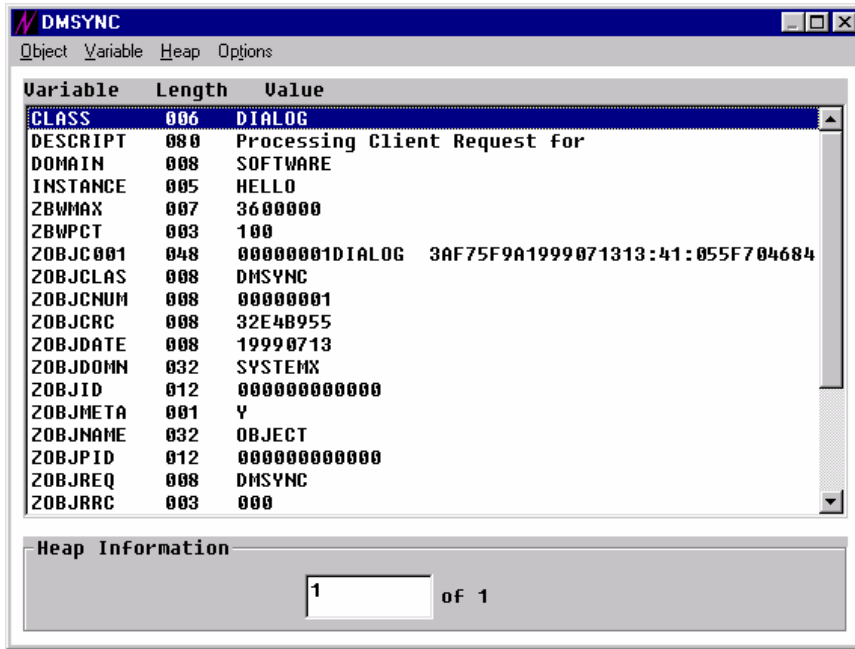


Figure 6.1 ~ DMSYNC object.

This object is stored in the IDMLIB location for the HELLO service, which is:

```
C:\Program Files\Hewlett-Packard\CM\Lib\username\ABC\SOFTWARE
\ZSERVICE\HELLO
```

Child objects for an object are listed in ZOBJC nnn variables (where nnn is 001 to the number of child objects belonging to the parent object), in the parent object. The number of child objects belonging to the parent object is stored in the ZOBJCNUM variable. If the ZOBJCNUM variable is not present in the object, or if its value is 00000000, the object has no child objects.

Each ZOBJC nnn variable contains a fixed format text string providing information about the child object, as follows:

Characters	Contains
1 – 8	Number of instances (heaps) in the child object.
9 - 16	Child object name
17 – 24	Child object CRC
25 – 40	Latest child object date and time stamp
41 - 48	Tree CRC

The ZOBJID variable contains the object ID. The name of the sub-directory containing child objects for this object is derived from the object ID value by concatenating the rightmost eight characters of the object ID (in this case 00000000) with characters 2-4 of the object ID (in this case 000), separated by a period. Thus, the sub-directory name containing the child object of the DMSYNC object, in this case, is 00000000.000.

This DMSYNC object has one child object – DIALOG. It is stored in the following directory:

```
C:\Program Files\Hewlett-Packard\CM\Lib\username\ABC\SOFTWARE
\ZSERVICE\HELLO\00000000.000
```

When viewed in the CM Admin Agent Explorer, the DIALOG object appears as follows:

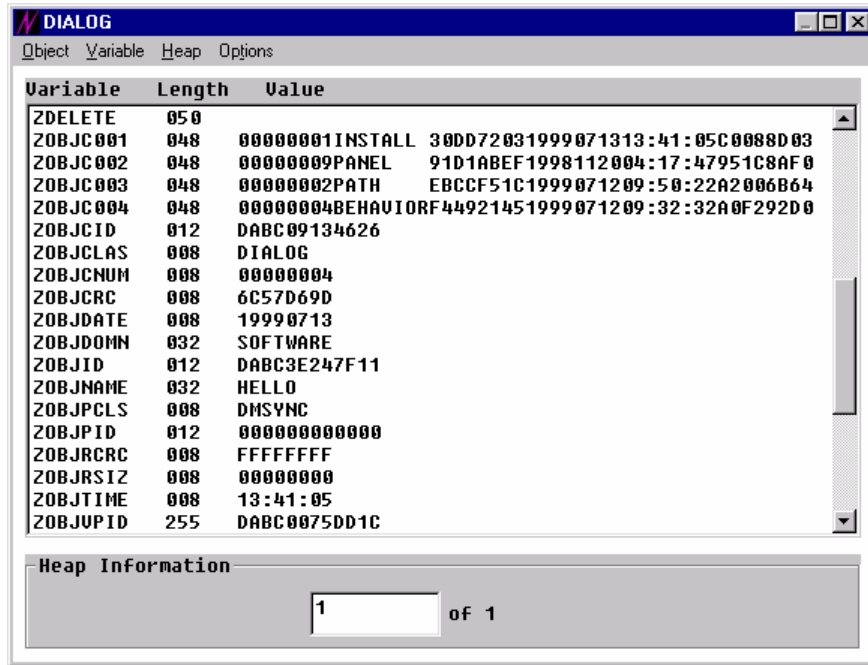


Figure 6.2 ~ DIALOG object.

The DIALOG object has four child objects: INSTALL (1 heap), PANEL (9 heaps), PATH (2 heaps) and BEHAVIOR (4 heaps). These objects are grandchildren of the DMSYNC object. They are stored in a sub-directory named 3E247F11.ABC (name derived from the ZOBJID variable), beneath the directory in which the DIALOG object is stored.

The current implementation of LOAD_CHILDREN provides visibility to child and grandchild objects only. If identically named objects are both a child and a grandchild object, only the grandchild object is made visible by LOAD_CHILDREN.

Warning

Do not modify the ZBJC*nnn*, ZOBJID, or ZBJCNUM variables.

Example 1

The following program fragment:

```
CALL LOAD_CHILDREN 'DMSYNC'
```

will make children and grandchildren of DMSYNC object visible.

Example 2

The following program fragment:

```
CALL EDMGET 'INSTALL'
```

will get the grandchild INSTALL object.

NOWAIT

Syntax `call NOWAIT 'command_line'`

Description Calling the NOWAIT function allows CM REXX to start a process, then immediately begin processing the next command.

Parameters

Parameter	Explanation
command_line	A platform-specific, valid command with proper syntax and parameters.

Example

This example leaves the display message: `Exiting CM REXX` on the screen and exits CM REXX.

```
NOWAIT 'EDMBOX.EXE "Exiting CM REXX."'
EXIT 0
```

NVDOBJECTS

Syntax

```
Count = NvdObjects( "AllObjects" )
```

Description

Calling NVDOBJECTS returns in the specified REXX (stem) variable, all the objects in the REXX object queue and information about these objects.

Parameters

Parameter	Explanation
Count	The name of the (REXX) variable that will be built as a stem list containing the objects allocated. If the (REXX) variable name is "objs", then objs.0 contains the number of allocated objects and each object can be found in the variables objs.1, objs.2 objs.n

Example

```
count = nvdobjects( 'curobjs' )
do oo = 1 to curobjs.0
  say curobjs.oo
end oo
```

The output for this might be:

```
Name=ZPOOLTAB VTAB=00835D40 Size=1024 Built=No OnDisk=No Managed=Yes
PathReset=No Saves=0 ForcedPath=No Path=
Name=ZMASTER VTAB=00838578 Size=4096 Built=No OnDisk=Yes Managed=Yes
PathReset=No Saves=0 ForcedPath=No Path=E:\RadClient\Lib
Name=ZLOCAL VTAB=0083ADB0 Size=1024 Built=No OnDisk=Yes Managed=Yes
PathReset=No Saves=0 ForcedPath=No Path=E:\RadClient\Lib
```

Where **Name=** is the object name, **VTAB=** is the internal buffer address, **Built=** is if the object was created by EDMBLD.

If **Managed=Yes**, the object can be removed from the internal object queue (not the REXX object queue). If No, the object will remain in the internal queue even after a EDMFREE removes the object from the REXX object queue.

PathReset if the path was reset via a EDM/RAD "SET".

Saves determines how many times the object was saved to disk.

ForcedPath if the specified directory was invalid and the path was reset to the current directory.

Path= is the path the object is allocated to.

This output can be (REXX) parsed like this:

```
count = nvdojects( 'curobjs' )  
  
do oo = 1 to curobjs.0  
  
  parse var curobjs.oo 1 "Name=" Objname .  
  
  parse var curobjs.oo 1 "Path=" Objpath .  
  
  say curobjs.oo  
  
end oo
```

NVDPATHS

Syntax RealPath = NvdPaths(PsuedoPath)

Description Calling NVDPATHS will return the value if the pseudo path specified. If the psuedopath is undefined, the value returned is the value passed to nvdpaths

Parameters

Parameter	Explanation
RealPath	The relative/pseudo paths are <code>__lib__</code> for IDMLIB, <code>__adm__</code> for IDMADM, <code>__sys__</code> for IDMSYS, <code>__data__</code> for IDMDATA, <code>__log__</code> for IDMLOG and <code>__root__</code> for IDMROOT. Note, that there are two underscores around each of the pseudo names and they are case neutral.

Example

```
current_idmlib = nvdpaths( "__lib__" )
say current_ibmlib /* outputs (maybe) c:\program files\Hewlett-Packard\CM\lib
*/

test = nvdpaths( "fred" )
say test /* outputs: fred */
```


RADGET

Syntax

```
RADGET(object_name,directory,heap_number[, 'NOLOAD'])
```

Description

Calling RADGET reads the specified heap from a CM object (located in the specified directory or folder) into memory, making it the currently selected heap. If you specify a non-existent heap, RADGET returns a value of 8.

NOTE: You can use this command on the CM Configuration Server OR the CM Agent. However, you must note that heap numbers on the RCS start at 1, while heap numbers on the client start at 0.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.
directory	Folder to read object_name. Enter a fully qualified directory name, or one of the following CM folder names. IDMR00T for the value of the IDMR00T folder set in NVD.INI. IDMLIB for the value of the IDMLIB folder set in NVD.INI or RADSETUP, the Bootstrap priming folder (IDMR00T/RADSETUP).
heap_number	Specifies the relative position of the heap or instance in the object. heap_number must be an integer between 0 and 65,536. The maximum value of heap_number is one less than the number of heaps in the object.
'NOLOAD'	If you include the NOLOAD option, RADGET will NOT "reload" the object template off from disk. If the object being process can "change" (variables/attributes added and/or deleted) while the REXX program/script is running, it best not to specify this argument. If the template does not change then specifying NOLOAD will save a (disk) read of the object's template.

Example

This program first reads MYOBJECT from a fully qualified location, and assigns test variables. After the read, the object is purged from the (REXX) object queue by EDMFREE, then allocated again with EDMBLD, and then written to a different folder location.

Successive parts of this program illustrate reading and writing MYOBJECT variables from and to the current heaps of the IDMROOT and IDMLIB folders (specified in NVD.INI), and the RADSETUP bootstrap folder. At the end of this program, MYOBJECT contains different variables in each of the folder locations specified by RADSET.

```
trace i
CALL RADGET "MYOBJECT", "C:\PROGRA~1\Hewlett-Packard\CM\LIB\RADSETUP"
SAY "Opened Object"

MYOBJECT.IP="1.1.2.2"
MYOBJECT.TESTVAR1="Hello"
MYOBJECT.TESTVAR2="World"
CALL EDMFREE "MYOBJECT"

CALL EDMBLD "MYOBJECT"
CALL RADSET "MYOBJECT", "C:\PROGRA~1\Hewlett-Packard\CM\LIB\"
CALL EDMFREE "MYOBJECT"

CALL RADGET "MYOBJECT", "IDMROOT", "0"
MYOBJECT.TESTVAR1="Jello"
MYOBJECT.TESTVAR2="World"
CALL RADSET "MYOBJECT", "IDMROOT"
CALL EDMFREE "MYOBJECT"

CALL RADGET "MYOBJECT", "RADSETUP", "0"
MYOBJECT.TESTVAR1="Merry"
MYOBJECT.TESTVAR2="World"
CALL RADSET "MYOBJECT", "RADSETUP"
CALL EDMFREE "MYOBJECT"

CALL RADGET "MYOBJECT", "IDMLIB", "0"
MYOBJECT.TESTVAR1="Hello"
MYOBJECT.TESTVAR2="World"
CALL RADSET "MYOBJECT", "IDMLIB"
CALL EDMFREE "MYOBJECT"
```



```
SAY "Saving Objects"
```

```
RETURN 2
```

RADSET

Syntax call RADSET (object_name,directory)

Description Calling RADSET saves the current heap for object_name to disk.

Parameters

Parameter	Explanation
object_name	A valid CM object. object_name can be up to eight characters long.
directory	The location/directory to which the object is written. The default is the (current) value of IDMLIB. Once the path is established via a GET/BLD, this directory will stay in effect until the object is purged from the REXX object queue via EDMFREE. RADSET can override the path if the object did not exist on disk or was add to the REXX object queue via EDMBLD. In addition to actual directory names, a pseudo CM folder name can be used. Their values are: IDMROOT - The IDMROOT folder set in NVD.INI IDMLIB - The IDMLIB folder set in NVD.INI RADSETUP - The Bootstrap priming folder (IDMROOT/RADSETUP)

RXXCommandKill

Syntax `Call RxxCommandKill Handle`

Description Exits the current process. Always returns 0.

Example

In this example, we "spawn" the command myapp.exe, checking every 2 seconds to see if it is finished. If 120 seconds pass before it is finished, then we kill the process and continue with the REXX program.

```
TotalSleep = 0
SleepFor    = 2
CMD         = "MyApp.Exe"
CMD_Handle = RxxCommandspawn( CMD )

Do Until DataType( RC, 'n' )
  TotalSleep = TotalSleep + SleepFor
  Call RxxSleep SleepFor
  If TotalSleep > 120
    Then Do
      Call RxxCommandKill CMD_Handle
      Leave
    End
  Else Say "Waiting ..."
  RC = RxxCommandwait( CMD_Handle, "t" )
End
```

RXXCommandSpawn

Syntax Handle = RxxCommandSpawn(CmdName)

Description Returns the "handle" of the spawned command. If the call fails, -1 is returned. Use this value with the RxxCommandKill and RxxCommandWait functions.

Example

In this example, we "spawn" the command myapp.exe, checking every 2 seconds to see if it is finished. If 120 seconds pass before it is finished, then we kill the process and continue with the REXX program.

```
TotalSleep = 0
SleepFor    = 2
CMD         = "MyApp.Exe"
CMD_Handle = RxxCommandspawn( CMD )

Do Until DataType( RC, 'n' )
  TotalSleep = TotalSleep + SleepFor
  Call RxxSleep SleepFor
  If TotalSleep > 120
    Then Do
      Call RxxCommandKill CMD_Handle
      Leave
    End
  Else Say "Waiting ..."
  RC = RxxCommandwait( CMD_Handle, "t" )
End
```

RXXCommandWait

Syntax `Status = RxxCommandWait(Handle, Option)`

Description Waits for a command spawned with RXXCommandSpawn. See RXXCommandSpawn.

Parameters

Parameter	Explanation
Handle	Value is returned by RXXCommandSpawn
Option	This option can be omitted. If omitted, RXXCommandWait will wait until the spawn command completes. If Option is specified as t, RXXCommandWait returns immediately with the numeric exit code of the completed command. If the command is not completed, it returns WAITPENDING.

Example

In this example, we "spawn" the command myapp.exe, checking every 2 seconds to see if it is finished. If 120 seconds pass before it is finished, then we kill the process and continue with the REXX program.

```
TotalSleep = 0
SleepFor   = 2
CMD        = "MyApp.Exe"
CMD_Handle = RxxCommandspawn( CMD )

Do Until DataType( RC, 'n' )
  TotalSleep = TotalSleep + SleepFor
  Call RxxSleep SleepFor
  If TotalSleep > 120
    Then Do
      Call RxxCommandKill CMD_Handle
      Leave
    End
  Else Say "Waiting ..."
  RC = RxxCommandwait( CMD_Handle, "t" )
End
```

RXXOSEndOfLineString

Syntax

EOL = RxxOSEndOfLineString()

Description

Returns the character(s) that mark the End of Line (EOL) for a text file for the OS. For UNIX EOL = LF (0x0d) and for Windows EOL = CRLF (0x0d0a).

RXXOSEnvironmentSeparator

Syntax

```
PathChar = RxxOSEnvironmentSeparator()
```

Description

Returns the character for building PATH enviromental variable. For UNIX PathChar= : and for Windows PathChar= ;.

RXXOSName

Syntax

```
OSClass = RxxOsName ()
```

Description

Return the class of the operating system that the REXX program is running on. For UNIX, it returns the string "UNIX" and for Windows it returns "WINNT", "WIN95" or "WIN98". To get the actual Windows OS, use the WinGetVersion function. See WinGetVersion for more information.

RXXOSPathSeparator

Syntax

SepChar = rxxospathseparator()

Description

Returns the character for building file paths. For Unix it returns a forward slash (/) and for Windows it returns a back slash (\).

RXXSleep

Syntax RXXSleep <seconds>

Description Suspends the current process for the amount of <seconds> specified.

Parameters

Parameter	Explanation
Seconds	Number of seconds for which to suspend the process

Example

The example suspends the current process for 2 seconds.
Call RxxSleep 2

Examples

```
KeyTag = WinMessageBox( "Is it OK to exit", "MyApp",  
"MB_OKCANCEL", "MB_DEFBUTTON2", "MB_TOPMOST" )
```

```
say keytag
```

```
KeyTag = WinMessageBox( "Continue Processing?", "MyApp",  
"MB_ABORTRETRYIGNORE", "MB_DEFBUTTON3", "MB_TOPMOST" )
```

```
say keytag
```

WinExpandEnvironmentStrings

Syntax `call WinExpandEnvironmentStrings
 string`

Description This function expands selected environment variables using a string parameter. It returns the string parameter with the selected environment variables substituted.

Parameters

This function expands selected environment variables using a string parameter. It returns the string parameter with the selected environment variables substituted.

Parameter	Explanation
string	String containing environment variables in the form %variable% to be substituted.

Example

The following string returns the subscriber computer's current path setting:

```
call WinExpandEnvironmentStrings "%path%"
```

WinGetVersion

Syntax Call WinGetVersion

Description Returns the class of Windows OS.

Usage

When invoked, this function creates five REXX variables:

MajorVersion

MinorVersion

BuildNumber

PlatformID

CSDVersion

If you call the function with an argument, the value is concatenated with the variables listed above.

For example, Call WinGetVersion "@" sets and creates:

@MajorVersion

@MinorVersion

@BuildNumber

@PlatformID

@CSDVersion

Example

This example gets the exact Windows OS.

```
/*-----*/
/* Show OS */
/*-----*/
Say TheOSName()
Exit
```

TheOSName:

```
Procedure
Call WinGetVersion "@"
OS = "?"
Wstr = RxxOSName()
Select
  When Wstr = "WINNT"
    Then Select
      When @MajorVersion = 3 | @MajorVersion = 4
        Then OS = "WINNT"
      When @MajorVersion = 5 & @MinorVersion = 0
        Then OS = "WIN2K"
      When @MajorVersion = 5 & @MinorVersion = 1
        Then OS = "WINXP"
      Otherwise
        OS = "WINNT"
    End
  When Left( Wstr, 4 ) = "WIN9"
    Then Select
      When @MajorVersion = 4 & @MinorVersion = 0
        Then OS = "WIN95"
      When @MajorVersion = 4 & @MinorVersion = 10
        Then OS = "WIN98"
      When @MajorVersion = 4 & @MinorVersion = 90
        Then OS = "WINME"
      Otherwise
        OS = Wstr
    End
  Otherwise
    OS = Wstr
End
Return OS
```

7 Registry Manipulation Functions

This chapter describes CM REXX functions that enable you to inspect and manipulate the Windows Registry.

Registry Manipulation Functions

The provided functions enable your REXX methods to open, inspect, create, modify, delete and close registry keys.

When you open access to a registry key, Windows returns a handle to that key. A handle is a value that Windows recognizes as an alias for the key. The handle is useful in calls to other functions that manipulate the designated key. You should store the handle value that Windows provides in a variable so you can provide the handle in subsequent function calls that refer to the same key. When you close the key, Windows destroys the handle.

In the function descriptions that follow, the **type** parameter must be chosen from this table:

Value	Data that can be stored in the associated Registry key
REG_BINARY	Binary data in any form.
REG_DWORD	A 32-bit number.
REG_DWORD_LITTLE_ENDIAN	A 32-bit number in little-endian format (same as REG_DWORD). In little-endian format, the most significant byte of a word is the right most. This is the most common format for computers running Windows NT and Windows 95.
REG_DWORD_BIG_ENDIAN	A 32-bit number in big-endian format. In big-endian format, the most significant byte of a word is the left most.
REG_EXPAND_SZ	A null-terminated string that contains unexpanded references to environment variables (for example, "%PATH%"). It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.
REG_LINK	A Unicode symbolic link.
REG_MULTI_SZ	An array of null-terminated strings, terminated by two null characters.
REG_NONE	No defined value type.
REG_RESOURCE_LIST	A device-driver resource list.
REG_SZ	A null-terminated string. It will be a Unicode or ANSI string depending on whether you use the Unicode or ANSI functions.

The examples in this chapter are based upon the TestKey key and its sub-keys, as seen here in the Registry Editor in Windows:

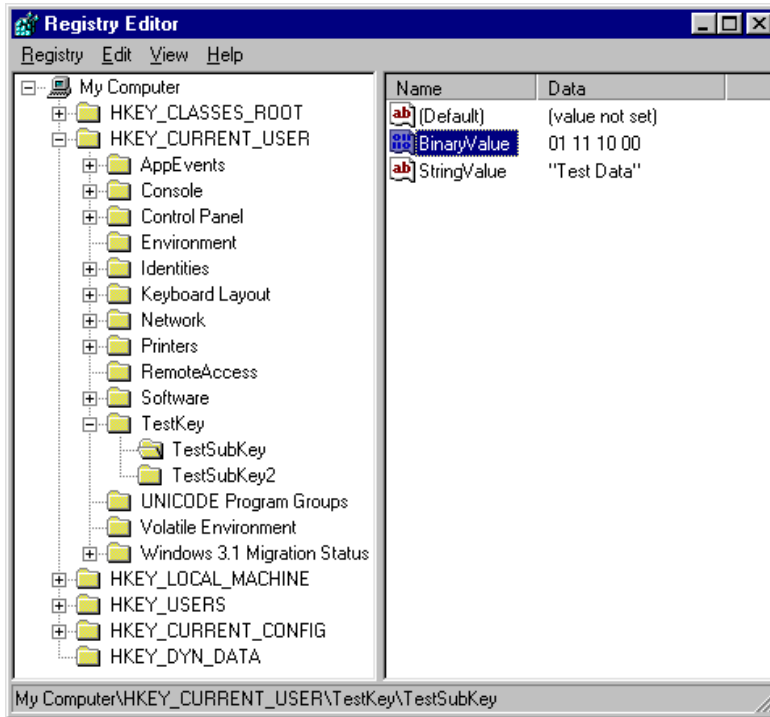


Figure 7.1 ~ Registry Editor in Windows.

WinRegCloseKey

Syntax WinRegCloseKey handle

Description This function closes a key previously opened by one of the other registry manipulation functions. It returns the Win32 error code (0 if successful).

Parameters

Parameter	Usage
handle	Handle of key to close.

Example

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey"

w32err=WinRegCreateKey(hive, key, myhandle,
    access,"orgstatus")
say w32err
say myhandle
say orgstatus
.
. /* some processing on the key */
.
w32err=WinRegCloseKey(myhandle)
say w32err /* 0 = key successfully closed */
```

WinRegCreateKey

Syntax

```
call WinRegCreateKey handle,  
                    key,  
                    hkey,  
                    access,  
                    class,  
                    options,  
                    disposition
```

Description This function creates a new registry key or opens an existing registry key. It returns the Win32 error code (0 if successful).

Parameters

Parameter	Usage
handle	Handle of parent key. handle can be a handle returned from a prior call to WinRegCreateKey or WinRegOpenKey, or a predefined hive name, one of the following: "HKEY_CLASSES_ROOT" "HKEY_CURRENT_USER" "HKEY_LOCAL_MACHINE" "HKEY_USERS" "HKEY_PERFORMANCE_DATA" "HKEY_CURRENT_CONFIG" "HKEY_DYN_DATA"
key	Name of the key to open or create.
hkey	Name of a variable to receive the handle to the key provided by Windows. Enclose the name in quotes.
access	Specifies an access mask that specifies the desired security access for the new key. Choose one of the following: "KEY_ALL_ACCESS" Key can be read and written. All permission granted by KEY_READ and KEY_WRITE plus permission to create a symbolic link. "KEY_READ" The key is read-only. Permission to query subkey data, enumerate subkeys and permission to receive change notification. "KEY_WRITE" Permission to add sub-keys and sub-key values to the key.
class	Class for newly created key.

Parameter	Usage
options	<p>How key is created. One of:</p> <p>"REG_OPTION_NON_VOLATILE" This key is not volatile; this is the default. The information is stored in a file and is preserved when the system is restarted. The RegSaveKey function saves keys that are not volatile.</p> <p>"REG_OPTION_VOLATILE" <i>Windows NT:</i> This key is volatile; the information is stored in memory and is not preserved when the system is restarted. The RegSaveKey function does not save volatile keys. This flag is ignored if the key already exists.</p> <p><i>Windows 95:</i> This value is ignored in Windows 95. If REG_OPTION_VOLATILE is specified, the WinRegCreateKey function creates a nonvolatile key and returns ERROR_SUCCESS.</p> <p>"REG_OPTION_BACKUP_RESTORE" <i>Windows NT:</i> If this flag is set, the function ignores the samDesired parameter and attempts to open the key with the access required to backup or restore the key. If the calling thread has the SE_BACKUP_NAME privilege enabled, the key is opened with ACCESS_SYSTEM_SECURITY and KEY_READ access. If the calling thread has the SE_RESTORE_NAME privilege enabled, the key is opened with ACCESS_SYSTEM_SECURITY and KEY_WRITE access. If both privileges are enabled, the key has combined accesses for both privileges.</p> <p><i>Windows 95:</i> This flag is ignored. Windows 95 does not support security in its registry.</p>
disposition	<p>Name of the variable that receives whether key already existed. Windows returns one of the following:</p> <p>"REG_CREATED_NEW_KEY" "REG_OPENED_EXISTING_KEY"</p> <p>Enclose the name of the variable in quotes.</p>

Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey"
class= "KEY_ALL_ACCESS"
options= REG_OPTION_NON_VOLATILE"

w32err=WinRegCreateKey(hive, key, "myhandle",
    access,class, options, "orgstatus")

say w32err      * 0 = no error          */
say myhandle   /* t44 = the handle assigned by */
               /* Windows.                */
say orgstatus  /* REG_OPENED_EXISTING_KEY =   */
               /* key_already existed.     */
```

WinRegDeleteKey

Syntax WinRegDeleteKey(handle, key)

Description This function deletes a key from the registry, and destroys the handle. It returns the Win32 error code (0 if successful).

Parameters

Parameter	Usage
handle	Handle of parent key. It can be a handle returned from a prior call to WinRegCreateKey or WinRegOpenKey, or a predefined hive name, one of the following: "HKEY_CLASSES_ROOT" "HKEY_CURRENT_USER" "HKEY_LOCAL_MACHINE" "HKEY_USERS" "HKEY_PERFORMANCE_DATA" "HKEY_CURRENT_CONFIG" "HKEY_DYN_DATA"
key	Name of the key to delete.

Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey"

w32err=WinRegCreateKey(hive, key, myhandle,
    access, "orgstatus")
.
.
.
w32err=WinRegDeleteKey(hive, key)
say w32err /* 0 = key successfully deleted. */
```

WinRegDeleteValue

Syntax WinRegDeleteValue(handle, value)

Description Delete a value from a key. Returns the Win32 error code (0 if successful).

Parameters

Parameter	Usage
handle	Handle of parent key. It can be a handle returned from a prior call to WinRegCreateKey or WinRegOpenKey, or a predefined hive name, one of the following: "HKEY_CLASSES_ROOT" "HKEY_CURRENT_USER" "HKEY_LOCAL_MACHINE" "HKEY_USERS" "HKEY_PERFORMANCE_DATA" "HKEY_CURRENT_CONFIG" "HKEY_DYN_DATA"
value	Name of the value to delete.

Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"
tvalue="TestValue"

w32err=WinRegCreateKey(hive, key, "myhandle",
    access, "orgstatus")
say w32err
say myhandle
say orgstatus /* REG_OPENED_EXISTING_KEY */

w32err=WinRegDeleteValue(myhandle, tvalue)
say w32err /* 0 = TestValue successfully */
/* deleted. */
```


WinRegEnumKey

Syntax

```
call WinRegEnumKey handle,  
                    index,  
                    key  
                    [ , class  
                      , timestamp ]
```

Description Get sub-key information by index. Returns the Win32 error code (0 if successful). If the index supplied selects a non-existent key, the function returns error code 259.

Parameters

Parameter	Usage
handle	Handle of parent key. It can be a handle returned from a prior call to WinRegCreateKey or WinRegOpenKey, or a predefined hive name, one of the following: "HKEY_CLASSES_ROOT" "HKEY_CURRENT_USER" "HKEY_LOCAL_MACHINE" "HKEY_USERS" "HKEY_PERFORMANCE_DATA" "HKEY_CURRENT_CONFIG" "HKEY_DYN_DATA"
index	index is an integer that selects which sub-key to access. An index value of zero accesses the first subkey; an index value of 1 accesses the second sub-key, and so forth.
key	Name of the variable that receives the selected sub-key's name. Enclose the name in quotes.
class	Name of the variable that receives the class of the selected sub-key. Enclose the name in quotes.
timestamp	Name of the variable that receives the timestamp of the selected sub-key. Enclose the name in quotes.

Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey"
svalue="StringValue"
bvalue="BinaryValue"
keyindex=0
keyname=''
keyclass=''
timestamp=''

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err
say myhandle

w32err=WinRegEnumKey(myhandle, keyindex,
    "keyname", "keyclass", "timestamp")
say w32err /* 0=successful completion. */
say keyname /* keyname="TestSubKey1" */
say keyclass /* keyclass="" */
say timestamp /* timestamp="08/03/99 15:57:28" */
```


Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"
svalue="StringValue"
bvalue="BinaryValue"
keyindex=0
valname=''
valtype=''
valdata=''

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err
say myhandle

w32err=WinRegEnumValue(myhandle, keyindex,
    "valname", "valtype", "valdata")
say w32err      /* 0=successful completion */
say valname     /* valname="StringValue" */
say valtype     /* valtype="REG_SZ" */
say valdata     /* valdata="TestData" */
```


Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err      /* 0=successful open          */
say myhandle    /* myhandle="t44" (returned by */
                /* Windows)                       */
```


Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"
class=''
skcnt=0
vcnt=0
timestamp=''

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err

w32err=WinRegQueryInfoKey(myhandle, "class",
    "skcnt", "vcnt", "timestamp")
say w32err /* 0=Successful completion. */
say class /* "" - class returned by Windows*/
say skcnt /* 0=Number of sub-keys of */
/* TestKey\TestSubKey. */
say vcnt /* 2=Number of values in */
/* TestKey\TestSubKey. */
say timestamp /* "08/03/99 19:32:29" */
```


Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"
valname='StringValue'
valtype=''
valdata=''

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err
say myhandle

w32err=WinRegQueryValue(myhandle, valname,
    "valtype", "valdata")
say w32err /* 0=Successful completion */
say valname /* "StringValue" */
say valtype /* "REG_SZ" */
say valdata /* "Test Data" */
```

WinRegSetValue

Syntax WinRegSetValue(handle, name, type, data)

Description Set key value. Returns the Win32 error code (0 if successful).

Parameters

Parameter	Usage
handle	Handle of parent key. It can be a handle returned from a prior call to WinRegCreateKey or WinRegOpenKey, or a predefined hive name, one of the following: "HKEY_CLASSES_ROOT" "HKEY_CURRENT_USER" "HKEY_LOCAL_MACHINE" "HKEY_USERS" "HKEY_PERFORMANCE_DATA" "HKEY_CURRENT_CONFIG" "HKEY_DYN_DATA"
name	A variable containing the name of the value, within the key, to be set.
type	A variable that specifies the type of data contained in the value to be set; one of: "REG_DWORD" "REG_DWORD_LITTLE_ENDIAN" "REG_EXPAND_SZ" "REG_MULTI_SZ" "REG_SZ" "REG_LINK" "REG_RESOURCE_LIST" "REG_BINARY" "REG_NONE" "REG_DWORD_BIG_ENDIAN"
data	A variable containing the new data for the key value being set.

Example

Refer to the Registry Editor example on page 308.

```
hive="HKEY_CURRENT_USER"
access="KEY_ALL_ACCESS"
key="TestKey\TestSubKey"
valname='StringValue'
valtype=''
valdata='Live Data'

w32err=WinRegOpenKey(hive, key, "myhandle",
    access)
say w32err
say myhandle

w32err=WinRegSetValue(myhandle, valname,
    valtype,valdata)
say w32err /* 0=Successful completion */
say valname /* "StringValue" */
say valtype /* "REG_SZ" */
say valdata /* "Live Data" */
```

As a result, the `StringValue` value of

`HKEY_CURRENT_USER\TestKey\TestSubKey`

is set to **Live Data**.

A Message Summary

This appendix lists the messages that can be generated by CM REXX. Each message is followed by a brief description of its meaning.

CM REXX Messages

03 Program is unreadable

CM REXX was unable to locate the program you are trying to execute. A file by this name does not exist in the current working directory or in any directory on the current PATH.

04 Program interrupted

The system interrupted execution of the program at the user's request. If interrupts are not trapped by CALL or SIGNAL ON HALT, CM REXX immediately terminates execution when an interrupt occurs.

05 Machine resources exhausted

The CM REXX program was not able to obtain the system resources required to continue execution of this program. This can indicate insufficient memory, swap space, or other system resources.

06 Unmatched /* or quote

A comment or literal string was started but not completed. Comments require a matching '/* */' pair. Literal strings require matching single or double quotes. Since comments may span multiple lines, the absence of a closing '*/' can be reported at the end

of the program rather than on the line where the opening '/' appears. Unmatched quotes can be reported at the end of the line on which the opening quote appears.

07 WHEN or OTHERWISE expected

A SELECT construct must include at least one WHEN clause and possibly an OTHERWISE clause. If no WHEN clause is encountered, or if any other instruction is found, this error occurs. This can occur if the OTHERWISE clause has been omitted and none of the WHEN conditions are satisfied. It can also occur if a list of instructions follows a WHEN without the necessary DO and END.

08 Unexpected THEN or ELSE

A THEN or an ELSE was encountered in the program for which a matching IF or WHEN was not found. This can occur if the instruction following THEN is DO, and its matching END is omitted.

09 Unexpected WHEN or OTHERWISE

A WHEN or OTHERWISE keyword was encountered outside the scope of a SELECT construct. This can occur if a required WHEN or OTHERWISE is inadvertently enclosed in a DO-END construct (often the result of a missing END somewhere else). It can also occur if an attempt is made to branch to the WHEN or OTHERWISE clause using SIGNAL.

10 Unexpected WHEN or OTHERWISE

An END was encountered in the program for which a matching DO or SELECT was not found. This can occur if the END is badly located so that it does not match the DO or SELECT for which it was intended. Also, this error can occur in the case of heavily nested DOs when too many ENDS are provided. Including the name of the DO loop control variable on the corresponding END clause is a good technique to avoid or identify this type of error.

This error can also occur if END immediately follows THEN or ELSE. Still another possible cause of this error is an attempt to branch into a DO loop using SIGNAL. In this case, the DO instruction will never have been executed and the END will be unexpected.

11 Control stack full

An implementation-specific limit on levels of nesting of control structures has been exceeded. This can occur with deeply nested DO-END or IF-THEN-ELSE constructs. It can also occur if an INTERPRET instruction is looping or if a recursive subroutine or internal function does not terminate correctly, resulting in an infinite loop.

12 Clause too long

An implementation-specific limit on the length of a clause has been exceeded.

13 Invalid character in program

A character appears in the program, outside of a literal string, that is not a blank or one of the following characters:

```
A-Z, a-z, 0-9  
@@# .?!_ $ & * ( ) - + = ^ \  
' " ; : , % / < > |
```

This can occur if the program contains accented or other national language-specific characters not specifically permitted by the implementation.

14 Incomplete DO/IF/SELECT

At the end of the program, the language processor has detected a DO or SELECT instruction without a matching END or an IF instruction that is not followed by a THEN clause. Including the name of the control variable on the corresponding END clause is a good technique for avoiding or identifying this type of error.

15 Invalid hexadecimal constant

Hexadecimal constants can contain only the digits 0-9 and the letters a-f and A-F. They cannot have leading or trailing blanks, and embedded blanks can occur only at byte boundaries (between pairs of hexadecimal digits).

Binary strings can contain only the digits 0 and 1. They cannot have leading or trailing blanks, and embedded blanks can occur only between groups of four binary digits.

This error may occur if the character x or b immediately follows a literal string - that is, if abuttal concatenation is used to append an x or b to the end of a literal string. In this case, it is necessary to use the concatenation operator to distinguish concatenation from an attempt to specify a hexadecimal or binary string.

16 Label not found

A SIGNAL instruction has been executed or a trapped condition has been raised, and the specified label is not found in the program. For trapped conditions, if the SIGNAL ON instruction does not include the NAME keyword, a label matching the name of the condition must exist.

17 Unexpected procedure

A PROCEDURE instruction was encountered that was not the first instruction after a CALL or function invocation. If present, the PROCEDURE instruction must be the first instruction executed after a subroutine is called or a function invoked. This error can occur if a program falls through into an internal routine that includes a PROCEDURE instruction.

18 THEN expected

All IF and WHEN clauses must be followed by a THEN clause. Another clause was encountered at the point where a THEN was expected to be.

19 String or symbol expected

The first token following a CALL or SIGNAL instruction must be a literal string or a symbol. The string or symbol was omitted or something else, such as an operator, was found.

20 Symbol expected

In an instruction where a symbol is required, the symbol was omitted or some other token was found.

21 Invalid data on end of clause

A keyword or instruction that has no operand (such as SELECT or NOP) was followed by something other than a comment.

22 Invalid character string

A literal string contains one or more characters that are not supported in this implementation.

24 Invalid TRACE request

The first character of the option specified on the TRACE instruction does not match one of the valid TRACE settings. Refer to the *Chapter 4: Instructions* for a list of valid TRACE settings.

25 Invalid sub-keyword found

An unexpected token was in the position where an instruction expected a specific keyword. This can occur if the token following NUMERIC is not DIGITS, FORM, or FUZZ. It can also occur with CALL or SIGNAL ON *condition* if the token following *condition* is not NAME.

26 Invalid whole number

One of the following did not evaluate to a whole number, or its value is greater than the implementation limit:

- The repetitor in a DO instruction.
- The FOR expression in a DO instruction values specified for DIGITS or FUZZ in a NUMERIC instruction.
- A positional pattern in a parsing template.
- A number used as a trace setting in the TRACE instruction.
- The exponent (right hand operator) of the power operator (**).

This error also occurs when the result of an integer divide (%) is not a whole number or when the specific value is not permitted in the context where it appears (such as a negative value for a DO repetitor).

27 Invalid DO syntax

A syntax error was found in the DO instruction. This can occur when a keyword such as TO appears without a control variable, or when such a keyword appears more than once.

28 Invalid LEAVE or ITERATE

A LEAVE or ITERATE instruction was unexpectedly encountered during execution. Either no loop is active, or the control variable name specified on the instruction does not

match that of an active loop. This can occur when attempting to use SIGNAL to branch into, or within, a loop.

29 Environment name too long

The host command environment specified on the ADDRESS instruction is longer than permitted by the operating system.

30 Name or string too long

The length of the name or string was greater than the implementation maximum.

31 Name starts with number or "."

To avoid confusion with numeric constants, a value cannot be assigned to a variable whose name begins with a number or a period.

33 Invalid expression result

The result of an expression is invalid in the context where it occurs. This can occur if the value for NUMERIC FUZZ is greater than that for NUMERIC DIGITS.

34 Logical value not 0 or 1

Any term operated on by a logical operator (^ \ | & &) must evaluate to 0 or 1. Likewise, the expression in an IF, WHEN, DO, WHILE, or UNTIL clause must evaluate to 0 or 1.

35 Invalid expression

There is an error in the syntax of an expression. This can be due to the absence or misplacement of an operator, the placement of two operators adjacent to each other, or the absence of an expression where one was expected. This can occur when an operator character is present in what is intended to be a literal string, but the string is not enclosed in quotes.

36 Unmatched "(" in expression

There are more left parentheses than right parentheses in an expression.

37 Unmatched ",", " or ")" in expression

Either a comma was found outside of a function call, or there are too many right parentheses in an expression.

38 Invalid template or pattern

One of the following errors has been detected:

- A special character (such as "*"), which is not allowed, was found in a parsing template.
- The syntax of a variable pattern is incorrect; this can occur if no symbol follows a left parenthesis or if a parenthesis is missing.
- The WITH is missing in a PARSE VALUE instruction.

39 Evaluation stack overflow

An expression is too complex to be evaluated within implementation-specific limits.

40 Incorrect call to routine

Arguments passed to a routine are of the wrong type, or the number of arguments passed to the routine exceeded an implementation-specific maximum. This can also occur if the routine is not compatible with the CM REXX language.

41 Bad arithmetic conversion

One of the terms in an arithmetic expression is not a valid number, or its exponent exceeds the implementation-specific limit.

42 Arithmetic overflow/underflow

The result of an arithmetic operation requires an exponent outside the range supported by the implementation. This can occur during an attempt to divide by zero.

43 Routine not found

A subroutine that has been called, or a function that has been invoked, cannot be found. It is neither an internal or external routine nor the name of a built-in function. This can be caused by the result of a typographical error, or the presence of a literal string or symbol immediately adjacent to a left parenthesis.

44 Function did not return data

An external function was invoked but it did not return a value for use within the expression. All functions must return a value.

45 No data specified on function RETURN

A routine was called as a function, but the RETURN instruction did not specify a value to be returned. All functions must return a value.

46 Invalid variable reference

The syntax of a variable reference is incorrect. The right parenthesis, which must immediately follow the variable name, is missing.

48 Failure in system service

An operating system service called by CM REXX resulted in an error. As a result, execution of the program terminated.

49 Interpretation error

A CM REXX internal error occurred during execution of the program. Please contact HP Technical Support for assistance.

B Programming Hints

This appendix is designed to help you avoid common pitfalls when using CM REXX to write programs. The more common programming mistakes are identified, and the correct CM REXX usage is shown.

Invoking a Built-in Function Like an Instruction

When a built-in function call is the only clause on a line, the function returns a value.

Example

```
LINEOUT('myfile', 'new data')
```

This value is then passed to the external environment where it is interpreted as a command. This usually results in an "Invalid command" message from the operating system. To avoid this, use `CALL` to invoke the function.

Failure to Use Commas with `CALL` and `PARSE ARG`

With `CALL`

When you `CALL` a routine or function, the arguments of the called routine must be separated by commas.

Example 1

The following example passes two arguments to the routine SUB.

```
CALL SUB X, Y
```

Example 2

This example passes one argument to SUB.

```
CALL SUB X Y
```

This argument is the result of concatenating X and Y.

With PARSE ARG

Commas must also be used between arguments in the template of a PARSE ARG instruction.

Example 3

The following example assigns all of the first argument to a1 and all of the second argument to a2.

```
PARSE ARG a1, a2
```


Example 4

The next example assigns the first word of the first argument to `a1` and the rest of the first argument to `a2`.

```
PARSE ARG a1 a2
```

Note that any arguments supplied on the program invocation command line are treated as one string by `PARSE ARG` or `ARG`.

Incorrect Use of Continuation

The statement:

```
x = min(1, 2, 3,  
       4, 5)
```

will fail with *Error 41: Bad arithmetic conversion*, because the comma after the 3 in the first line is treated as a continuation character, resulting in a function invocation that looks like:

```
x = min(1, 2, 3 4, 5)
```

The arguments to the `MIN` built-in function must be separated by commas. The correct way to write such a continued clause is to provide an additional comma for continuation on the first line as in:

```
x = min(1, 2, 3,,  
       4, 5)
```

Incorrect CALL Syntax

The correct syntax for calling a routine with arguments is:

```
CALL SUB X, Y
```

If you use the `CALL` instruction, it is not proper to enclose the arguments in parentheses. Enclose the arguments in parentheses when you invoke a routine as a function, as in:

```
x = SUB(X,Y)
```

Failure to Enclose Command Arguments Within Quotes

Consider the example of attempting to set the REXX environment variable:

```
dir = 'c:\mydir'  
putenv(REXX=dir)
```

The function argument includes an operator and is therefore treated as an expression that must be evaluated before it is used in the function. The expression is treated as a logical comparison and returns the value 0 (FALSE). The result is passed to the `PUTENV` function; but since 0 is not a valid command to set an environment variable, `PUTENV` appears to have no effect. The correct way to write the sample above is shown below:

```
dir = 'c:\mydir'  
rc = putenv('REXX='dir)
```

Similar pitfalls exist in the use of host commands that include strings that might be interpreted as operators. In XEDIT macros, for example, the `EXTRACT` command requires the use of the forward slash character as in:

```
extract /curline
```

If this command is not enclosed in quotes, CM REXX sees the clause as an attempt to divide the value of the symbol `extract` by the value of the symbol `curline`. Since such variables would not normally be initialized to a numeric value in an editor macro, execution of the clause results in *Error 41: Bad arithmetic conversion*. If the clause is enclosed in quotes, it is treated by CM REXX as a literal string and is automatically passed to the host command environment (in this case, XEDIT) for execution.

Failure to Close a File

Any I/O operation to a file (`CHARIN`, `CHAROUT`, `CHARS`, `LINEIN`, `LINEOUT`, `LINES`, or `EXECIO`) can leave the file in an open state. Therefore, it might be necessary to close the file with `CHAROUT`, `LINEOUT`, or `STREAM` before subsequent attempts to read from or write to the file.

C System Limitations

Very few implementation-specific limitations exist in CM REXX. These limitations are documented in this appendix.

Implementation-Specific Limits

Description of Limitation	Limitation
Maximum length of a string.	1 billion characters
Maximum length of a symbol or variable name.	1 billion characters
Maximum number of variables in a program.	60 thousand variables
Maximum setting of numeric digits.	1000
Maximum length of a host command environment name.	16 characters
In general, all internal maximums are equivalent to 1 billion bytes.	It is likely that your system memory will be exceeded before you approach these limits.

Bibliography

Amiga Programmers Guide to AREXX, Eric Giguere, Commodore-Amiga, Inc., 1991.

Application Development Using OS/2 REXX, Anthony Rudd, John Wiley and Sons, Inc., 1994.

Mastering OS/2 REXX, Gabriel F. Gargiulo, John Wiley and Sons, Inc., 1994.

Modern Programming Using REXX, Bob O'Hara and Dave Gomberg, Prentice Hall, 1988.

OS/2 2.1 REXX Handbook - Basics, Applications, and Tips, Hallett German, Van Nostrand Reinhold, 1994.

Practical Usage of REXX, Anthony Rudd, Ellis Horwood Limited, 1990.

Programming in REXX, Charles Daney, McGraw-Hill, Inc., 1992.

REXX Handbook, edited by Gabe Goldberg and Phil Smith, McGraw-Hill, Inc., 1992.

REXX in the TSO Environment, Gabriel F. Gargiulo, QED Information Sciences, Inc., 1990.

REXX Tools and Techniques, Barry Nirmal, QED Publishing Group, 1993.

REXX: Advanced Techniques for Programmers, Peter Kiesel, McGraw-Hill, Inc., 1993.

The AREXX Cookbook, Merrill Callaway, Whitestone, 1992.

The REXX Language: A Practical Approach to Programming, M. F. Cowlshaw, Prentice Hall, Second Edition, 1990.

Using ARexx on the Amiga, Chris Zamara and Nick Sullivan, Abacus, 1991.

Note

In addition to these references, published proceedings of the annual REXX Symposium are available from the Stanford Linear Accelerator Center.

Index

A

A (All) trace option, 110
ABBREV function, 124
ABS function, 126
absolute positional pattern, 85
ADDRESS [VALUE] expr2 parameter, 48
ADDRESS function, 127
ADDRESS instruction, 43, 47, 48
ADDRESS settings, 56
after parameter, 171
agent methods, 250
ARG function, 128
ARG instruction, 47, 53
ARG parameter, 79
arithmetic operators, 30
arithmetic overflow/underflow message, 333
assignment instructions, 26

B

B2X function, 133
bad arithmetic conversion message, 332
before parameter, 171
binary strings, 25
BITAND function, 130

BITOR function, 131
BITXOR function, 132
built-in functions, 32, 120
 invoking, 335
BY parameter, 61

C

C (Commands) trace option, 110
C2D function, 149
C2X function, 150
CALL instruction, 34, 47, 55
CENTER function, 134
char parameter, 215
char_list parameter, 233
CHARIN
 function, 136
 operation, 35
CHAROUT
 function, 138
 operation, 35
CHARS
 function, 140
 operation, 35
CHDIR function, 141
clause
 definition, 24
 types of, 26

clause too long message, 327
CM REXX executable, 38
CM REXX extensions. *See* extensions
CM REXX functions. *See* functions
CM REXX programs, coding, 42
CMD host command, 43
command instructions, 26
command output, 43
command parameter, 198
command_line parameter, 264, 282, 283
comparative operators, 30
 normal, 31
 strict, 31
COMPARE function, 142
compound symbol
 definition, 27
 tail, 27
concatenation operators, 30
CONDITION function, 144
CONDITION information, 56
condition parameter, 55
condition traps, 34
 and CALL, 34
 and SIGNAL, 34
 ERROR, 34
 FAILURE, 34
 HALT, 34
 LOSTDIGITS, 34
 NOTREADY, 34
 NOVALUE, 34
 SYNTAX, 34
CONDITION traps, 56

constant symbol, definition, 28
Control stack full message, 327
DATE formats, 155
COPIES function, 146
copyright notices, 2
count parameter, 187
CUSERID function, 148
customer support, 7

D

D2C function, 164
D2X function, 165
DATATYPE function, 152
DATE function, 155
date_string parameter, 155
DELSTR function, 159
DELWORD function, 161
DIGITS function, 163
DIGITS parameter, 76
directory parameter, 141, 287, 290
DO instruction, 47, 60
document changes, 4
documentation updates, 4
DROP instruction, 47, 65

E

E (Error) trace option, 110
EDM REXX language. *See* language structure
EDMADD extension, 259
EDMATTR extension, 261

- EDMBLD extension, 263
- EDMCMD extension, 264
- EDMDELHEAP extension, 265
- EDMDELVAR extension, 266
- EDMFREE extension, 267
- EDMGET extension, 268
- EDMGETV extension, 270
- EDMLOC extension, 271
- EDMRST extension, 272
- EDMSET extension, 273
- EDMSORT extension, 275
- EDMWIN, 43
- EDMWIN host command, 43
- Elapsed time clocks, 57
- ELSE parameter, 69
- end parameter, 242
- environment name too long message, 331
- environment parameter, 48
- ERROR condition, 56, 107
- ERROR condition trap, 34
- ERROR FAILURE condition, 144
- ERRORTXT function, 166
- evaluation stack overflow message, 332
- executing methods, overview, 38
- EXIT instruction, 47, 67
- EXPOSE parameter, 89
- expp parameter, 171
- expr parameter, 55
- expr1 parameter, 48, 76

- expr3 parameter, 76
- expression parameter, 67, 69, 72, 95, 97, 99, 102, 104
- expr1 parameter, 60
- exprn parameter, 61
- expt parameter, 171
- extensions
 - EDMADD, 259
 - EDMATTR, 261
 - EDMBLD, 263
 - EDMCMD, 264
 - EDMDELHEAP, 265
 - EDMDELVAR, 266
 - EDMFREE, 267
 - EDMGET, 268
 - EDMGETV, 270
 - EDMLOC, 271
 - EDMRST, 272
 - EDMSET, 273
 - EDMSORT, 275
 - function calls, 252
 - GET_CHILD_OBJECT, 276
 - LOAD_CHILDREN, 277
 - NOWAIT, 282
 - NVDOBJECTS, 283
 - NVDPATHS, 285
 - overview, 250
 - RADGET, 287
 - RADSET, 290
 - return values, 252
 - RXXCommandKill, 291
 - RXXCommandSpawn, 292
 - RXXCommandWait, 293
 - RXXOSEndOfLineString, 294
 - RXXOSEnvironmentSeparator, 295
 - RXXOSName, 296
 - RXXOSPathSeparator, 297
 - RXXSleep, 298

- WinExpandEnvironmentStrings, 301
- WinGetVersion, 302
- WinMessageBox, 299

external functions, 32, 42

F

- F (Failure) trace option, 110
- FAILURE condition, 56, 107
- FAILURE condition trap, 34
- failure in system service message, 333
- filename parameter, 261, 271
- FIND function, 168
- FOR parameter, 61
- FOREVER parameter, 61
- FORM function, 170
- FORM parameter, 76
- FORMAT function, 171
- FULLSCR parameter, 264
- function calls, 29, 252
- function did not return data message, 333
- functions, 32
 - ABBREV, 124
 - ABS, 126
 - ADDRESS, 127
 - ARG, 128
 - B2X, 133
 - BITAND, 130
 - BITOR, 131
 - BITXOR, 132
 - built-in, 32, 120
 - C2D, 149
 - C2X, 150
 - CENTER, 134

- CHARIN, 136
- CHAROUT, 138
- CHARS, 140
- CHDIR, 141
- COMPARE, 142
- CONDITION, 144
- COPIES, 146
- CUSERID, 148
- D2C, 164
- D2X, 165
- DATATYPE, 152
- DATE, 155
- DELSTR, 159
- DELWORD, 161
- DIGITS, 163
- ERRORTXT, 166
- external, 32
- FIND, 168
- FORM, 170
- FORMAT, 171
- FUZZ, 175
- general rules, 122
- GETCWD, 176
- GETENV, 177
- INDEX, 178
- INSERT, 180
- internal, 32
- JUSTIFY, 181
- LASTPOS, 182
- LEFT, 184
- LENGTH, 186
- LINEIN, 187
- LINEOUT, 189
- LINES, 191
- LOWER, 193
- MAX, 194
- MIN, 195
- OVERLAY, 196
- overview, 120

POPEN, 198
POS, 200
PUTENV, 202
QUEUED, 203
RANDOM, 204
REVERSE, 206
RIGHT, 207
SIGN, 209
SOURCELINE, 210
SPACE, 211
STREAM, 213
STRIP, 215
SUBSTR, 217
SUBWORD, 219
SYMBOL, 220
TIME, 222
TRACE, 225
TRANSLATE, 226
TRUNC, 228
UPPER, 229
USERID, 230
VALUE, 231
VERIFY, 233
WORD, 236
WORDINDEX, 237
WORDLENGTH, 238
WORDPOS, 239
WORDS, 241
X2B, 244
X2C, 245
X2D, 247
XRANGE, 242

FUZZ function, 175
FUZZ parameter, 76

G

GET_CHILD_OBJECT extension, 276
GETCWD function, 176

Index

GETENV function, 177

H

HALT condition, 56, 107, 144
HALT condition trap, 34
heap_number parameter, 268, 287
heap_size parameter, 263
hexadecimal strings, 24
HIDE parameter, 264
host command, executing, 42

I

I (Intermediates) trace option, 110

I/O operations, 35
 CHARIN, 35
 CHAROUT, 35
 CHARS, 35
 LINEIN, 35
 LINEOUT, 35
 LINES, 35
 PARSE LINEIN, 35
 PARSE PULL, 35
 PULL, 35
 PUSH, 35
 QUEUE, 35
 QUEUED, 35
 SAY, 35
 STREAM, 35

IF instruction, 47, 69

in_option parameter, 155

in_tbl parameter, 226

incomplete DO/IF/SELECT message, 328

incorrect call to routine message, 332

INDEX function, 178

info parameter, 124

information parameter, 124

INSERT function, 180

instruction parameter, 69, 104

instruction, definition, 26

instructions

- ADDRESS, 48
- ARG, 53
- CALL, 55
- DO, 60
- DROP, 65
- EXIT, 67
- IF, 69
- INTERPRET, 72
- ITERATE, 73
- LEAVE, 74
- NOP, 75
- NUMERIC, 76
- overview, 46
- PARSE, 79
- PROCEDURE, 89
- PULL, 93
- PUSH, 95
- QUEUE, 97
- RETURN, 99
- SAY, 102
- SELECT, 104
- SIGNAL, 106
- TRACE, 110
- UPPER, 117

instructions, types of

- assignment, 26
- command, 26
- keyword, 26

internal functions, 32

INTERPRET instruction, 47, 72

interpretation error message, 334

invalid character in program message, 327

invalid character string message, 329

invalid data on end of clause message, 329

invalid DO syntax message, 330

invalid expression message, 331

invalid expression result message, 331

invalid hexadecimal constant message, 328

invalid LEAVE or ITERATE message, 330

invalid sub-keyword found message, 330

invalid template or pattern message, 332

invalid TRACE request message, 329

invalid variable reference message, 333

invalid whole number message, 330

ITERATE instruction, 73

J

JUSTIFY function, 181

K

keyword instructions, 26

L

L (Labels) trace option, 110

label not found message, 328

label parameter, 106

label, definition, 26

language structure

- clauses, 24

- condition traps, 34
- expressions, 29
- functions, 32
- input/output operation, 35
- parsing, 36
- special variables, 33
- symbols, 27

LASTPOS function, 182

LEAVE instruction, 47, 74

LEFT function, 184

legal notices, 2

- copyright, 2
- restricted rights, 2
- trademark, 2
- warranty, 2

LENGTH function, 186

length parameter, 124, 134, 136, 159, 161, 180, 181, 196, 217, 219

line parameter, 187, 189

line prefixes, 111

LINEIN

- function, 187
- operation, 35

LINEIN parameter, 79

LINEOUT

- function, 189
- operation, 35

LINES

- function, 191
- operation, 35

literal strings, 24, 29

LOAD_CHILDREN extension, 277

logical operators, 30

logical value not 0 or 1 message, 331

LOSTDIGITS condition trap, 34

LOWER function, 193

M

machine resources exhausted message, 325

MAX function, 194

max parameter, 204

messages

- arithmetic overflow/underflow, 333
- bad arithmetic conversion, 332
- clause too long, 327
- control stack full, 327
- environment name too long, 331
- evaluation stack overflow, 332
- failure in system service, 333
- function did not return data, 333
- incomplete DO/IF/SELECT, 328
- incorrect call to routine, 332
- interpretation error, 334
- invalid character in program, 327
- invalid character string, 329
- invalid data on end of clause, 329
- invalid DO syntax, 330
- invalid expression, 331
- invalid expression result, 331
- invalid hexadecimal constant, 328
- invalid LEAVE or ITERATE, 330
- invalid sub-keyword found, 330
- invalid template or pattern, 332
- invalid TRACE request, 329
- invalid variable reference, 333
- invalid whole number, 330
- label not found, 328
- logical value not 0 or 1, 331
- machine resources exhausted, 325
- name or string too long, 331
- name starts with number or ".", 331

- no data specified on function RETURN, 333
- program interrupted, 325
- program is unreadable, 325
- routine not found, 333
- string or symbol expected, 329
- symbol expected, 329
- THEN expected, 329
- unexpected procedure, 328
- unexpected THEN or ELSE, 326
- unexpected WHEN or OTHERWISE, 326, 327
- unmatched "(" in expression, 332
- unmatched ", " or ")" in expression, 332
- unmatched /* or quote, 325
- WHEN or OTHERWISE expected, 326
- messages generated by CM REXX, 325
- methods, executing, 38
- MIN function, 195
- min parameter, 204
- modifier parameter, 264

N

- N (Normal) trace option, 110
- n parameter, 128, 146, 149, 159, 161, 164, 165, 166, 180, 184, 196, 207, 210, 211, 217, 219, 228, 236, 237, 238, 247
- name or string too long message, 331
- name parameter, 55, 61, 73, 74, 136, 138, 140, 187, 189, 191, 213, 220, 231
- name starts with number or "." message, 331
- NEWPANEL.LOG, 40
- no data specified on function RETURN message, 333
- NOLOAD parameter, 268, 287
- NOP instruction, 47, 75
- NOTREADY condition, 56, 107, 144

- NOTREADY condition trap, 34
- NOVALUE condition, 107, 144
- NOVALUE condition trap, 34
- NOWAIT extension, 282
- NOWAIT parameter, 264
- null clause, definition, 26
- num parameter, 171
- number parameter, 126, 194, 195, 209, 228
- NUMERIC instruction, 47, 76
- NUMERIC settings, 56
- NVDOBJECTS extension, 283
- NVDPATHS extension, 285



- O (Off) trace option, 110
- object_name parameter, 259, 263, 265, 266, 267, 268, 270, 272, 273, 275, 277, 287, 290, 298
- OFF parameter, 55
- ON parameter, 55
- operation parameter, 213
- operator tokens, 25
- operators, 29, 30
- option parameter, 110, 128, 144, 215, 225, 233
- optioneter*, 198
- OTHERWISE parameter, 104
- out_option parameter, 155, 222
- out_tbl parameter, 226
- OVERLAY function, 196

P

- pad parameter, 130, 131, 132, 134, 142, 180, 181, 184, 196, 207, 211, 217, 226
- PARSE instruction, 47, 79
- PARSE LINEIN operation, 35
- PARSE PULL operation, 35
- parsing, 36
 - by patterns, 83
 - by position, 84
 - by words, 82
 - positional patterns, 84
 - summary, 88
 - templates, 81
 - with placeholders, 87
- PINSCOMP.EDM, 40
- PINSCOMP.LOG, 40
- PNLREXX.LOG, 40
- POPEN function, 198
- POS function, 200
- prefixes, 111
- PROCEDURE instruction, 47, 89
- PROCEDURE parameter, 89
- program interrupted message, 325
- program is unreadable message, 325
- programming hints, 335
- PULL instruction, 47, 93
- PULL operation, 35
- PULL parameter, 79
- PUSH instruction, 47, 95
- PUSH operation, 35
- PUTENV function, 202

Index

Q

- QUEUE
 - instruction, 47, 97
 - operation, 35
- QUEUED
 - function, 203
 - operation, 35

R

- R (Results) trace option, 110
- RADGET extension, 287
- Radia Agent REXX methods, 250
- RADPNLWR
 - functions, 38
 - invoking, 38
- RADPNLWR executable, 38
- RADPNLWR log files, 40
- RADREXXW.EXE, 37
- RADSET extension, 290
- RANDOM function, 204
- RC variable, 33
- redirect*, 49
- registry manipulation functions
 - WinRegCloseKey, 308
 - WinRegCreateKey, 309
 - WinRegDeleteKey, 311
 - WinRegDeleteValue, 312
 - WinRegEnumKey, 313
 - WinRegEnumValue, 315
 - WinRegOpenKey, 317
 - WinRegQueryInfoKey, 319
 - WinRegQueryValue, 321
 - WinRegSetValue, 323

registry, manipulating, 305
relative positional pattern, 85
restricted rights legend, 2
RESULT variable, 33
RETURN instruction, 47, 99
REVERSE function, 206
REXX instructions. *See* instructions
REXX methods, executing from Windows, 41
RIGHT function, 207
routine not found message, 333
RXXCommandKill extension, 291
RXXCommandSpawn extension, 292
RXXCommandWait extension, 293
RXXOSEndOfLineString extension, 294
RXXOSEnvironmentSeparator extension, 295
RXXOSName extension, 296
RXXOSPathSeparator extension, 297
RXXSleep extension, 298

S

SAY

instruction, 47, 102
operation, 35

seed parameter, 204

SELECT instruction, 47, 104

SELECT parameter, 104

SHOW parameter, 264

SIGL variable, 33

SIGN function, 209

SIGNAL instruction, 34, 47, 106

simple symbol, definition, 28

SOURCE parameter, 80

SOURCELINE function, 210

SPACE function, 211

special characters, 25

special variables, 33

Standard Error Stream, 43

Standard Output Stream, 43

start parameter, 136, 138, 178, 182, 200, 233, 239, 242

STDERROR, 43

STDIN, definition, 36

STDOUT, 43
definition, 36

stem, definition, 28

STREAM

function, 213
operation, 35

string or symbol expected message, 329

string parameter, 133, 134, 138, 146, 149, 150, 152, 159, 161, 177, 181, 184, 186, 189, 193, 202, 206, 207, 211, 215, 217, 219, 226, 229, 233, 236, 237, 238, 241, 244, 245, 247, 301

string1 parameter, 130, 131, 132, 142, 168, 178, 180, 182, 196, 200, 239

string2 parameter, 130, 131, 132, 142, 168, 178, 180, 182, 196, 200, 239

strings

binary, 25
hexadecimal, 24
literal, 29

STRIP function, 215

strmcmd parameter, 214

- sub-expression, 29
- subroutines, 42
- SUBSTR function, 217
- SUBWORD function, 219
- support, 7
- symbol, 25, 29
 - definition, 27
 - tokens, 25
- symbol expected message, 329
- SYMBOL function, 220
- SYNTAX condition, 107, 144
- SYNTAX condition trap, 34
- system limitations, 339

T

- tail, definition, 27
- technical support, 7
- template parameter, 53, 79, 93
- THEN expected message, 329
- THEN parameter, 69, 104
- TIME formats, converting, 222
- TIME function, 222
- TO parameter, 61
- tokens
 - binary, 25
 - hexadecimal, 24
 - literal string, 24
 - operator, 25
 - special characters, 25
 - symbol, 25
- TRACE function, 225

- TRACE instruction, 47, 110
- TRACE settings, 56
- trademark notices, 2
- TRANSLATE function, 226
- troubleshooting, 335
 - CALL and PARSE ARG, 335
 - CALL syntax, 337
 - closing a file, 338
 - command arguments, 338
 - continuation, 337
 - invoking a built-in function, 335
- TRUNC function, 228
- type parameter, 152

U

- unexpected procedure message, 328
- unexpected THEN or ELSE message, 326
- unexpected WHEN or OTHERWISE message, 326, 327
- unmatched "(" in expression message, 332
- unmatched "," or ")" in expression message, 332
- unmatched /* or quote message, 325
- UNTIL parameter, 60
- updates to doc, 4
- UPPER function, 229
- UPPER instruction, 47, 117
- USERID function, 230

V

- VALUE function, 231
- VALUE parameter, 80
- VAR parameter, 80

var_list parameter, 117

variable parameter, 266, 275

variables

RC, 33

RESULT, 33

SIGL, 33

varlist parameter, 65, 89

VERIFY function, 233

VERSION parameter, 81

W

WAIT parameter, 264

warranty, 2

WHEN or OTHERWISE expected message, 326

when_list parameter, 104

WHILE parameter, 60

whole-number parameter, 164, 165

Windows registry, manipulating, 305

WinExpandEnvironmentStrings extension, 301

WinGetVersion extension, 302

WinMessageBox extension, 299

WinRegCloseKey function, 308

WinRegCreateKey function, 309

WinRegDeleteKey function, 311

WinRegDeleteValue function, 312

WinRegEnumKey function, 313

WinRegEnumValue function, 315

WinRegOpenKey function, 317

WinRegQueryInfoKey function, 319

WinRegQueryValue function, 321

WinRegSetValue function, 323

WITH, 49

WORD function, 236

WORDINDEX function, 237

WORDLENGTH function, 238

WORDPOS function, 239

WORDS function, 241

X

X2B function, 244

X2C function, 245

X2D function, 247

XRANGE function, 242

Z

ZMASTER variables

ZPANEL, 39

ZPCONT, 39

ZPHEAPNO, 39

ZPREXEC, 39

ZPSEL, 40

ZPANEL variable, 39

ZPCONT variable, 39

ZPHEAPNO variable, 39

ZPREXEC variable, 39

ZPSEL variable, 40