

uni-REXX

Reference Manual

525 Capital Drive
Lake Zurich, Illinois 60047
Voice: (800) 228-0255
FAX: (847) 540-5629
Email: tech@wrkgrp.com
WWW: <http://www.wrkgrp.com/>

Preface

This manual describes the features of the uni-REXX® software product, through release 3.00. uni-REXX is a UNIX implementation of the REXX language, as defined by M. F. Cowlshaw in *The REXX Language, A Practical Approach to Programming* (Second Edition, 1990) and ANSI standard X3.274:1996, *Programming Language REXX*. The Workstation Group is an active participant in the work of ANSI Committee X3J18.

This manual also refers to uni-XEDIT® and uni-SPF™. uni-XEDIT is a UNIX full screen text editing program patterned after IBM's VM/CMS System Product Editor. uni-SPF is a panel driven productivity tool for the UNIX environment patterned after IBM's ISPF Program Product. Both are available from The Workstation Group.

Reproduction of this manual without the written consent of The Workstation Group is strictly prohibited.

UNIX is a trademark licensed in the United States and other countries through X-Open Company, Ltd.

IBM is a trademark of International Business Machines Corporation.

© Copyright 1992-2002, The Workstation Group, Ltd.
All Rights Reserved.

RESTRICTED RIGHTS: Use, duplication, or disclosure by the U.S. Government is subject to restrictions as set forth in subparagraph (c) (1) (ii) of the Rights in Technical Data and Computer Software clause at DFARS 252.227-7013 or subparagraphs (c) (1) and (2) of Commercial Computer Software – Restricted Rights at 48 CFR 52.227-19, as applicable. Contractor/Manufacturer is The Workstation Group Ltd., 1900 North Roselle Road Suite 408, Schaumburg, Illinois 60195.

Document Number: RXV2-9

TABLE OF CONTENTS

Preface	iii
Chapter 1: Introduction	1
Documentation Conventions	2
Chapter 2: Language Features	5
Clauses	5
Symbols	8
Expressions.....	10
Functions	13
Special Variables.....	14
Condition Traps.....	14
Input/Output	15
Parsing.....	17
Language Extensions.....	18
Chapter 3: Operation	21
Program Names	21
Program Execution.....	22
The rxx Command.....	24
The uni-REXX Compiler	27
The rxc Command.....	28
External Functions and Subroutines.....	33
External Data Queue	33
Host Command Execution	34
Chapter 4: Instructions	37
ADDRESS.....	39

ARG	44
CALL	46
DO	53
DROP	59
EXIT	61
IF	63
INTERPRET	66
ITERATE	67
LEAVE	68
NOP	69
NUMERIC	70
OPTIONS	73
PARSE	74
PROCEDURE	85
PULL	89
PUSH	91
QUEUE	92
RETURN	93
SAY	95
SELECT	96
SIGNAL	99
TRACE	103
UPPER	110

Chapter 5: Built-In Functions111

ABBREV	114
ABS	115
ADDRESS	115
ARG	117
BITAND	118
BITOR	119
BITXOR	120
B2X	121
CENTER	122
CHANGESTR	123
CHARIN	124
CHAROUT	126
CHARS	128
CHDIR	129
COMPARE	130
CONDITION	131
COUNTSTR	133
COPIES	133

CUSERID.....	134
C2D	135
C2X	135
DATATYPE.....	136
DATE	138
DELSTR.....	142
DELWORD.....	143
DIGITS.....	144
D2C	145
D2X.....	146
ERRORTXT	146
FIND.....	147
FORM.....	149
FORMAT	150
FUZZ.....	154
GETCWD.....	155
GETENV	156
INDEX	157
INSERT.....	158
JUSTIFY	159
LASTPOS.....	159
LEFT	160
LENGTH.....	162
LINEIN.....	163
LINEOUT.....	165
LINES.....	167
LOWER.....	168
MAX.....	169
MIN	170
OVERLAY.....	170
POPEN	172
POS.....	173
PUTENV	174
QUALIFY	175
QUEUED	176
RANDOM	177
REVERSE.....	178
RIGHT.....	178
SIGN.....	179
SOURCELINE	180
SPACE	181
STREAM.....	183
STRIP.....	185
SUBSTR.....	186

SUBWORD.....	187
SYMBOL.....	188
TIME.....	190
TRACE.....	193
TRANSLATE.....	194
TRUNC.....	196
UPPER.....	197
USERID.....	197
VALUE.....	198
VERIFY.....	200
WORD.....	202
WORDINDEX.....	203
WORDLENGTH.....	204
WORDPOS.....	205
WORDS.....	206
XRANGE.....	206
X2B.....	207
X2C.....	208
X2D.....	209

Chapter 6: uni-REXX Extensions.....211

uni-REXX Specific Functions.....	211
RXFUNCCADD.....	212
RXFUNCDROP.....	212
RXFUNDCQUERY.....	212
RXXCOMMANDSPAWN.....	213
RXXCOMMANDKILL.....	213
RXXCOMMANDWAIT.....	213
RXXOSENDOFLINESTRING.....	214
RXXOSENVIROMENTSEPARATOR.....	214
RXXOSPATHSEPARATOR.....	214
RXXSLEEP.....	214
RXXFUNCTIONPACKAGELOAD.....	215
RXXCOMMANDPACKAGELOAD.....	218
UNIX-Specific Functions.....	222
_ACCEPT.....	225
_BIND.....	227
_CLOSEDIR.....	229
_CLOSESOCKET.....	229
_CONNECT.....	230
_ERRNO.....	232
_EXIT.....	233
_FD_CLR.....	233

_FD_ISSET	234
_FD_SET	234
_FD_ZERO	234
_FORK	234
_GETEUID.....	235
_GETHOSTBYADDR.....	236
_GETHOSTBYNAME	236
_GETHOSTID	238
_GETHOSTNAME	238
_GETPEERNAME.....	239
_GETPID.....	240
_GETPPID	241
_GETSERVBYNAME.....	241
_GETSOCKNAME.....	242
_GETSOCKOPT	243
_GETUID	244
_IOCTL	245
_KILL.....	246
_LISTEN	247
_OPENDIR.....	248
_REaddir	249
_RECV	249
_REGEX.....	251
_SELECT	252
_SEND	253
_SETSID	254
_SETSOCKOPT.....	255
_SLEEP	257
_SOCKET	257
_STAT	258
_SYS_ERRLIST.....	260
_SYSTEMDIR	260
_TRUNCATE.....	261
_UMASK	261
_WAIT	262
_WAITPID	263
Client/Server Sample Application	265
Operating System Facilities	274
DESBUF.....	275
DROPBUF	276
EXECIO	277
GLOBALV.....	284
MAKEBUF	289
RXQUEUE.....	290

SENTRIES	290
Chapter 7: Application Programming Interfaces	291
IRXEXCOM.....	293
IRXEXEC.....	297
IRXEXITS.....	300
IRXJCL	305
IRXSTK	308
IRXSTOP	312
IRXSUBCM	315
Building Embedded Applications	319
External Function Packages.....	323
Control Blocks.....	335
ARGLIST	335
CPCKDIR.....	335
EVALBLOCK.....	336
EXECBLK	336
EXITBLK.....	337
FPCKDIR.....	339
INSTBLK.....	341
IRXSUBCT	343
SHVBLOCK	344
STMT	347
Interprocess Communication with uni-REXX	348
Appendix A: Message Summary	353
Appendix B: Common Pitfalls in uni-REXX Programs	365
Appendix C: Bibliography	371
Appendix D: System Limitations	375
Appendix E: uni-REXX Environment Variables	377

Chapter 1: Introduction

uni-REXX from The Workstation Group is a UNIX implementation of the REXX programming language. It provides an alternative to shell programming and other languages such as awk or perl for command procedures, rapid prototyping, or more complex application development. It may also be embedded as a macro language within applications written in compiled languages such as C. It is the macro language for uni-XEDIT Extended and the language for development of customized dialogs in uni-SPF Extended, both from The Workstation Group.

Unlike shell, awk, or perl programs, applications written in the REXX language are highly portable across a wide variety of platforms from mainframes to UNIX to OS/2. Thus, REXX is the language of choice for applications that must function simultaneously in a network of diverse platforms. Further, uni-REXX facilitates migration of existing applications from the mainframe or OS/2 to UNIX-based workstations.

uni-REXX implements all the features described in the standard reference document for the language, *The REXX Language, A Practical Approach to Programming* by M. F. Cowlshaw (Second Edition, Prentice Hall, 1990). With the exception of minor messages, uni-REXX also implements all the functionality included in ANSI standard X3.274:1996, *Programming Language REXX*.

In addition, uni-REXX includes extensions to the language that are specific to UNIX operating systems and that allow programs to effectively interact with and manipulate their external environment. It also includes a set of application program interfaces that permit development of embedded applications.

Documentation Conventions The following conventions are used throughout this document to facilitate syntax descriptions.

Uppercase

Uppercase letters indicate keywords or function names that must be typed exactly as shown. You may type the keyword or function name in any case, regardless of the documentation convention used.

Lowercase

Lowercase letters indicate variable information that you supply. A single character (usually *n*) represents a number that you specify. Other variable data is represented by a descriptive name such as *string*, *expression*, or *pad*. Variable data is also italicized to facilitate references to it within descriptive text.

Optional Operands

Instructions or functions may have optional keywords or operands. These are shown within brackets in the syntax diagram. The syntax diagram for the LINEIN built-in function illustrates optional operands:

LINEIN([*name*] [, [*lineno*] [, *count*]])

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

TRACE *[option]*
 [[**VALUE**] *expression*]

You actually type only one of the choices – for example, TRACE E.

Required Operands

Required operands are shown without brackets as in the INTERPRET instruction:

INTERPRET *expression*

When a required operand may 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 may have more than one value:

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

Repeating Operands

An ellipsis (...) in a syntax diagram indicates that an operand may be repeated zero or more times. This is illustrated by the MAX built-in function:

MAX(*number* [, *number*] ...)

where you may specify a list of numbers for which the maximum value is to be determined. Do not include the ellipsis when typing your function call.

Delimiters

The following special characters are token delimiters when used outside literal strings:

comma	,
semicolon	;
colon	:
parentheses	()

These characters must be used 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”.

Chapter 2: Language Features

uni-REXX is implemented according to the language definition contained in *The REXX Language, A Practical Approach to Programming*, by M. F. Cowlshaw (Second Edition, Prentice Hall, 1990) and ANSI standard X3.274:1996, *Programming Language REXX*. The elements of the language are described in detail in these documents. This chapter summarizes the language structure for those not already familiar with it.

Clauses

The basic element of the 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 may be any of the following

- literal string
- hexadecimal string
- binary string
- symbol
- operator
- special character

Note that wherever **blanks** are specified in the REXX language, other whitespace characters may appear. These are specified in the ANSI standard, and include the ASCII space character, the carriage return, form feed, new line, horizontal tab, and vertical tab characters.

A **literal string** is a sequence that may include any character and that is enclosed in single or double quotes. A literal string that includes no characters is known as a null string. Examples of literal strings include

```
`Hello world!`  
"What's in a name?"  
` `
```

A **hexadecimal string** is a series of hexadecimal digits grouped in pairs, enclosed in quotes, and followed immediately by the character “x” (upper or lower case). The pairs of hexadecimal digits may be optionally separated by one or more blanks. Examples of hexadecimal strings include

```
`c1c3`x  
"abcdef"X  
`61 62 63`x  
""x
```

A **binary string** is a series of binary digits grouped in fours, enclosed in quotes, and followed immediately by the character “b” (upper or lower case). The groups of binary digits may be optionally separated by one or more blanks. Examples of binary strings include

```
`0001`b  
`10011001`B  
"1111 0000"b  
` `b
```

A **symbol** is any group of alphanumeric characters. Symbols may also include the characters “.”, “!”, “?”, “@”, and “_”. If a symbol begins with a digit, it may also include the letter “e” (upper or lower case) followed optionally by a plus or minus sign (“+” or “-”) and one or more digits, in which case it may be a number in exponential notation. A symbol may 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 “Symbols”. Examples of symbols include

```
abc
```



```
data.1
new_data
17
31416E-4
```

An **operator** is a character used to indicate operations in expressions. The complete list of operators supported in uni-REXX is included in the section entitled “Expressions”. Examples of operator characters include

```
+
-
>
=
```

Special characters include both the operator characters and the characters “.”, “;”, “:”, “(”, and “)”. Special characters function as token delimiters.

A REXX clause may be any of the following types:

- instruction
- label
- null clause

An **instruction** describes an action to be performed by the interpreter. Instructions may be any of the following

- assignment
an instruction of the form *symbol = expression*, which 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

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.

A **null clause** is any clause comprised only of blanks or comments.

A comment is any sequence of characters preceded by “/*” and followed by “*/”. Comments may appear anywhere in the program and may be nested.

A clause in a REXX program may 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
```

Symbols

A symbol in REXX is any group of characters A-Z, a-z, 0-9, “.”, “!”, “?”, “@”, or “_”. The meaning of a symbol is derived from its context. A symbol may be any of the following types:

- constant symbol
 - begins with a digit and may include the letter “e” followed optionally by a plus or minus sign and one or more digits; the value of a constant symbol may not be changed; examples include

```
10  
3.1416  
15e-3  
4x
```

- simple symbol
does not begin with a digit and does not contain any embedded periods; its default value is the symbol translated to uppercase; a simple symbol may be used as a variable and may be assigned a value; examples include

```
list
file1
Date
```

- compound symbol
does not begin with a digit, contains at least one embedded period, and may not end with a period; the name begins with a stem (see description of stem symbols, next in this section) followed by a period followed by a tail; the tail may 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:
 - the value assigned to the stem
 - the symbol name translated to uppercase if no value has been assigned to the stem

Examples include

```
data.1 = 5
say data.1
```

output is "5"; the tail is a constant symbol so this compound symbol does not have a derived name

```
x = 3
data.3 = 7
say data.x
```

output is "7"; the value of the simple symbol "x" is substituted to produce the derived name "data.3", which has been assigned the value "7"

- stem

does not begin with a digit and contains only one period, which must be the last character; may be assigned a value, which effectively assigns that value to all compound symbols which begin with this stem; stems may represent a collection (or array) of variables; examples of stems and their use include

```
list. /* a stem whose value is "LIST." */
```

```
list. = animals  
list.3 = 'cows'  
say list.1 list.new list.3
```

output is “ANIMALS ANIMALS cows” since only the compound symbol list.3 has been assigned a value different from the value assigned to the stem

Expressions

A REXX clause may contain one or more expressions. An expression consists of one or more terms and zero or more operators designating operations to be performed on the terms.

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

- literal string
- symbol
- function call
- sub-expression

Literal strings are treated as constants.

Symbols are translated to uppercase and may be treated as constants or as variables. Symbols that do not begin with a digit may be the name of a variable, in which case the value of that variable is used in the expression.

Function calls are of the form

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

where *function_name* may be a symbol or a string.

A **sub-expression** is any expression enclosed in parentheses.

Operators may be grouped into four categories:

- arithmetic
- comparative
- concatenation
- logical

Arithmetic operators are used to perform operations on numbers. uni-REXX supports the following arithmetic operators:

+	addition
-	subtraction
*	multiplication
/	division
%	integer division (returns integer portion of result)
//	remainder (not modulo - may be negative)
**	exponentiation (raise to a whole number power)

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. Two terms must be absolutely identical to be strictly equal – that is, 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. For strict less than and greater than comparisons, the collating sequence of the internal character representation is used. Thus, these results

may be platform-dependent. Further, for strict comparisons, if *string1* is shorter than *string2* and is also a leading substring of *string2*, *string1* is considered strictly less than *string2*.

uni-REXX supports the following comparative operators:

Normal comparison

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

Strict comparison

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

Concatenation operators combine two strings to form a single string. Concatenation may be indicated in any of the following ways:

	concatenate with no intervening blanks
blank	concatenate with one blank between strings
abuttal	concatenate with no intervening blanks

It is important to remember that concatenation is implied when two adjacent terms are not separated by some other operator.

Logical operators take one or two logical values as their operand(s) and return a logical result – 1 or 0. uni-REXX supports the following logical operators:

&	and; returns 1 if both terms are true
	or; returns 1 if either term is true
&& both)	exclusive or; returns 1 if either (but not term is true
\ or ^	not; 1 becomes 0 or 0 becomes 1

Functions

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

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

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

- built-in
- internal
- external

Built-in functions are part of the language and are always available. These functions are documented in *Chapter 5, Built-in Functions*.

Internal functions are routines 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 are stand-alone routines that may be called by a REXX program. They may be written in REXX or in any language which supports the application program interfaces defined in Chapter 7. External functions are available to any program. In the case of external functions written in REXX, uni-REXX locates the function through use of the environment variable REXXPATH. This is discussed in detail in *Chapter 3, Operation*.

Special Variables

There are three special variables whose values may be set automatically during execution of a REXX program:

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

Condition Traps

While the flow of execution in a program is normally controlled by the instructions in the program, REXX recognizes certain conditions that may alter the flow. Condition traps may 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 conditions that may be trapped are

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 a numeric result has been rounded to fit within the current setting of NUMERIC DIGITS

NOTREADY

indicates an error or end of file detected 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

Input/Output

Input and output operations in uni-REXX are implemented according to the I/O model defined by Cowlshaw in *The REXX Language* (Second Edition) and the ANSI standard X3.274:1996. This includes both character input and output streams and the external data queue. The following instructions and built-in functions for performing I/O are included in uni-REXX:

ADDRESS

keywords of the address instruction support redirection of command input or output

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

Under UNIX, transient I/O streams include STDIN, STDOUT, and pipes, including named pipes. Persistent I/O streams are disk files. The default input stream is STDIN. The default output stream is STDOUT. Using uni-REXX I/O functions with pipes allows you to write filter programs for use with other commands or programs.

uni-REXX is packaged with an EXECIO program that provides an alternative to the REXX I/O model. Use of EXECIO is described in detail in *Chapter 6, uni-REXX Extensions*.

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, REXX includes the PARSE instruction which 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*

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

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

Language Extensions

A number of types of language extensions are implemented in uni-REXX:

- built-in functions that allow a program to manage its environment. These include
 - CHDIR** change the current working directory for this process
 - CUSERID** return the userid of the current process; this is the same as the USERID built-in function but is included for the convenience of those accustomed to writing C programs in UNIX
 - GETCWD** return the full path of the current working directory for this process
 - GETENV** return the current definition of the specified environment variable
 - POPEN** redirect host command output to the external data queue
 - PUTENV** define the specified environment variable for this process; this definition does not persist after the uni-REXX program's process terminates

Detailed descriptions of these functions may be found in *Chapter 5, Built-In Functions*.

- External commands that are part of the operating system in other environments where REXX is implemented. These include:
 - EXECIO** perform file input and output

GLOBALV	define and manage global variables
MAKEBUF	define a new buffer in the program stack
DROPBUF	clear the program stack buffer created most recently or clear a specific program stack buffer and all buffers created subsequently
DESBUF	clear all program stack buffers
RXQUEUE	pipe output to the program stack
SENTRIES	does something about entries in program stack

Detailed descriptions of these commands may be found in *Chapter 6, uni-REXX Extensions*.

- uni-REXX specific, to implement system functions in an operating system independent manner:

RXFUNCADD	register a dynamically loaded function
RXFUNCDROP	deregister a function
RXFUNCQUERY	determine whether a function is registered
RXXCOMMANDSPAWN	start an external command process
RXXCOMMANDKILL	kill a spawned process
RXXCOMMANDWAIT	wait for process completion
RXXCOMMANDPACKAGELOAD	dynamically load commands
RXXFUNCTIONPACKAGELOAD	dynamically load functions
RXXOSENDOFLINESTRING	return end of line characters
RXXOSENVIROMENTSEPARATOR	return environment separator

RXXOSPATHSEPARATOR

return file path separator

RXXSLEEP

sleep the program for nn seconds

Detailed descriptions of these functions may be found in *Chapter 6, uni-REXX Extensions*.

- implementations of UNIX-specific functions normally available only in the C library. These include

_accept	_getsockopt
_bind	_getuid
_closedir	_ioctl
_closesocket	_kill
_connect	_listen
_errno	_opendir
_exit	_readdir
_fd_clr	_recv
_fd_iset	_regex
_fd_set	_select
_fd_zero	_send
_fork	_setgid *
_geteuid	_setsockopt
_gethostbyaddr	_sleep
_gethostbyname	_socket
_gethostid *	_stat
_gethostname	_systemdir
_getpeername	_sys_errlist
_getpid	_truncate
_getppid	_umask
_getservbyname	_wait
_getsockname	_waitpid *

* not available in all UNIX implementations

Detailed descriptions of these functions may be found in *Chapter 6, uni-REXX Extensions*.

Chapter 3: Operation

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

uni-REXX provides both an interpreter (`rx`) and a compiler (`rx`). The interpreter executes uni-REXX programs. The compiler allows you to

- create an intermediate pseudo-code version of a program
- regenerate the interpreter to include user-written functions

The uni-REXX Developer's Kit further includes

- a compiler option to create platform-specific binaries from uni-REXX source programs
- a redistribution license under which you may freely redistribute such binaries as well as any of the UNIX extension commands (`EXECIO`, `GLOBALV`, `MAKEBUF`, `DROPBUF`, `DESBUF`, `RXQUEUE`, `SENTRIES`) required by those programs.

Program Names

A uni-REXX program may have any filename that is supported by your platform's operating system.

Many users may find it helpful to use the extension “.rex” to identify uni-REXX programs. This is not required. It is, however, convenient if you plan to compile your programs, as “.rex” is the default extension used by uni-REXX. You may change the default extension using the environment variable `REXXSUFFIXSOURCE`, described later in this chapter.

Program Execution

Two methods of program execution are available:

- explicit
- implicit

Explicit execution is supported by all UNIX implementations. With this method, you invoke the interpreter as a UNIX command and specify the name of the program to be run. The interpreter executable is found either through the `$PATH` environment variable, or by typing the fully-qualified pathname to the program. uni-REXX is executed under control of a License Manager, which requires that a component file named `twgfiles` be located in the same directory as the interpreter binary.

Typing

```
rxr filename
```

invokes uni-REXX to execute the program *filename*.

uni-REXX locates *filename* by searching first in the current working directory, then in directories specified in the environment variable `REXXPATH`

A uni-REXX program to be explicitly executed must have read permission for all authorized users. Use the UNIX `chmod` command to set read permission for authorized users.

```
chmod +r filename
```

sets read permission on *filename* for owner, group, and other. Other variations of `chmod` allow you to limit this authorization.

Implicit execution (also known as *interpreter files*) allows you to run a uni-REXX program as if it were an operating system command.

There are two requirements for implicit execution of programs under UNIX:

- The first line of the program must be an implicit execution string.
- The program file must have execute permission for all authorized users.

An implicit execution string identifies to the UNIX shell the processor to be used for this program. It must begin with the two characters “#!”. These characters are followed by the full path name of the processor to be used. If you have installed uni-REXX in `/usr/local/rexx`, the implicit execution string for your program would be

```
#!/usr/local/rexx/rxx
```

Some UNIX implementations place one or more of the following limitations on implicit execution strings:

- string may not exceed 32 characters, including the # and ! Characters
- string may not contain embedded blanks

Use the UNIX `chmod` command to set execution permission for authorized users.

```
chmod +x filename
```

sets execution permission on *filename* for owner, group, and other. Other variations of `chmod` allow you to limit this authorization.

The following uni-REXX program, named “sayhi”, is written for implicit execution:

```
#!/usr/local/bin/rxx
/* My greeting program */
say 'Hi there - welcome to uni-REXX'
```

To run this program, you need only type
sayhi

With implicit execution, the UNIX shell uses the normal search methods for locating any command to find the program to be run. Specifically, it searches the directories specified in the environment variable PATH. If the current working directory (specified as “.”) is not included in PATH, the shell will not find your program even if it is in your current directory. In addition, some UNIX versions may require that the uni-REXX executable (rxx) be found in the PATH.

The rxx Command

The rxx command invokes the uni-REXX interpreter. It is normally used for explicit execution of programs but has two additional options.

```
rxx filename[.extension]  
rxx -c 'string'  
rxx -v
```

The syntax shown assumes that rxx is installed in a directory that is included in your PATH environment variable. You may also invoke the interpreter by typing its full path name as in

```
/usr/local/bin/rxx filename[.extension]
```

The first form (**rxx** *filename*[*.extension*]) executes a uni-REXX program. *filename* specifies the name of the program to be run. *extension* is optional. For programs whose filenames have an extension, “rex” is the default.

The environment variable REXXSUFFIXSOURCE controls the default extension for program names. If this environment variable is unset, the default extension is “rex”. To modify this behavior, set REXXSUFFIXSOURCE to the desired filename extension. In this case, uni-REXX runs only a program with

the specified extension, regardless of the presence of programs with other extensions (including “.rex”).

`rx` uses the following search order to locate the program to execute

- current working directory
- directories specified in the environment variable `REXXPATH`

If only *filename* is specified on the `rx` command, the program to be executed is chosen according to the following order of priority.

- *filename* (with no extension)
- *filename*.`rex` (if `REXXSUFFIXSOURCE` is unset)
- *filename.extension*, where *extension* is specified by the current setting of `REXXSUFFIXSOURCE`

Specifying *filename.extension* on the `rx` command always executes the program with the filename and extension specified regardless of the setting of `REXXSUFFIXSOURCE`.

The second form (`rx -c `string``) allows you to specify a string to be interpreted. *string* may be any legal `REXX` clause or a combination of clauses separated by semi-colons. *string* must be enclosed in quotes. This feature allows you to invoke uni-`REXX` for limited purposes without saving the statements in a disk file. It may be used from the system prompt, in another uni-`REXX` program, or in a shell script.

The third form (`rx -v`) displays the version number of your currently installed interpreter.

Examples:

```
rxX hello
```

execute the program named "hello"

```
rxX doit.rex
```

execute the program named "doit.rex"

```
rxX inventory.report
```

execute the program named "inventory.report"

```
rxX -c "say hello world"
```

execute the string enclosed in quotes; output of this command to the terminal screen is

```
HELLO WORLD
```

```
rxX -c "do i=1 to 3; say i; end"
```

execute uni-REXX DO loop enclosed in quotes; output of this command to the terminal screen is

```
1  
2  
3
```

```
rxX -v
```

display the version number of the currently installed interpreter; the output of this command to your terminal screen is

```
uni-REXX (R) The Workstation Group. Version 2.97  
Open-REXX (TM) Copyright (C) iX Corporation 1989-1999.  
All rights reserved.
```

The return code from the rxX command will be the error message number (see Appendix A) if the interpreter encountered a syntax or runtime error in the program. Otherwise, the value of the expression on the EXIT or RETURN instruction supplied by the program is returned. If no value was supplied, zero is returned. To avoid confusion with interpreter detected errors, we suggest that the user program return values between 200-254.

The uni-REXX Compiler

The uni-REXX compiler serves three distinct purposes:

- creation of an intermediate code version of a source program
- regeneration of the interpreter to include user-written functions
- for the uni-REXX Developer's Kit, generation of a platform-specific binary

Intermediate Code Generation. The uni-REXX compiler is not a true compiler in that it does not generate machine language code. It does, however, generate an intermediate pseudo-code that, for some types of programs, provides improved execution speed and memory utilization. Programs that include extensive do loops, subroutines, or function calls will not realize any performance benefits from compilation. However, large programs that are essentially end-to-end code may benefit from compilation. The degree of performance improvement is dependent on the size of the source program.

Intermediate code also provides source code security. The pseudo-code version of the program is not “human readable” and therefore cannot be modified by end users.

The compiler performs a syntax check on the entire program as part of compilation. Thus, users will not encounter syntax errors when executing the compiled program. If any program error is encountered, rxc generate a return code of the error message number (see Appendix A).

uni-REXX marks compiled programs for implicit execution. The compiler automatically places an implicit execution string as the first line in the output file and sets execute permission for user, group, and other. If your operating system does not support implicit execution, you must use the rxx command to run a compiled program.

An intermediate code version of a uni-REXX program requires access to the interpreter at run time. A uni-REXX license is therefore required for execution of a program “compiled” in this manner.

Regeneration of uni-REXX. One of the strengths of the REXX language is that it is extensible through the addition of user-written functions. *Chapter 7, Application Programming Interfaces* provides details on writing such functions and using the compiler to rebuild the interpreter to include them.

Creation of a Platform-Specific Binary. The uni-REXX Developer’s Kit includes the option of the compiler that allows you to create a platform-specific binary from your source program. Such a binary combines the interpreter and the uni-REXX program into a single executable that does not require access to an external interpreter at run time. Such binaries are therefore significantly larger than either a source or intermediate code version of the program. They are, however, freely distributable and may be executed by users who have not licensed uni-REXX. In addition, a license for the Developer’s Kit conveys the right to redistribute the UNIX extension commands (EXECIO, GLOBALV, MAKEBUF, DROPBUF, DESBUF, RXQUEUE, SENTRIES) with any program that uses these commands.

The compiler performs a syntax check on the entire program as part of compilation. Thus, users should not encounter syntax errors when running the binary.

The rxc Command

The rxc command invokes the uni-REXX compiler.

```
rxc filename[.extension]  
rxc -G objfile [-Llibpath] [-llibname] [-o intname]
```

The syntax shown assumes that rxc is installed in a directory that is included in your PATH environment variable. You may also invoke the compiler by typing its full path name as in

/usr/local/bin/rxc filename[.extension]

The first form generates an intermediate code version of a uni-REXX program. *filename* is the name of the source program to be compiled. Source programs **must** have an extension. The output of rxc is a file named *filename* without the extension. If *.extension* is omitted, the default is “.rex”.

The environment variable REXXSUFFIXSOURCE controls the default extension for program names. If this environment variable is unset, the default extension is “rex”. To modify this behavior, set REXXSUFFIXSOURCE to the desired filename extension. In this case, rxc compiles only programs with the specified extension, regardless of the presence of programs with other extensions (including “.rex”).

rxc automatically places an implicit execution string at the beginning of the output file. For this reason, you **must** have access to the interpreter at compile time. Normally, rxx and rxc are installed in the same directory, making this access automatic. If rxc cannot locate the interpreter, the message

```
Error 102 in filename.ext: Cannot find rxx for implicit execution
```

appears.

The compiler automatically sets execute permissions on the output file for owner, group, and others.

The second form of the rxc command rebuilds the interpreter to include user-written functions. ***Chapter 7, Application Programming Interfaces*** gives details of this syntax and its use.

If you have licensed the uni-REXX Developer's Kit, there is a third form of the rxc command available to generate an executable binary.

rxc -m *filename*[.*extension*]

filename is the name of the uni-REXX source program to be compiled. Source programs **must** have an extension. The output of rxc is a file named *filename* without the extension. If *.extension* is omitted, the default is ".rex" or the extension specified by the REXXSUFFIXSOURCE environment variable as described previously. The compiler automatically sets execute permissions on the output file for owner, group, and others.

When generating a binary, the uni-REXX compiler must have access to all of the following resources on your system:

- space to create temporary files
- C compiler and loader
- uni-REXX libraries

Access to these resources may be controlled by the setting of appropriate environment variables. Defaults and options are shown in the table on the following page. *Appendix E, uni-REXX Environment Variables* includes additional details.

Resource	Default	Environment Variable
Temp space	/tmp	REXXTEMP set this environment variable to the directory in which you want temporary files to be placed
C compiler	cc	REXXMODULECC if you do not use the command "cc" to access your C compiler, set this environment variable to the correct command

C compiler flags	platform-specific	REXXMODULECFLAGS set this environment variable to the proper compilation flags for your C compiler
Loader flags	platform-Specific	REXXMODULELDFLAGS set this environment variable to the proper “ld” flags for your loader; at minimum, these must include the math library and the socket library; additional libraries may be required on some platforms
uni-REXX libraries	none – site-specific	REXXLIB set this environment variable to the directory containing librx.a and librx1.a; typically, this is the same directory where uni-REXX is installed
Resulting module	stripped	REXXMODULELDSYMBOLS if this environment is unset, the resulting module is automatically stripped of all symbol tables; set this variable to any value to suppress stripping of the module

Note that no default is provided for the location of the uni-REXX libraries since this may be highly site-specific. Therefore you **must** set REXXLIB before attempting to generate a binary. If rxc cannot locate the archive library, the message

```
Error 105 in filename.ext: Environment variable REXXLIB not set
```

appears.

To set environment variables, select the appropriate command(s) from the following table, which uses REXXLIB as an illustration. The examples assume that librx.a is located in /usr/local/rexx.

Shell	Command(s)
C	setenv REXXLIB /usr/local/rexx
Bourne	REXXLIB=/usr/local/rexx export REXXLIB
Korn	REXXLIB=/usr/local/rexx export REXXLIB

Examples:

`rxcc doit.rex`
create an intermediate code version of the program named “doit.rex”; output is named “doit”

`rxcc doit`
same as previous example; `rxcc` uses the default extension “.rex” if no extension is supplied

`rxcc report1.prog`
create an intermediate code version of the program named “report1.prog”; output is named “report1”

`rxcc -G myfuncs.o`
regenerate the interpreter (`rxcc`) to include the user-written functions in the object file “myfuncs.o”

`rxcc -G myfuncs.o -o fred`
regenerate the interpreter to include the functions in the object file “myfuncs.o”; name the new interpreter “fred”

`rxcc -m doit`
create an executable binary of the program “doit.rex”; output is named “doit”; this option is valid only with a license for the Developer’s Kit

External Functions and Subroutines

uni-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:

- current working directory
- directories specified in the environment variable REXXPATH

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

```
Error 43 on line n in filename: Routine not found appears.
```

External Data Queue

Unlike other environments in which the REXX language is implemented (notably VM/CMS), UNIX does not have the concept of a persistent stack. Thus the external data queue in uni-REXX is implemented as an internal facility and is referred to as the “uni-REXX program stack” or the “program stack” throughout this manual.

Programs executed under UNIX run in separate processes. uni-REXX implements special interprocess communication capabilities to permit sharing of the program stack between uni-REXX programs. This facility cannot be used to communicate with applications, commands, or programs that are not written in uni-REXX.

The environment variable REXXSTACKSHARED is used to control this feature. If this environment variable is set, stack sharing is enabled. The default is to not share the stack. To enable sharing of the program stack, set this variable to any value.

There are three options for capturing the output of UNIX commands or non-uni-REXX programs onto the stack for use by your REXX program:

- the POPEN function, which automatically places STDOUT from the command or program onto the stack
- the RXQUEUE extension command; pipe the output from a command or program to RXQUEUE and that output is placed on the stack; this is especially useful when addressing other than the default host command environment
- the ADDRESS instruction, using keywords to redirect command input, output, or error to the stack

uni-REXX also includes UNIX-specific extensions that implement interprocess communications. This facility provides an alternative method of communication between programs. You may create stand-alone applications that implement interprocess communications with uni-REXX programs. *Chapter 7, Application Programming Interfaces*, provides details on the use of interprocess communications, including examples of programs that use the facility.

Host Command Execution

There are a number of alternatives for executing host commands in uni-REXX. The choice depends on the command to be executed and whether or not you need access to output from the command for further processing. You may execute a host command in one of the following ways:

- directly, by including the command as a clause in the program. The command may or may not be enclosed in quotes. Use quotes to insure that it is treated as a host command if there is any risk that a program variable may have the same name as the host command or any of its operands. Quotes may also be necessary to insure the case-sensitivity of the host command. In this case, command output is directed to STDOUT. You must redirect STDOUT to a file if the output is required for later use.

- by use of the ADDRESS instruction. ADDRESS allows you to specify the name of the host command environment that is to process the command. The default host command environment is UNIX. Commands addressed to UNIX are processed by the Bourne shell. Additional host command environments are supported for the special purposes indicated

cs	for commands that are only available in the C shell or for command syntax specific to the C shell
ks	for commands that are only available in the Korn shell or for command syntax specific to the Korn shell
sh	for commands that are only available in the Bourne shell or for command syntax specific to the Bourne shell; this is identical to the default UNIX command environment
command	for commands that have operands which might normally be expanded by the shell, such as “*”; no shell is used; the command is executed directly; because no shell is invoked, piping (), redirection(>, >>, or <), filename expansion (*, ?, [], etc.), and backgrounding (&) are unavailable (for compatibility with VM/CMS programs, cms is a synonym for command)

Command output is directed to STDOUT. You may redirect STDOUT to a file if the output is required for later use.

Keywords of the ADDRESS instruction also provide for redirection of command input, output, or error streams to the uni-REXX program stack, a stem in the current program, or a file.

- by using the POPEN built-in function. POPEN redirects command output to the uni-REXX program stack where it is available to the program through PULL or PARSE PULL.

Chapter 4: Instructions

A REXX instruction is one or more clauses that may either

- control the program flow
- manipulate data
- 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 “=” (which implies assignment) or “:” (which indicates a label)

Instruction keywords are reserved when used in the context just described. 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-dependent. Further, adjacent blanks have no effect other than to separate the keyword from surrounding tokens.

The following instructions are provided in uni-REXX:

ADDRESS	OPTIONS
ARG	PARSE
CALL	PROCEDURE
DO	PULL
DROP	PUSH
EXIT	QUEUE
IF	RETURN
INTERPRET	SAY
ITERATE	SELECT
LEAVE	SIGNAL
NOP	TRACE
NUMERIC	UPPER

ADDRESS

The ADDRESS instruction specifies the external environment for the execution of host commands. It also supports redirection of input, output, and error streams associated with the command.

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

environment is the name of the host command environment for subsequent host commands.

expr1 is the host command to be executed. This may 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 permanent change to the default host command environment.

When a new host command environment is specified, this becomes the primary host command environment. uni-REXX retains the previous environment name and any associated I/O redirection as the alternate environment. Repeated execution of ADDRESS without operands then has the effect of a toggle between the primary and alternate environments.

ADDRESS [VALUE] *expr2* is 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 (that is, if it starts with a special character), you may omit the sub-keyword VALUE.

redirect represents the keyword syntax that supports I/O redirection. This syntax is as follows:

INPUT		PULL	
		STEM	<i>stem_name</i>
		STREAM	<i>file</i>
		NORMAL	
OUTPUT	[REPLACE] [APPEND]	PUSH	
		QUEUE	
		STEM	<i>stem_name</i>
		STREAM	<i>file</i>
		NORMAL	
ERROR	[REPLACE] [APPEND]	PUSH	
		QUEUE	
		STEM	<i>stem_name</i>
		STREAM	<i>file</i>
		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 uni-REXX program stack. **PUSH** and **QUEUE** redirect

command output or error to the uni-REXX program stack in the same manner as the PUSH and QUEUE instructions. These keywords are uni-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 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 uni-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**.

Normally, the default host command environment is UNIX, though this may not be the case for applications that embed uni-REXX as a macro language. The following additional host command environments are automatically supported:

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).

cs

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 backgrounding (&) are unavailable. For compatibility with VM, **cms** is accepted as a synonym for **command**.

Applications that embed uni-REXX as a macro language may define additional host command environments and/or set a different default. This is accomplished through the IRXSUBCM application programming interface described in *Chapter 7, Application Programming Interfaces*.

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

In the UNIX environment, any host command sent to the default host command environment or to one of the

automatically recognized environments spawns a new process to execute the command. When the command completes, the spawned process terminates. If the command changes an attribute that is unique for each process (such as current working directory), the change is associated only with the spawned process and has no effect on the process in which uni-REXX is running.

Examples:

```
/*
 * 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'
```

```
/*
 * 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
```

```
/*
 * 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
/*
 * 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
```

ARG

The ARG instruction retrieves the argument string(s) of a program or an internal routine and puts them into variables.

ARG [*template*]

The ARG instruction is simply a short form of

PARSE UPPER ARG [*template*]

Thus, characters in the argument string(s) 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 string(s).

template is the parsing template that defines how the argument string(s) 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.

As with the PARSE instruction, ARG may be used repeatedly with different templates to separate the argument string(s) in different ways.

The argument string(s) and information about the argument string(s) are also accessible from the ARG built-in function, described in *Chapter 5, Built-In Functions*.

Examples:

```
/*
 * 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!"
 */
```

```

/*
 * the following program fragment accepts a
 * maximum of 2 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

```

```

/*
 * the following program fragment illustrates
 * repeated use of ARG to separate the argu-
 * ment 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

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

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

name is the subroutine to be invoked. It may refer to any of the following types of routines, using the search order shown below:

Internal routine

any subroutine or function contained within the current program and identified by a label

Built-in function

one of the uni-REXX built-in functions described in *Chapter 5, Built-In Functions* or one of the UNIX-specific functions described in *Chapter 6, uni-REXX Extensions*

External routine

an external program written in REXX or a function written in a language other than REXX that has been added to the uni-REXX interpreter or that is part of an application that embeds uni-REXX as a macro language

name must be either a symbol or a literal string. If it is a literal string, it may 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.

expr is any valid REXX expression. The expression(s) are evaluated from left to right with the results passed to *name* as the calling argument(s).

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

If name is an external routine, you **must** use PROCEDURE EXPOSE to make variables from the caller available to the subroutine.

The following state information is saved across calls to internal subroutines and restored when control is returned to the caller:

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

Elapsed time clocks

the subroutine may 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

CALL	ON	<i>condition</i> [NAME <i>trapname</i>]
	OFF	<i>condition</i>

The ON and OFF sub-keywords of CALL 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 is the name of the condition to be detected.
If a condition trap is enabled, when that condition oc-
curs, control is passed to one of the following:

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

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

The following conditions may be controlled using the CALL instruction:

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 or end of file detected during an I/O operation

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

- CALL cannot be used with the LOSTDIGITS, NOVALUE and SYNTAX conditions
- state information is preserved across the CALL so the trap routine may return to the caller, which may resume execution; with SIGNAL, program execution terminates when the trap routine completes

Examples:

```
/*
 * 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
```

```
/*
 * 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
```

```

/*
 * the following program fragment uses CALL to
 * control condition traps
 */
call on error
call on halt name interrupt
address csh 'holycow'
:
i = 1
do 100000
    i = i + 5
    say i
end
exit
error:
say 'Error condition detected at line' sigl
return
interrupt:
say 'Ctl-C detected; exiting at your request'
exit
/*
 * because the C shell 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
 */

/*
 * this fragment reads a file, displays
 * A message when end of file detected.
 */
call on notready
n = 0

do forever
    line = linein()
    n = n + 1
end

notready:
    say 'End of file after record:' n
    exit

```

```

/*
 * 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

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

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

A DO instruction group consists of the DO instruction followed by one or more instruction clauses followed by the keyword END. The END keyword **must** begin a new clause. *instr_list* represents the instruction clause(s) included in the group. Any uni-REXX instruction may appear in the group, including the DO instruction.

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

repetitor may be any of the following:

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

exprn is any expression that evaluates to a number. It is rounded before use according to the current setting of NUMERIC DIGITS. When used alone or with the FOR keyword, *exprn* must evaluate to a non-negative whole number.

name is a control variable. It may be any valid symbol. *name* is assigned an initial value at the beginning of the loop and is stepped BY a specified increment TO a maximum value or FOR a designated number of iterations. The value of the control variable may 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 may have an unexpected and undesirable effect on the result. Again, this is not normally considered to be good programming practice.

TO, BY, and FOR may be used in any combination and in any order. They are evaluated in the order in which they appear in the DO instruction clause. The default value for “BY *exprn*” is 1. The expressions associated with TO, BY, and FOR are evaluated only once – when the DO instruction is first executed. The TO condition and the FOR count are checked at the beginning of each iteration of the loop. If the TO condition is already satisfied at the start of the first iteration, the instruction group is never executed.

The TO, BY, and FOR keywords are reserved within the context of a DO instruction. This means that they cannot be used in any of the expressions that appear in conjunction with the specification of a control variable.

The FOREVER keyword indicates that the instruction group should be repeated until some instruction is executed that deactivates the loop.

conditional may be any of the following:

WHILE *exprl*
UNTIL *exprl*

exprl is any expression that evaluates to 0 or 1. *exprl* is evaluated for each pass through the loop using the current values for all variables. The instruction group is repeated WHILE *exprl* evaluates to 1 or UNTIL *exprl* evaluates to 1. 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. 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 may also be modified by the execution of a LEAVE or ITERATE instruction.

Examples:

```
/*
 * the following program fragment illustrates
 * the simplest form of 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
```

```
/*
 * 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
```

```

/*
 * the following program fragment illustrates
 * a simple controlled repetitive DO loop; the
 * instruction group is repeated until the con-
 * trol variable "i" is equal to the value of
 * "march.0", which was automatically set to
 * the number of lines read by execio; since
 * the BY keyword is not present, "i" is incre-
 * mented by 1 each time through the loop
 */
msd = '/home/sales/march_sales_data'
total = 0
address command
'execio * diskr' msd '(stem march.'
do i = 1 to march.0
    total = total + word(march.i, 7)
end
say 'Total March sales:  '$' total

```

```

/*
 * the following program fragment illustrates
 * a controlled repetitive do loop in which the
 * number of iterations is determined by either
 * the value of the control variable or the
 * number of iterations performed; this example
 * estimates sales projections for the current
 * month by totalling all sales records or the
 * first 10 records, whichever is fewest, since
 * sales typically average 10 per week
 */
msd = '/home/sales/april_sales_data'
total = 0
address command
'execio * diskr' msd '(stem april.'
do i = 1 to april.0 for 10
    total = total + word(april.i, 7)
end
guess = total * 4
say 'Sales projection for this month:  '$' guess
/*
 * if there are fewer than 10 records in the
 * file, the value of "i" becomes equal to the
 * value of "april.0" before 10 iterations and
 * this condition terminates the loop; if there
 * are more than 10 records in the file, the
 * loop terminates after the 10th iteration,
 * regardless of the value of "i"
 */

```

```

/*
 * the following program fragment is a varia-
 * tion of the previous example; it makes an
 * annual sales forecast based on data for all
 * months or for the most recent 6 months,
 * whichever is fewer; note the use of a
 * negative increment with BY to work backward
 * through the data
 */
msd = '/home/sales/annual_sales_data'
total = 0
address command
'execio * diskr' msd '(stem annual.'
do i = annual.0 to 1 by -1 for 6
    total = total + word(annual.i, 7)
end
if annual.0 > 6 then guess = total * 2
else guess = total * (12/annual.0)
say 'Annual sales projection: $' guess

```

```

/*
 * 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 illus-
 * trates 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 selecton:' thislang
        say 'Must be one of the following:' list
        thislang = ''
        say ''
    end
end
end

```

```

/*
 * 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

```

```

/*
 * 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

The DROP instruction restores one or more variables to their uninitialized state.

DROP *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 may not include a variable reference – that is, it must be a list of symbols, representing valid variables, separated by blanks. *varlist* may include the same variable more than once. It may also contain variables that have never been assigned a value.

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.

Examples:

```
x = 10
drop x
say x
/* the output is "X" */
```

```
x.a = 'cow'
x.b = 'pig'
drop x.
say x.b
/* the output is "X.B" */
```

```
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" */
```

```

/*
 * 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
 */

```

```

/*
 * 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

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

EXIT [*expression*]

expression is any valid uni-REXX expression. Its value is returned to the caller as a character string. If the program is exiting to the UNIX shell, *expression* must have a numeric value in the range of 0-255.

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 routines, 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 routine.

Examples:

```
say 'Hello world'
exit
/*
 * this is identical to the one-line program
 *   say 'Hello world'
 */

/*
 * 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
```

```

/*
 * the following program fragment illustrates
 * the use of exit to terminate a program when
 * an unexpected condition occurs; it generates
 * a report which 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:
/*
 * function to calculate number of days in
 * February
 */
:
:
return howmany

```


IF

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

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

expression must evaluate to 0 or 1.

instruction may be an assignment, a command, or an instruction, including IF and SELECT constructs and DO groups.

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

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, then *instruction* is bypassed. It is not necessary for the keyword THEN to begin a new clause.

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.

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 REXX, so putting an extra semicolon after the THEN or ELSE results in Error 14, Incomplete DO/SELECT/IF or Error 8, Unexpected THEN or ELSE.

Examples:

```
rc = linein('data.file')
if rc \= 0 then say 'Error reading data.file'
/*
 * the simplest form of IF
 */

/*
 * the following program fragment still uses
 * the simplest form of IF but uses a function
 * which evaluates to 0 or 1 as the conditional
 * expression
 */
val = 'abc'
if datatype(val, 'l') then
    upper_val = translate(val)

/*
 * 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

/*
 * the following program fragment extends the
 * previous example to illustrate 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
```

```

/*
 * 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 selecton:' thislang
    say 'Must be one of the following:' list
end

/*
 * the following program fragment illustrates
 * the use of nested IF instructions; this
 * program would be run under uni-SPF
 */
stab = 'SDATA.tbl'
spanel = 'sales.data.pnl'
say 'Enter directory name for your sales data'
parse pull dir
if chdir(dir) = 0 then
    if stream(stab,'c','query exists')\='' then do
        address ispeexec
            `tbdispl' stab `panel('spanel')' ,
                `cursor(zcmd)'
                :
                :
        end
        else say 'Table not found'
        else say 'Change directory failed'

/*
 * the following program fragment illustrates
 * the use of NOP in nested IF instructions;
 * file exists & is writable?; if not, is di-
 * rectory writable?; runs under uni-SPF
 */
parse arg directory file
if cf(directory"/"file)
    then if \cf(directory"/"file, "fw")
        then do
            `setmsg msg(rxxb02)'; return 4
        end
        else nop
    else if \cf(directory, "dw")
        then do
            `setmsg msg(rxkb03)'; return 4
        end
end

```

INTERPRET

The INTERPRET instruction executes dynamically created instructions.

INTERPRET *expression*

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

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.

Examples:

```
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
 */

/*
 * 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

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

ITERATE [*name*]

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.

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.

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.

Examples:

```
/*
 * 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

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

LEAVE [*name*]

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.

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.

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.

Examples:

```
/*
 * 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

The NOP instruction is a dummy instruction.

NOP

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

Examples:

```
/*
 * 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

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

NUMERIC	DIGITS	[<i>expr1</i>]
	FORM	[SCIENTIFIC] [ENGINEERING] [[VALUE] <i>expr2</i>]
	FUZZ	[<i>expr3</i>]

DIGITS controls the precision for arithmetic operations and for the evaluation of arithmetic functions.

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 upper limit is 1000.

It should be noted that small values of NUMERIC DIGITS may produce unexpected or undesirable results in some cases since the setting affects all computations. For example, the execution of a DO loop may 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 5, Built-In Functions*.

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

The NUMERIC FORM setting may 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 may 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 5, Built-In Functions*.

FUZZ controls the number of digits, at full precision, that are ignored for numeric comparisons.

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.

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 5, Built-In Functions*.

Examples:

```
/*
 * 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
 */

/*
 * 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
 */

/*
 * 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
 */
```

OPTIONS

The **OPTIONS** instruction passes special requests to the language processor. At the present time, this instruction is ignored in uni-REXX.

PARSE

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

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

template is a list of symbols separated by blanks and/or patterns. The symbols are the names of variables to which data is assigned. If *template* is omitted, variables are not set but data is prepared for parsing in one of the following ways:

- for LINEIN or PULL
a line is removed from a character stream or the uni-REXX program stack
- for VALUE
expr is evaluated
- for VAR
if the variable does not have a value, the NOVALUE condition is raised

A detailed discussion of parsing templates is included in this section.

ARG indicates that the data to be parsed is the argument string(s) 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

```
PARSE VALUE LINEIN() WITH [template]
```

In UNIX, the default character input stream is STDIN, which may be the terminal or a pipe. If no data is available on the default character input stream, the program pauses for input.

PULL indicates that the data to be parsed is one of the following:

- if data is available on the uni-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

SOURCE indicates that the data to be parsed is a special string that defines the source of the program being executed. The SOURCE string is fixed and contains the following tokens

- system where the program is running
for uni-REXX, this is UNIX
- how the program was invoked
this is either COMMAND, FUNCTION, or SUBROUTINE
- the full pathname of the program
- the name of the program without the path
- the default host command environment
normally this is UNIX but it may be different in applications that embed uni-REXX as a macro language

VALUE 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 may not be included in *expr*.

VAR 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.

VERSION indicates that the data to be parsed is a special string describing this version of uni-REXX. The **VERSION** string is fixed and contains the following tokens:

- language name
the first four characters are “REXX” with the remainder of the token being implementation-dependent; for uni-REXX, this token is “REXX:uni-REXX:2.00”
- language level
this indicates the degree of compliance with the language level definitions in *The REXX Language* by Cowlshaw; 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)

Parsing Templates

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

Parsing by words

The simplest form of parsing template 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 may 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.

Examples:

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

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

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

Note that 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.

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.

Examples:

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

```
string = 'time and time again'
parse upper var string a 'and' b
/*
 * a == 'TIME'
 * b == ' TIME AGAIN'
 */
```

```
parse arg x ',' y
/*
 * if the argument string passed to this
 * program is "4,3", then
 * x == '4'
 * y == '3'
 */
```



```

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

/*
 * 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
 */

```

Note that 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.

Parsing by position

When parsing by position, the template includes column numbers where the next token begins. These may 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.

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 equal 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.

Examples:

```
x = 1234567890
parse var x 5 y
/*
 * move to the 5th column and assign the rest
 * of the string to variable "y"
 * y = '567890'
 */
```

```
x = 1234567890
parse var x y 3 +4 z
/*
 * assign to "y" data up to column 3, then move
 * forward 4 columns and assign the rest of the
 * string to "z"
 * y = '12'
 * z = '7890'
 */
```

```

x = 'abcdefghijklmnop'
parse var x a 4 b +5 +1 c
/*
 * assign to "a" data up to column 4; assign to
 * "b" data in the next 5 columns; move forward
 * one column; assign the rest of the string to
 * "c"
 * a == 'abc'
 * b == 'defgh'
 * c == 'jklmnop'
 */

```

```

x = abcdefgh
parse var x a 4 -2 b 1 c +4
/*
 * assign to "a" data up to column 4; move back
 * 2 columns; assign the rest of the string to
 * "b"; move to column 1; assign the next four
 * columns to "c"
 * a == 'ABC'
 * b == 'BCDEFGH'
 * c == 'ABCD'
 */

```

```

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
 */

```

```

x = 1234567890
parse var x 3 a +0 b +3 +1 c
/*
 * move to column 3; assign the rest of the
 * string to "a" but don't move the parsing
 * position; assign the next three characters
 * to "b"; move forward 1 column; assign the
 * rest of the string to "c"
 * a == '34567890'
 * b == '345'
 * c == '7890'
 */

```

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

Parsing with placeholders

Parsing templates may 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.

Examples:

```
x = 'How are you'
parse var x a . b
say a b 'be?'
/*
 * the output is "How you be?"
 */
```

```
x = 'one potato two potato three potato four'
parse var x a . b . c . rest
say a b c rest
/*
 * the output is "one two three four"
 */
```

```
file = '/home/salesmgr/ytd_commissions'
do while lines(file) \= 0
  parse value linein(file) with month . amount
  end
/*
 * the second word on each line of the file is
 * discarded - presumably this is data that is
 * irrelevant to the current processing
 */
```

Putting it all together

Parsing templates may include any combination of the elements discussed above. This makes PARSE an extremely powerful and flexible tool for manipulating data. The following example illustrates several uses of PARSE.

Examples:

```
/*
 * the following program generates the name
 * of a directory where a TWG product is
 * stored; normally, input is provided as a
 * calling argument; if the calling argument is
 * omitted, the program looks first at the
 * environment variable VER; if this is not
 * set, it prompts the user; the user is re-
 * prompted until valid input is received; for
 * user convenience, any abbreviation or case
 * is accepted; the product can be called as a
 * subroutine because it exits with the name
 * that it constructed; PARSE is used exten-
 * sively throughout
 */
parse arg prod .
prodlist = ` XEDIT REXX SPF'
thisprod = ``
fromver = 0
firstpass = 1

do while thisprod = `` /* for re-prompting */
  if prod = `` then do /* if no calling arg */
/*      don't check VER if re-prompting */
    if firstpass then ver = getenv('VER')
      else ver = ``
    if ver \= `` then do /* if VER set */
/*      value of VER is a directory path */
      parse var ver '/' first '/' second '/' .
      if first = 'prod' then do /*if rite dir*/
        prod = substr(second, 4)
        fromver = 1 /* got something valid */
      end
    end
    if \fromver then do /*didn't get from VER*/
      say ``
      say 'Product?' /* so prompt user */
      parse pull prod
    end
  end
end
```

```

/*
 * uppercase what we have and drop a blank in
 * front of it so it can be used in a pattern
 * matching PARSE; thus any valid abbrevia-
 * tion preceded by a blank will match with
 * prodlist; parse template uses positional
 * parameter to remove the blank if a match is
 * found; if no match, thisprod will be null,
 * triggering the re-prompt
 */
blprod = ` `upper(prod)
parse var prodlist (blprod) +1 thisprod .
if thisprod = `` then do
  say ``
  say `Invalid product selection:` prod
  say `Product must be one of the following:`
  say ` ` lower(prodlist)
  say ``
  prod = ``          /* reset for next pass */
  firstpass = 0 /*no longer first time thru*/
end
end
product_dir = `uni`thisprod
exit product_dir

```

PROCEDURE

The **PROCEDURE** instruction is used in an internal or external routine to protect the caller's variables from modification during execution of the routine. It also has the effect of insuring that the subroutine's variables are in their uninitialized state each time the routine is called. **PROCEDURE** may not be used in the main program. **PROCEDURE EXPOSE** in external routines is a uni-REXX extension to the ANSI standard and should not be used if portability to other platforms is a consideration.

PROCEDURE [EXPOSE *varlist*]

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 may be modified by the subroutine. Using **PROCEDURE** protects the caller's variables from modification by the subroutine.

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.

The **EXPOSE** sub-keyword allows you to selectively expose variables from the caller's environment for manipulation by the subroutine. In an external routine, you **must** use **PROCEDURE EXPOSE** to make any of the caller's variables available to the subroutine.

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 may not include a variable reference – that is, it must be a list of symbols, representing valid variables, separated by blanks. *varlist* may include the same vari-

able more than once. It may also contain variables that have never been assigned a value.

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.

Examples:

```
/* 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
 */
```

```
/* 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
 */
```



```
/*
 * 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
 */
```

```
x = 10; y = 20; z = 30
call blotz
say y
exit
blotz:
procedure expose y
say y
return
/*
 * the output is
 *      20
 *      20
 */
```

```
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
 */
```

```
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")
 */
```

```
/*
 * 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
say p.b
b = 4
return
/*
 * the output is
 *      200
 *      unknown value
 */
```

PULL

The PULL instruction reads a line from the uni-REXX program stack. If the program stack is empty, PULL reads from the default character input stream (STDIN).

PULL [*template*]

The PULL instruction is simply a short form of

PARSE UPPER PULL [*template*]

The data read is 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.

template is the parsing template that defines how the data is assigned to variables. For details on parsing templates, refer to the PARSE instruction in this chapter. If *template* is omitted, the data read by PULL is simply discarded. This is functionally equivalent to using “PULL .”, where the template is comprised solely of the placeholder symbol.

The number of lines currently available in the program stack is accessible with the QUEUED built-in function described in *Chapter 5, Built-In Functions*.

Examples:

```
/*
 * the following program fragment processes
 * all data currently available on the
 * uni-REXX program stack
 */
do j = 1 while queued() > 0
  pull order.j . amount.j .
end
```

```
/*
 * the following program fragment assumes that
 * no data is 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

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

PUSH [*expression*]

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

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 5, Built-In Functions*.

Examples:

```
and = 'not'
shove = 'nice'
push and shove
/*
 * places "not nice" at the top of the program
 * stack
 */

/*
 * 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

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

QUEUE [*expression*]

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

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 5, Built-In Functions*.

Examples:

```
for = 'how much'
entry = 'longer?'
queue for entry
/* places "how much longer?" at the bottom of
 * the program stack */

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

RETURN

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

RETURN [*expression*]

expression is the value to be returned to the caller. *expression* may evaluate to any character string, including the null string.

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.

Examples:

```
/*
 * 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:
:
:
```

```

/*
 * 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

```

```

/*
 * 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:
/*
 * function to calculate number of days in
 * February
 */
    :
    :
return howmany

```


SAY

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

SAY [*expression*]

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

In UNIX, the default character output stream is STDOUT and is normally the terminal unless STDOUT 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.

Examples:

```
say 'Hello world'
/*
 * writes the string 'Hello world' to STDOUT,
 * normally the terminal
 */

say 'Enter amount of sale'
pull amount
say 'Commission is:' amount * .06
/*
 * the output is 6% of the sale amount entered
 */

retcode = linein('data.file')
if retcode \= 0 then
  say 'Error reading "data.file"'
/*
 * if the read operation fails, the message
 * is displayed
 */
```

SELECT

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

SELECT

whenlist

[OTHERWISE [;] [*instr_list*]]

END

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.

whenlist defines the conditions under which each alternative is selected. *whenlist* is made up of one or more constructs of the form

WHEN *expression* [;] **THEN** [;] *instruction*

expression must evaluate to 0 or 1.

instruction may be an assignment, a command, or an instruction, including the DO, IF, or SELECT instruction.

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

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.

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.

If you are certain that one of the **WHEN** alternatives will be executed, the **OTHERWISE** clause may 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 **REXX**, so putting an extra semicolon after the **THEN** results in an error.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of SELECT to choose among alterna-
 * tive 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
```

```

/*
 * the following program fragment uses SELECT
 * to set processing parameters; this program
 * would be run under uni-SPF
 */
arg region
select
  when region = 'EAST' then do
    rtable = 'EREG.tbl'
    ctable = '/home/smith/COMM.tbl'
  end
  when region = 'WEST' then do
    rtable = 'WREG.tbl'
    ctable = '/home/jones/COMM.tbl'
  end
  otherwise do
    rtable = 'CORPSALES.tbl'
    ctable = '/home/vpsales/CORPCOMM.tbl'
  end
end
address ispxec 'vput (rtable ctable) shared'
address ispxec 'select cmd(do_commissions)'

```

```

/*
 * 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.file') \= 0
  dowhat = word(linein('parms.file'), 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

```

SIGNAL

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

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

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.

As an alternative, the label name may be derived from the expression following the keyword VALUE. *expression* must evaluate to a valid label name in the current program. The keyword VALUE may 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.

```
SIGNAL  ON    condition [NAME trapname]
          OFF   condition
```

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:

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

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

The following conditions may be controlled using the SIGNAL instruction:

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 a numeric result has been rounded to fit within the current setting of NUMERIC DIGITS

NOTREADY

indicates an error or end of file detected 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 LOSTDIGITS, 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 may return to the caller, which may resume execution.

Examples:

```
/*
 * this fragment reads a file, displays
 * A message when end of file detected.
 */
signal on notready
n = 0

do forever
  line = linein()
  n = n + 1
end

notready:
  say 'End of file after record:' n
  exit
```

```

/*
 * 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 en-
 * vironment, 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 doesn't
 * 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

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

TRACE [*option*]
[**VALUE**] *expression*

option specifies the level of tracing to occur. Alternatively, the level may be taken from the value of *expression*. The keyword **VALUE** may 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”.

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

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 as well; 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

Any trace level may be prefixed by a question mark (?) to enable interactive tracing. When interactive tracing is enabled, uni-REXX pauses for input after each trace output. During interactive tracing, TRACE instructions in the program are ignored. Interactive tracing may be disabled by use of the “?” prefix in a TRACE command pause. Typing TRACE O when tracing has paused also turns off interactive tracing.

During interactive tracing, you may type any valid REXX instruction. Pressing Enter causes the execution of the next instruction in the program.

The trace level may also be specified as a whole number. If it is positive, then that number of interactive traces are skipped before the next pause. If it is negative, then that number of all traces (including interactive traces) are skipped before the next pause.

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 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:

_	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:

>V>	contents of a variable
>L>	literal (constant symbol, uninitialized 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

Examples:

```
/*
 * 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"
 */
#!/usr/local/bin/rxx
trace value arg(1)
file = '/home/'userid()'/infile'
line = linein(file)
x = word(line, 1)
if datatype(x) = 'NUM' then do
    y = x + 456 / 100
    say y
end
call subr
if result >= 4 then address UNIX 'pwd'
exit
subr:
say now in subroutine
return 4
```

OUTPUT FROM: traceit a

```
3 ** file = '/home/'userid()'/infile'
4 ** line = linein(file)
5 ** x = word(line, 1)
6 ** if datatype(x) = 'NUM'
6 **                                     then
6 **                                     do
7 **     y = x + 456 / 100
8 **     say y
9 **     end
11238.56
10 ** call subr
13 ** subr:
14 ** say now in subroutine
NOW IN SUBROUTINE
15 ** return 4
11 ** if result >= 4
11 **                                     then
11 **                                     address UNIX 'pwd'
>>> "pwd"
/home/user1
12 ** exit
```

OUTPUT FROM: traceit c

```
111238.56
NOW IN SUBROUTINE
  >>> "pwd"
/home/user1
```

OUTPUT FROM: traceit e

(No errors occurred)

```
111238.56
NOW IN SUBROUTINE
/home/user1
```

OUTPUT FROM: traceit f

(No failure occurred)

```
111238.56
NOW IN SUBROUTINE
/home/user1
```

OUTPUT FROM: traceit l

```
111238.56
13 *-* subr:
NOW IN SUBROUTINE
/home/user1
```

OUTPUT FROM: traceit n

(No failure occurred)

```
111238.56
NOW IN SUBROUTINE
/home/user1
```

OUTPUT FROM: traceit o

```
111238.56
NOW IN SUBROUTINE
/home/user1
```

OUTPUT FROM: traceit i

```
3 *- * file = '/home/'userid()'/infile'
  >L>      "/home/"
  >F>      "ul"
  >O>      "/home/user1"
  >L>      "/infile"
  >O>      "/home/ul/infile"
  >>>      "/home/ul/infile"
4 *- * line = linein(file)
  >V>      "/home/ul/infile"
  >F>      "111234 John Doe"
  >>>      "111234 John Doe"
5 *- * x = word(line, 1)
  >V>      "111234 John Doe"
  >L>      "1"
  >F>      "111234"
  >>>      "111234"
6 *- * if datatype(x) = 'NUM'
  >V>      "111234"
  >F>      "NUM"
  >L>      "NUM"
  >O>      "1"
  >>>      "1"
6 *- *                                     then
6 *- *                                     do
7 *- *      y = x + 456 / 100
  >V>      "111234"
  >L>      "456"
  >L>      "100"
  >O>      "4.56"
  >O>      "111238.56"
  >>>      "111238.56"
8 *- *      say y
  >V>      "111238.56"
  >>>      "111238.56"
111238.56
9 *- *      end
10 *- * call subr
13 *- * subr:
14 *- * say now in subroutine
  >L>      "NOW"
  >L>      "IN"
  >O>      "NOW IN"
  >L>      "SUBROUTINE"
  >O>      "NOW IN SUBROUTINE"
  >>>      "NOW IN SUBROUTINE"
NOW IN SUBROUTINE
15 *- * return 4
  >L>      "4"
  >>>      "4"
```

```

11 *-* if result >= 4
    >V>     "4"
    >L>
    >O>     "1"
    >>>     "1"
11 *-*                                     then
11 *-*                                     address UNIX 'pwd'
    >L>                                     "UNIX"
    >L>                                     "pwd"
    >>>                                     "pwd"
    >>> "pwd"
/home/ul
12 *-* exit

```

OUTPUT FROM: traceit r

```

3 *-* file = '/home/'userid()'/infile'
    >>>                                     "/home/ul/infile"
4 *-* line = linein(file)
    >>>                                     "111234 John Doe"
5 *-* x = word(line, 1)
    >>>                                     "111234"
6 *-* if datatype(x) = 'NUM'
    >>>                                     "1"
6 *-*                                     then
6 *-*                                     do
7 *-*     y = x + 456 / 100
    >>>                                     "111238.56"
8 *-*     say y
    >>>                                     "111238.56"
111238.56
9 *-*     end
10 *-* call subr
13 *-* subr:
14 *-* say now in subroutine
    >>>                                     "NOW IN SUBROUTINE"
NOW IN SUBROUTINE
14 *-* return 4
    >>>                                     "4"
11 *-* if result >= 4
    >>>                                     "1"
11 *-*                                     then
11 *-*                                     address UNIX 'pwd'
    >>> "pwd"
/home/ul
12 *-* exit

```

UPPER

The UPPER instruction converts one or more variables to uppercase.

UPPER *var_list*

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.

UPPER converts lower-case characters only. Uppercase characters or numbers in a string are unchanged.

Examples:

```
a = 'Hello world'
upper a
say a
/* the output is "HELLO WORLD" */
```

```
a = 'c3po'
b = 'r2d2'
upper a b
say a 'and' b
/* the output is "C3PO and R2D2" */
```


Chapter 5: Built-In Functions

uni-REXX includes a powerful set of built-in functions that may be called by any program. 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 may be used in any expression wherever any other term would be valid. The argument expressions may also be function calls. There may not be intervening blanks between the *function_name* and the opening parenthesis. The presence of such blanks would cause the expression to be interpreted as two unrelated symbols or expressions.

You may 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.

In addition to the built-in functions defined in *The REXX Language* and ANSI X3.274:1996, uni-REXX includes functions that provide compatibility with various

IBM implementations and functions to interface with the UNIX environment.

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

The following built-in functions are available in uni-REXX:

* functions to interface with the UNIX environment

** functions provided for compatibility with IBM implementations

The following general rules should be observed 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 may 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 may be omitted from the right with or without providing the preceding comma.
- Any function name or function argument may be specified in upper-, lower-, or mixed case.
- For functions with arguments that must be one of a specified set of characters, those arguments should be enclosed in quotes. Without the quotes, the argument is an uninitialized symbol. So long as the symbol remains uninitialized, the function behaves as expected since the value of the uninitialized symbol is the symbol in uppercase. If, however, an assignment statement sets the value of that symbol to something else, the function results in Error 40, Incorrect call to routine.

ABBREV

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.

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

information is the unabbreviated string.

info is the abbreviated string. When *info* is the null string, it matches any value of *information* so long as *length* is omitted or specified as 0.

length specifies the minimum length of *info*. If *length* is omitted, the default is the length of *info*.

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 the abbreviation is valid and the function returns 1. If either of these conditions is not met, the abbreviation is invalid and the function returns 0.

Examples:

```
valid = abbrev('month', 'mo')      /* valid = 1 */
valid = abbrev('month', 'mo', 2)   /* valid = 1 */
valid = abbrev('month', 'mo', 3)   /* valid = 0 */

valid = abbrev('month', m)
/* valid = 0; the value of the symbol "m", when
 * not specifically assigned a value, is "M" */

valid = abbrev('month', '')
/* valid = 1; the null string matches any value
 * of information */

month = 'January'
mo = 'Jan'
if abbrev(month, mo) then say 'valid'
  else say 'invalid'
/* output of this program fragment is 'valid' */
```

ABS

The ABS function returns the absolute value of a number.

ABS(*number*)

number is any valid number. The result is formatted according to the current NUMERIC settings.

Examples:

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

numeric digits 4
number = abs(-123456.7890)
say number
/*
 * the output of this program fragment is
 *      1.235E+5
 */
```

ADDRESS

The ADDRESS function returns the name of the current host command environment or the current settings for command input/output redirection.

ADDRESS([*option*])

option controls the information returned by the function. If *option* is omitted, ADDRESS returns the current host command environment.

option may be any of the following:

N (Normal)

the current host command environment; this is the same as the default when *option* is omitted

I (Input)

the current settings for command input redirection

O (Output)

the current settings for command output redirection

E (Error)

the current settings for command error redirection

For option I, O, or E, the function returns a string of two or three blank delimited words. Word 1 identifies the type of I/O redirection. Word 2 indicates the target of the redirection. Word 3 is null unless the redirection target is “STEM” or “STREAM”, in which case it is the name of the stem or stream.

Option	Values
---------------	---------------

I	Word 1	INPUT
	Word 2	default: NORMAL may also be PULL, STEM, or STREAM
	Word 3	default: null may be name of STEM or STREAM
O, E	Word 1	default: REPLACE output replaces previous output may also be APPEND output appended to previous output
	Word 2	default: NORMAL may also be PUSH, QUEUE, STEM, or STREAM
	Word 3	default: null may be name of STEM or STREAM

The default host command environment is “UNIX”. Other host command environments may be specified using the ADDRESS instruction.

The default value for I/O redirection is “NORMAL”. Other values correspond to the redirection keywords of the ADDRESS instruction, which is used to control command I/O redirection.

Examples:

```
env = address()                               /* env = UNIX */

/*
 * the following program fragment sets the
 * default host command environment to "csh"
 * before executing a C shell command
 */
address csh
`history > cmd_list'
say address()
/* the output is "CSH" */

/*
 * the following program fragment redirects
 * command output and later tests the status
 * of output redirection
 */
address csh with output stem cmdout.
:
:
if word(address('o'), 2) \= 'NORMAL' then
    say address('o')
/*
 * the output is "REPLACE STEM CMDOUT."
 */
```

ARG

The ARG function returns the argument string or information about the argument string.

ARG([*n* [, *option*]])

With no arguments, ARG returns the number of arguments passed to the program or internal subroutine.

n indicates the argument number to be returned and must be a positive integer. When only *n* is specified, ARG returns the *n*th argument string.

option is used only in conjunction with *n*. When both arguments are specified, ARG tests for the existence of the *n*th argument string. *option* may be either of the following:

- E** exists; if the *n*th argument exists, ARG returns 1; otherwise, it returns 0
- O** omitted; if the *n*th argument is omitted, ARG returns 1; otherwise, it returns 0

Examples:

```

call subr                /* no arguments specified */
:
subr:
arglist = arg()          /* arglist = 0 */
arg1 = arg(1)            /* arg1 = '' */
arg1_exist = arg(1,'e')
/*
 * arg1_exist = 0; first argument does not exist
 */

call subr a,,b
:
subr:
arglist = arg()          /* arglist = 3 */
arg1 = arg(1)            /* arg1 = "A" */
arg2_omitted = arg(2,'o')
/*
 * arg2_omitted = 1; second argument is omitted
 */

```

BITAND

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

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

string1 and *string2* are the two strings on 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 is 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 ter-

minates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

Examples:

```
anded = bitand('52'x, '43'x) /* anded = '42'x*/  
  
anded = bitand('52'x, '4343'x)  
/* anded = '4243'x */
```

BITOR

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

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

string1 and *string2* are 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 is 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 terminates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

Examples:

```
ord = bitor('52'x, '43'x') /* ord = '53'x' */  
  
ord = bitor('52x', '4343'x) /* ord = '5343'x' */
```

BITXOR

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

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

string1 and *string2* are 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 is 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 terminates at the end of the shorter string, and the remaining portion of the longer string is appended to the result.

Examples:

```
xord = bitxor('52'x, '43'x) /* xord = '11'x */
```

```
xord = bitxor('52'x, '4343'x)
/* xord = '1143'x */
```

B2X

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

B2X(*string*)

string is the character representation of the binary data to be converted. It may be of any length and may 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 – it is not specified in the form ``1010'b`.

The value returned is a character representation of the equivalent hexadecimal string. It does not contain embedded blanks.

The results of B2X() may be used as the input for the functions X2D() or X2C() to convert binary strings into other representations.

Examples:

```
hexval = b2x('0110 0001')    /* hexval = '61' */  
  
charval = x2c(b2x('01100001'))  
/* charval = 'a' */
```

CENTER

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

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

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

string is 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 at both ends as necessary to fit within the length specified.

pad is the character that occupies character positions at either end of *string*. If *pad* is omitted, the default is blank.

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

Examples:

```
greeting = center('Hello!',10)
/* greeting = "  Hello!  " */

news = center('Headline', 12, '*')
/* news = "***Headline**" */

quote = 'To be or not to be?'
line_length = 18
sayit = center(quote, line_length)
say sayit
/*
 * output from this program fragment is
 *      "To be or not to be"
 */
```

CHANGESTR The CHANGESTR function changes all occurrences of one substring to another within an input string.

CHANGESTR(*from_string*, *input_string*, *to_string*)

from_string is the substring to be changed. If *from_string* is the null string, *input_string* is unchanged.

to_string is the replacement substring.

input_string is the string within which the changes are to be made. If *input_string* is the null string, CHANGESTR returns the null string regardless of the value of *from_string* or *to_string*.

Examples:

```
/*
 * change all occurrences of "which" to "that"
 * in input_string
 */
s="This is the cat which lived in the house",
  "which Jack built"
say changestr('which', s, 'that')
/*
 * output is:
 *
 * This is the cat that lived in the house that
 * Jack built
 */

/*
 * change all the occurrences of 'a' in the
 * input string to blank; this could be done
 * using the literal strings, but this
 * example illustrates the use of other string
 * types and expressions as arguments for
 * changestr()
 */
b = '0010 0000'b
say changestr('61'x, copies('ab', 5), b)
/*
 * output is   " b b b b b"
 */
```

CHARIN

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

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

name is the name of the character input stream. This may 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 must be within the bounds of the input stream specified. If *start* is omitted, the default is the current read position. *start* may **not** 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 may 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 may be used to determine the state of a character stream.

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 wish 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 may 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.

Examples:

```
emp_number = charin('personnel.file',,5)
/*
 * returns 5 characters from the current read
 * position and assigns that value to the
 * variable "emp_number"
 */

/*
 * 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
 */
say 'Enter report number'
num = charin()
address UNIX `lpr report.`num

/*
 * this one line program is named "doit";
 * if you execute it by typing
 * "echo 'abcdefg' | rxx doit"
 * at the UNIX system prompt, the output is
 * "abcde"
 * if you execute it by typing
 * "cat data.file | rxx doit"
 * at the UNIX system prompt, the output is
 * the first 5 characters in the disk file
 * "data.file"
 */
say charin(, ,5)
```

CHAROUT

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

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

name is the name of the character output stream. This may 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 is the character string to be written. If *name* is a persistent stream (usually a disk file), then *string* may 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; the function returns 0
- if *start* is also omitted, CHAROUT closes the output stream; the function returns 0

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

Examples:

```
/*
 * the following program fragment writes the
 * string specified by the variable
 * "emp_number" to the file "personnel.file";
 * rc is normally 0
 */
emp_number = 'DEV003'
rc = charout('personnel.file', emp_number)
```

```
/*
 * the following program fragment writes the
 * string specified by the variable
 * "emp_number" to the file "personnel.file"
 * beginning at the 75th character position;
 * note use of "CALL" to invoke the function
 */
emp_number = 'DEV003'
call charout 'personnel.file', emp_number, 75
```

```
out_rc = charout(, 'Hello world')
/*
 * writes "Hello world" to STDOUT, usually the
 * terminal; out_rc is normally 0
 */
```

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

CHARS

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

CHARS([*name*])

name is the name of the character input stream. This may 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.

When the input stream is a transient stream, CHARS returns 1 if there is any data available in the stream. It returns 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 may 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.

Examples:

```
count = chars('myfile')
/*
 * count is set to the number of characters in
 * the disk file named "myfile"
 */

/*
 * 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 UNIX 'rm myfile'
```

CHDIR

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

CHDIR(*[directory]*)

directory specifies the path to which the current working directory is to be set. *directory* may be any valid directory path on your system. If *directory* is omitted, the default is the path specified by the HOME environment variable.

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 uni-REXX program and has no effect on the current working directory for the program.

Examples:

In the following examples, current directory was `/home/user1` when program was started

```
olddir = getcwd()
cd_rc = chdir('/home/user2')
newdir = getcwd()
say olddir
say newdir
/* the output is
*      /home/user1
*      /home/user2
*/
```

```
olddir = getcwd()
address UNIX `cd /home/user2'
newdir = getcwd()
say olddir
say newdir
/* the output is
*      /home/user1
*      /home user1
*/
```

COMPARE

The COMPARE function determines if two strings are identical.

COMPARE(*string1*, *string2* [, *pad*])

string1 and *string2* are 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.

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

Examples:

```
comp_rc = compare('a', 'a ')
/*
 * 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(q, 'q')
/*
 * 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
 */

a = 'alpha'
b = 'alphabet'
c = compare(a, b)
/*
 * c is 6; pad is omitted so the value of a is
 * padded with blanks, making the string effec-
 * tively "alpha "; the first discrepancy is
 * in position 6, where "a" has a blank and "b"
 * has a "b"
 */
```

CONDITION

The **CONDITION** function returns information about the current trapped condition.

CONDITION(*[option]*)

option specifies the type of information to be returned. If *option* is specified, it must be one of the following:

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 'CALL' or 'SIGNAL'

S (state)

the state of the current trapped condition; this is 'ON', 'OFF', or 'DELAY'

option may be any string beginning with one of the characters shown above. If *option* is omitted, the default value is "I".

The descriptive strings for each condition are as follows:

ERROR and FAILURE

the string that was passed to the external environment and that resulted in the condition being raised

HALT

any string associated with the halt request by the external environment; this may be a null string

LOSTDIGITS

the literal string `NUM', which triggered the
LOSTDIGITS condition

NOVALUE

the derived name of the variable referenced which
raised the condition

NOTREADY

the name of the stream being accessed when the
condition was raised; if this is a default stream,
then a null string is returned

SYNTAX

any string associated with the error by the inter-
preter; this may be a null string

Examples:

```
/*
 * 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
end
say `
say str condition('d')
exit
```

COUNTSTR

The COUNTSTR function returns the number of occurrences of a specified substring within a string.

COUNSTR(*string1*, *string2*)

string1 is the substring to be counted.

string2 is the input string to be searched.

COUNTSTR returns 0 if either *string1* or *string2* is the null string.

Examples:

```
x = countstr('o', 'To be or not to be')
/*
 * x = 4
 */
```

COPIES

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

COPIES(*string*, *n*)

string is the original string to be copied.

n specifies the number of copies of *string* to concatenate. *n* must be a positive number or zero.

Examples:

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

str = '616263'x
newstring = copies(str, 2)
say newstring
/*
 * output from this program fragment is
 * "abcabc"
 */
```

```

do i = 0 to 3
  say copies('ho', i)
end
/* output from this program fragment is
*
*      ho
*      hoho
*      hohoho
* The first line of output is a null string
* since n is 0
*/

/*
* the following program fragment uses COPIES
* to provide leading zeroes so that each
* number is exactly 6 characters long
*/
num.0 = 37
:
:
do i = 1 to num.0
  num.i = copies('0', 6-length(num.i)) || num.i
end

```

CUSERID

The CUSERID function returns the UNIX userid. It is identical to the USERID built-in function.

CUSERID()

Examples:

```

say cuserid()
/*
* displays the userid of the individual
* running the program
*/

/*
* the following program fragment changes the
* current working directory to the user's
* home directory and displays a directory
* list
*/
cd_rc = chdir('/home/'cuserid())
'ls'

```


C2D

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

C2D(*string* [, *n*])

string is the character string to be converted.

If *n* is specified, then *string* is interpreted as a signed number. If the leftmost 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.

Examples:

```
decval = c2d('abc')          /* decval = '979899' */
hexval = d2x(c2d('abc'))     /* hexval = '616263' */
```

C2X

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

C2X(*string*)

string is 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.

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

Examples:

```
hexval = c2x('a')           /* hexval = '61' */
hexval = c2x('61'x)         /* hexval = '61' */
bval = x2b(c2x('a'))       /* bval = '01100001' */
```

DATATYPE

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

DATATYPE(*string* [, *type*])

string is the string for which the data type is to be tested. *type*, if specified, is one of the valid data types.

If *type* is omitted, the function returns the data type of the string as follows:

- `NUM' if *string* is a number that can be added to zero without error
- `CHAR' if *string* does not meet the criteria for `NUM'

If *type* is specified, it must be one of the valid data types from the list below. The function returns 1 if *string* matches the specified type; otherwise, it returns 0.

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 uni-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

Examples:

```
type = datatype('abc')          /* type = 'CHAR' */

val = 10
type = datatype(val)            /* type = 'NUM' */

string = 'April 15'
type = datatype(string, 'A')    /* type = 1 */

/*
 * the following program fragment tests the
 * data type of a variable to determine if it
 * is composed entirely of lowercase charac-
 * ters; if so, the string is converted to
 * uppercase
 */
val = 'abc'
if datatype(val, 'L') = 1 then
    upper_val = translate(val)
```

```

/*
 * 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

/*
 * 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

The DATE function returns the current date or converts dates from one format to another.

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

out_option specifies the format in which the date is returned. 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

If *out_option* is specified, it must be one of the valid date formats from the following list:

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 ddddd.

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 ddddd.

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 *option* is omitted.

O (ordered)

the current date in the format yy/mm/dd.

S (standard)

the current date in the format `yyyymmdd`.

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.

The second and third arguments of `DATE` provide support for converting dates from one format. Date format conversion permits arithmetic operations to be performed on dates of any format.

date_string is 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 date formats described above.

in_option specifies the format of *date_string* and must be one of the date format options described above except Weekday or Month.

Examples:

```
today = date()
/* today = '4 Jul 1994', for example */

thisdate = date('U')
/* thisdate = '07/04/94', for example */

sdate = date('s')
/*
 * sdate = '19940704', for example; dates in
 * this format are suitable for sorting and
 * other ordering operations
 */
```

```

/*
 * the following program fragment converts a
 * date in "normal" US format to a format
 * suitable for sorting
 */
newdate = date('s', '04 Jul 1996', 'n')
/* nesdate is "19960704" */

/*
 * the following program adds 90 days to the
 * current date
 */
today = date()
plus90=date('u', date('b', today, 'n')+90, 'b')
/*
 * if today is 04/30/96, plus90 is "07/29/96"
 */

/*
 * 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'

/*
 * 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

```

```

/*
 * the following program fragment is a slight-
 * ly different approach to the previous
 * example; in this case, however, the subrou-
 * tines 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

```

Note that the literal strings in the INTERPRET statements ("call", etc.) are enclosed in quotes.

DELSTR

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

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

string is 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.

Examples:

```
str = delstr('string', 4)      /* str = 'str' */

airborne = 'paratroops'
infantry = delstr(airborne, 1, 4)
/* infantry = 'troops' */

/*
 * 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
 * 5 digits (as in 60018-6300) have the 6th
 * 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

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

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

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.

Any blanks preceding the first word deleted **are not** removed. Any blanks following the last word deleted **are** removed.

Examples:

```
s = delword('how now brown cow', 2)
/* s = 'how' */
```

```
s = delword('hi there world', 2, 1)
/* s = hi world' */
```

```
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

The DIGITS function returns the current setting of NUMERIC DIGITS.

DIGITS()

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

Examples:

```
x = digits()
/*
 * x = 9 if the default for NUMERIC DIGITS is
 * in effect
 */
```

```

/*
 * 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

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

D2C(*whole-number* [, *n*])

whole-number is 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 is 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.

Examples:

```

charval = d2c(97)           /* charval = 'a' */
charval = d2c(979899)     /* charval = 'abc' */

```

D2X

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

D2X(*whole-number* [, *n*])

whole-number is 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 is 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.

Examples:

```
hexval = d2x(97)                /* hexval = '61' */
bval = x2b(d2x(97))            /* bval = '01100001' */
```

ERRORTTEXT

The ERRORTTEXT function returns the message text associated with the specified uni-REXX error number.

ERRORTTEXT(*n*)

n is a number in the range 0-99. If *n* is not a currently defined uni-REXX error, then ERRORTTEXT returns a null string. If *n* is not within the valid range, then ERRORTTEXT results in Error 40, Incorrect call to routine.

Examples:

```
msg = errortext(11)
/* msg = 'Control stack full' */
```

```

/*
 * 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

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

FIND(*string1*, *string2*)

string1 is the string to be searched. *string2* is the search string.

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 uni-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 insure portability of an application across all implementations of REXX.

Examples:

```
x = find("How now brown cow", "brown cow")
/* x = 3 */

y = find("Once upon a time", "a time")
/* y = 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

The FORM function returns the current setting of NUMERIC FORM.

FORM()

The description of the NUMERIC instruction in *Chapter 4, Instructions* contains information on using NUMERIC FORM to control the form of exponential notation used in the results of arithmetic operations and the evaluation of arithmetic functions.

Examples:

```
expform = form()
/*
 * expform = 'SCIENTIFIC' if the default
 * setting of NUMERIC FORM is in effect
 */

/*
 * the following program fragment insures that
 * NUMERIC FORM is set correctly for this
 * application before proceeding with other
 * operations
 */
if form() \= 'ENGINEERING' then
    numeric form engineering
```

FORMAT

The FORMAT function rounds and formats a number.

FORMAT(*num* [,*before*] [,*after*] [,*expp*] [, *expt*]))

FORMAT first rounds the number using the standard REXX rules that would be applied if the operation “number + 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.

num is the number to be formatted. If no additional arguments are specified, FORMAT simply rounds the number.

before is 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 is the number of places to the right of the decimal point (the decimal portion) of the result. *after* may 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 and *expt* are 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 is 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.

Examples:

```
x = format(12,5)           /* x = '  12' */

/*
 * 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
 */
```

```

/*
 * 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
 */

```

```

/*
 * 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
 */

```

```

/*
 * 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
 */

/*
 * 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

The FUZZ function returns the current setting of NUMERIC FUZZ.

FUZZ()

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

Examples:

```
expfuzz = fuzz()  
/*  
 * expfuzz = 0 if the default setting of  
 * NUMERIC FUZZ is in effect  
 */
```

GETCWD

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

GETCWD()

Using GETCWD is identical to executing a `pwd` host command. It is more convenient setting the value of a variable to the current working directory.

Examples:

```
dir = getcwd()
/*
 * if current directory is /home/user1, then
 *   dir = '/home/user1'
 */

/*
 * the following program fragment is identical
 * to the previous example but requires extra
 * processing
 */
call popen('pwd')
parse pull dir

/*
 * the following program fragment creates an
 * output file name within the current working
 * directory
 */
dir = getcwd()
outfile = dir'/output_file'
```

GETENV

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

GETENV(*string*)

string is 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.

It is recommended that the *string* argument be enclosed in quotes. Without the quotes, *string* is an uninitialized symbol. So long as the symbol remains uninitialized, GETENV behaves as expected since the value of the uninitialized 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*.

GETENV is identical to using the VALUE function with the first argument specified as the name of the environment variable and the third argument specified as 'ENVIRONMENT'. Because VALUE is defined in the ANSI standard, it is more portable than GETENV.

Examples:

```
home = getenv('HOME')
/* home = the current value of $HOME; this is
 * the same value that would result from typing
 * the UNIX command   echo $HOME          */

/*
 * the following program fragment alters the
 * REXXPATH environment variable used for
 * locating external uni-REXX programs called
 * by this program
 */
util_dir = '/home/'userid()'/utilities'
old_path = getenv('REXXPATH')
new_path = util_dir':'old_path
call putenv 'REXXPATH='new_path
```

INDEX

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

INDEX(*string1*, *string2* [, *start*])

string1 is the string to be searched. *string2* is the search string.

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

start is 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.

INDEX is included in uni-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 insure portability of an application across all implementations of REXX.

Examples:

```
where = index('abcdef', 'c')      /* where = 3 */

where = index('abrakadabra', 'a', 5)
/* where = 6 */

/*
 * 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

The INSERT function inserts one string into another string.

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

string1 is the string to be inserted into *string2*.

n is 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 is 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 is the character used to pad *string1* before insertion. If *pad* is omitted, the default pad character is a blank.

Examples:

```
/*
 * 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

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

JUSTIFY(*string*, *length* [, *pad*])

string is the string of blank-delimited words. *length* is the length of the string returned by the function.

pad is the character used to pad string. If *pad* is omitted, the default pad character is a blank.

JUSTIFY is included in uni-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 insure portability of an application across all implementations of REXX.

Examples:

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

LASTPOS

The LASTPOS function finds the last occurrence of one string within another string.

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

string1 is the search string. *string2* is the string to be searched.

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

The function scans backward from the end of *string2*. *start* is the 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*.

Examples:

```
x = lastpos('a', 'abrakadabra')      /* x = 11 */
```

```
x = lastpos('a', 'abrakadabra', 7)  /* x = 6 */
```

```
/*
 * 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 = 'uni-REXX uni-XEDIT uni-SPF'
if lastpos(' ', product_list) = 0 then
    say 'Only one TWG product installed'
else say 'Several TWG products installed'
/*
 * the output is
 *      "Several TWG products installed"
 */
```

LEFT

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

LEFT(*string*, *n* [, *pad*])

string is the original string.

n is 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 is the character used to pad the result. If *pad* is omitted, the default is a blank character.

Examples:

```
x = left('abcdefg', 3)           /* x = 'abc' */

alphabet = left('abc', 26)
/* alphabet = 'abc'           \ */

alphabet = left('abc', 6, '.')
/* alphabet = 'abc...'      */

/*
 * the following program fragment processes an
 * input file by selecting data only from
 * those lines that do not begin with a
 * comment character ('#')
 */
input = '/home/user1/mydata'
j = 1
do lines(input)
  line = linein(input)
  if left(line, 1) \ne '#' then do
    parse var line num.j descr.j .
    j = j + 1
  end
end

/*
 * 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

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

LENGTH(*string*)

string is the string for which the length is to be determined.

Examples:

```
x = length('Hello')                                /* x = 5 */

/*
 * 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

The `LINEIN` function reads a line from a character input stream. It may 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).

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

name is the name of the character input stream. This may 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* may **not** 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.

If a complete line is not available in the stream, the program may 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` may be used to determine the state of a character stream.

Use `LINEIN` to read complete lines that have normal line-end terminators. 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` may leave a persistent output 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.

Examples:

```
emp_record = linein('personnel.file')
/*
 * reads one line from the current read
 * position and assigns that value to the
 * variable "emp_record"
 */
```

```
/*
 * 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 UNIX 'lpr report.'num
```

```
/*
 * the following program fragment processes
 * all lines in an input file, one line at a
 * time
 */
infile = '/home/user1/report'
do i = 1 while lines(infile) > 0
  line.i = linein(infile)
end
```

```
/*
 * this is identical to using execio as
 * illustrated in the following program
 * fragment
 */
infile = '/home/user1/report'
address command
'execio * diskr' infile '(stem line.'
```

LINEOUT

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.

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

name is the name of the character output stream. This may 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 is the character string to be written. If *name* is a persistent stream, then *string* may 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; the function returns 0
- if *line* is omitted, LINEOUT closes the output stream; 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.

The program waits until the write operation is complete. If it is not possible to write the line to the output stream, the function returns 1 (the number of lines not written) and raises the NOTREADY condition.

Use of an I/O function such as LINEOUT may leave a persistent output 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.

Examples:

```
/*
 * the following program fragment writes the
 * string specified by the variable
 * "emp_data" to the file "personnel.file";
 * rc is normally 0
 */
emp_data = 'DEV003 Smith Joe Software Engineer'
rc = lineout('personnel.file', emp_data)
if rc \= 0 then
    say 'Error in writing to personnel file'

out_rc = lineout(, 'Hello world')
/*
 * writes "Hello world" to STDOUT, usually the
 * terminal; out_rc is normally 0
 */

/*
 * the following program fragment writes the
 * lines specified by the compound variables
 * "emp.<n>" to the file "personnel.file";
 * after the last line is written, it closes
 * the file; note the use of "CALL" to invoke
 * the function
 */
outfile = '/home/admin/personnel.data'
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

/*
 * the do loop in the preceding example is
 * identical to using execio as illustrated
 * below
 */
address command
'execio * diskw' outfile '(finis stem emp.'
```


LINES

The LINES function returns the number of complete lines remaining in a character input stream.

LINES(*[name]*)

name is the name of the character input stream. This may 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.

When the input stream is a transient stream, LINES returns 1 if there is any data available in the stream. It returns 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 LINES may leave the file in an open state. Thus, it may be necessary to close the file using LINEOUT, CHAROUT, or STREAM before performing subsequent read or write operations to the file.

Examples:

```
count = lines('myfile')
/*
 * count is set to the number of lines in the
 * disk file named "myfile"
 */

/*
 * 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('myfile') > 0 then
    address UNIX 'rm myfile'
```

```

/*
 * 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

The LOWER function converts characters in a string to lower case.

LOWER(*string*)

string is the string of characters to be converted.
string may be upper-, lower-, or mixed case.

Examples:

```

low = lower('ABCD')           /* low = 'abcd' */

/*
 * the following program fragment converts
 * user input to lower case 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'

```

```

/*
 * the following program is functionally
 * equivalent to the previous example but
 * insures 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

The MAX function returns the largest number in a list of numbers.

MAX(*number* [, *number*] ...)

number is any valid number.

Examples:

```
x = max(10, 12, 9)                                /* x = 12 */
```

```

/*
 * the following program fragment illustrates
 * using MAX to default to the highest index
 * for a compound variable
 */
order_file = '/home/sales/commissions_earned'
address command
'execio * diskr' order_file '(stem comms.'
say 'Enter commission percent'
pull number
tail = max(number, comms.0)
say comms.tail
/*
 * if the user enters a number <= the number
 * of lines read in from the file, the out-
 * put is the value of that line number; if the
 * user enters a number greater than the number
 * of lines in the file, the output is the
 * contents of the last line in the file
 */

```

MIN

The MIN function returns the smallest number in a list of numbers.

MIN(*number* [, *number*] ...)

number is any valid number.

Examples:

```
x = min(10, 12, 9)                                /* x = 9 */

/*
 * 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

The OVERLAY function overlays one string with characters from another string.

OVERLAY(*string1*, *string2* [, [*n*] [, [*length*] [, [*pad*]]])

string1 is the overlay string. *string2* is the original string in which characters are to be replaced by characters from *string1*.

n is 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 is the 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 is the character to be used for padding *string1*. If *pad* is omitted, the default is a blank character.

Examples:

```
str = overlay('old', 'new data')
/* str = 'old data' */

str = overlay('old', 'Some new data', 6)
/* str = 'Some old data' */

str = overlay('change', 'New data', 12, 8, '')
/* str = 'New data***change**' */

/*
 * 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 = '/home/tech/auto_reply_letter'
mail_msg = '/home/tech/msg'
d = "Insert today's date here"
ld = length(d)
do lines(auto_reply)
  line = linein(auto_reply)
  if wordpos(d, line) \= 0 then
    line = overlay(date(), line, pos(d, line), ld)
  call lineout mail_msg, line
end
call lineout mail_msg /*be sure file is closed*/
'mail -s "Auto Reply"' inquirer '<' mail_msg
'rm' mail_msg
```

POPEN

The POPEN function executes a host command and places the results on the uni-REXX program stack. It returns the completion code of the host command.

POPEN(*command* [, *option*])

command is any host command that is valid in the Bourne shell.

option 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`.

POPEN redirects STDOUT to the program stack. Use it

- to capture the output of a host command for subsequent processing
- to execute any host command that may write to STDOUT when you do not wish that output to appear on the terminal screen

Examples:

```
state = popen('test -f myfile')
/*
 * invokes the UNIX test command to check for
 * existence of a file; if the file exists,
 * test sets a completion code of 0 and there-
 * fore state = 0; if the file does not exist,
 * test sets a completion code of 1 and there-
 * fore state = 1
 */
```

```

/*
 * 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
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

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

POS(*string1*, *string2* [, *start*])

string1 is the search string. *string2* is the string to be searched.

POS returns the position of the first character in *string2* where a match is found. If no match is found, POS returns 0.

start is 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.

Examples:

```
where = pos('c', 'abcdef')          /* where = 3 */
```

```
where = pos('a', 'abrakadabra', 5)
/* where = 6 */
```

```

/*
 * 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

The PUTENV function sets the value of an environment variable.

PUTENV(*string*)

string is a Bourne shell command to set the value of an environment variable. The command is of the form

VARIABLE=*value*

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 uni-REXX program is running. Environment variables set by PUTENV are not retained after the uni-REXX program terminates.

PUTENV is equivalent to using the VALUE function with the first argument specified as the environment variable, the second argument specified as its new value, and the third argument specified as 'ENVIRONMENT'. Because VALUE is defined in the ANSI standard, it is more portable than PUTENV.

Examples:

```
rc = putenv('PATH=/tmp:/usr/bin:/home/user1')
/*
 * sets the PATH environment variable; if
 * PUTENV executes successfully, the value of
 * rc is 0; if an error occurs, the value of
 * rc is non-zero
 */

/*
 * the following program fragment checks the
 * current setting of REXXPATH to be sure that
 * it includes all directories needed for
 * finding external programs and modifies it
 * if necessary
 */
needed = '/usr/local/bin/rexx'
current_path = getenv('REXXPATH')
if pos(needed, current_path) = 0 then
    call putenv 'REXXPATH='needed':'current_path'
```

QUALIFY

The QUALIFY function creates a fully qualified name for a filename by prepending the current working directory.

QUALIFY(*filename*)

filename is the name of a file for which a fully qualified name is to be created. A fully qualified name is in the form of a full directory path specification that includes the filename.

If *filename* is omitted, the function returns the full path specification for the current working directory.

Examples:

```
fqn = qualify('InputFile')
/*
 * if the current directory is '/home/user1',
 * then fqn = '/home/user1/InputFile'
 */
```

```
CurrentDir = qualify()  
/*  
 * CurrentDir is possibly '/usr/local/bin'  
 */
```

QUEUED

The QUEUED function returns the number of lines remaining on the uni-REXX program stack.

QUEUED()

Examples:

```
/*  
 * the following program processes every line  
 * remaining on the uni-REXX program stack,  
 * based on some pre-determined criterion  
 */  
do queued()  
  pull nextone  
  if word(nextone, 3) > checkit then call bigger  
  else call smaller  
end
```

```
/*  
 * the following program fragment searches for  
 * the occurrence of a string, specified as a  
 * calling argument, in all C programs in the  
 * current directory  
 */  
parse arg search_string  
call popen 'grep -i' search_string `*.c'  
if queued() = 0 then  
  say 'Search string' search_string 'not found'
```

RANDOM

The RANDOM function returns a quasi-random, non-negative whole number.

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

RANDOM(*max*)

If no arguments are specified, RANDOM returns a value between 0 and 999. The function arguments allow some control over the random number generated.

min and *max* define the inclusive range from which the random number is generated. *min* is the lower value of the range. *min* must be non-negative. If *min* is omitted, the default is 0. *max* is the upper value of the range. *max* must be non-negative. If *max* is omitted, the default is 999.

The magnitude of the range specified may not exceed 100000. Specifically, the following must be true:

$$\mathit{max} - \mathit{min} \leq 100000$$

If only one argument is specified, it is assumed to be *max*. The range is then 0 through the value of *max*.

seed is an initial seed value that may 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 may be time-dependent.

Examples:

```
x = random()                /* possibly, x = 983 */

x = random(9)                /* possibly, x = 2 */

/*
 * 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

The REVERSE function reverses the characters in a string.

REVERSE(*string*)

string is the original string in which the characters are to be reversed.

Examples:

```
str = reverse('string')      /* str = 'gnirts' */
time = reverse('noon ')     /* time = ' noon' */
```

RIGHT

The RIGHT function returns the right-most characters in a string.

RIGHT(*string*, *n* [, *pad*])

string is the original string.

n is 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 is the character used to pad the result. If *pad* is omitted, the default is a blank character.

Examples:

```
x = right('abcdefg', 3)      /* x = 'efg' */
alphabet = right('xyz', 26)
/* alphabet = '                xyz' */
```

```

alphabet = right('xyz', 6, '.')
/* alphabet = '...xyz' */

/*
 * the following program fragment removes
 * 6-character sequence numbers from the
 * beginning of each line of a file
 */
input = '/home/user1/mydata'
output = '/home/user1/data_nonum'
do lines(input)
  line = linein(input)
  line = right(line, length(line)-6)
  call lineout output, line
end
call lineout output

/*
 * 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

The SIGN function returns a value that indicates the sign of a number.

SIGN(*number*)

number is 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.

Examples:

```
x = sign(10)                                /* x = 1 */

/*
 * 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 The SOURCELINE function returns either the number of lines in the current program or the contents of the specified line.

SOURCELINE([*n*])

n is a line number within the range of the current program. *n* must be positive and may not 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.

Examples:

```
prog_length = sourceline()
/*
 * if the current program contains 50 lines,
 * then prog_length = 50
 */
```

```

/*
 * 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 UNIX program_name
:
:
exit
uhoh:
parse value sourceline(sig1) with 'UNIX' failed
say 'Host command failed'
interpret 'say' failed "'not found in $PATH'"
return

```

SPACE

The SPACE function reformats a string of blank-delimited words such that the specified number of pad characters appears between each word.

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

string is the string of blank-delimited words to be formatted.

n is 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 is the character used between each word in the result. If *pad* is omitted, the default pad character is a blank.

Leading and trailing blanks in string are always removed from the result of SPACE.

Examples:

```

x = space('Good morning')
/* x = 'Good morning' */

```

```

/*
 * 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
 *
 */

/*
 * the following program uses SPACE in con-
 * junction with TRANSLATE to remove characters
 * from a string
 */
string = 'work group'
/*
 * translate all blanks to "o" and all "o" or
 * "u" to blank
 */
string = translate(string, 'o', ' ou')
/* remove all blank spaces */
string = space(string, 0)
/*
 * translate all blanks to "o" and all "o" to
 * blank
 */
string = translate(string, 'o', ' o')
/* remove any remaining blank spaces */
string = space(string, 0)
say string
/*
 * the output is
 *
 *      wrkgrp
 */

```


STREAM

The STREAM function is used to determine the state of or to perform an operation on a stream and return the result. Result strings should match those in OS/2 Rexx.

STREAM(*name* [, *operation* [, *strmcmd*]])

name is the name of the stream of interest.

operation is the operation to be performed on the stream. *If operation* is specified, it must be one of the following:

C (command)

the command to execute on this stream, as specified by the *strmcmd* argument

D (description)

a 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

S (state)

the current state of the specified stream; *strmcmd* must **not** be specified; the value returned is one of the following:

ERROR

an erroneous operation has been attempted on the stream

NOTREADY

normal input or output operations would raise the NOTREADY condition

READY

the stream is ready for normal input or output operations

UNKNOWN

the state of the stream cannot be determined

operation may be any string beginning with one of the characters shown above. If ***operation*** is omitted, the default value is “S”.

strmcmd is 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; returns “READY:”

close

close the stream for input or output operations; returns “READY”

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 *size* in number of characters

query datetime [d[t]]

retrieve the date/time stamp of the file with optional formatting; by default the function returns the information in the form mm-dd-yy hh:mm:ss; **d** is one of the date() function formats “bcejnosu”; **t** is one of the time() function formats “cln”; **d** and **t** are optional, but **d** must be specified if **t** is.

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; returns *offset*

= 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

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the STREAM function
 */
strm = '/home/user1/sales.data'
state = stream(strm, 'c', 'query exists')
if state \= '' then
    if stream(strm, 'c', 'open') \= 'READY:' then
        say 'error opening file' strm
    else
        address command 'execio * disk' strm
```

STRIP

The STRIP function removes leading, trailing, or both leading and trailing characters from a string.

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

string is 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* may 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 is the character to be removed from string. If specified, *char* may be only one character. If *char* is omitted, the default is a blank.

Examples:

```
x = strip('  Gypsy Rose  ')
/* x = 'Gypsy Rose' */
```

```

x = strip('000123', 'l') /* x = '123' */

x = strip('In retrospect....', 'Trail', '.')
/* x = 'In retrospect' */

/*
 * the following program fragment removes lead-
 * ing & trailing blanks from a value used
 * as the tail in referencing a compound symbol
 */
pfile = '/u/reports/parms'
do lines(pfile)
  parse value linein(pfile) with arg1 arg2 prog
  prog = strip(upper(prog))
  interpret 'call subr.'prog arg1',' arg2
end

```

SUBSTR

The SUBSTR function returns a sub-string of a string.

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

string is the string from which the sub-string is to be extracted.

n is the 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 or the null string are returned.

length is 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*. If *n* is greater than the length of the string and *length* is omitted, then the null string is returned.

pad is the pad character to be used. If *pad* is omitted, the default is a blank character.

Examples:

```
x = substr('uni-REXX', 5)          /* x = 'REXX' */

herbs = 'parsley sage rosemary thyme'
herb2 = substr(herbs, 9, 4)
/* herb2 = 'sage' */

today = substr(date('u'), 4, 2)
/* today = '18' on the 18th day of any month */

/*
 * the following program fragment extracts a
 * substring 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

The SUBWORD function returns a sub-string from a string of blank-delimited words.

SUBWORD(*string*, *n* [, *length*])

string is the string from which the sub-string is to be extracted.

n is 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 is the 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*.

The result does not include leading or trailing blanks. All blanks between words are preserved in the result.

Examples:

```
n = subword('over the rainbow', 3)
/* n = 'rainbow' */

days = 'Mon Tue Wed Thur Fri Sat Sun'
weekend = subword(days, 6)
/* weekend = 'Sat Sun' */
```

SYMBOL

The SYMBOL function returns the status of a symbol.

SYMBOL(*name*)

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.

SYMBOL returns one of the following :

- | | |
|------------|---|
| BAD | indicates that <i>name</i> is not a valid REXX symbol |
| VAR | indicates that <i>name</i> is a variable (a symbol to which a value has been assigned) |
| LIT | indicates that <i>name</i> is a literal; this could be either a constant symbol or a symbol to which no value has yet been assigned |

Examples:

```
/*
 * 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 */
 */
```

```
/*
 * 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

The TIME function returns the current time of day or converts times from one format to another.

TIME([*out_option* [, *time_string*, *in_option*]])

out_option specifies the format in which the time is returned. If *out_option* is omitted, the format returned is

hh:mm:ss

where

hh	hours; value is from 00 through 23, with leading zeroes
mm	minutes; value is from 00 through 59, with leading zeroes
ss	seconds; value is from 00 through 59, with leading zeroes; fractional seconds are ignored (that is, no rounding occurs)

If *out_option* is specified, it must be one of the valid time formats from the list below.

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 issssss, 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 UNIX 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 `mmm`, 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.

O (offset)

the time offset from GMT in microseconds. This can be used to determine the timezone where the program is being executed.

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.

The second and third arguments of `TIME` provide support for converting times from one format. Time format conversion permits arithmetic operations to be performed on times of any format.

time_string is the time to be converted. It may be a literal string, a variable reference, or an expression that evaluates to a time. It **must** be in one of the time formats described above.

in_option specifies the format of *time_string* and must be one of the time format options described above except Elapsed or Reset.

Examples:

```
now = time()
/* now = '10:30:15', for example          */

cnow = time('c')
/* cnow = '10:30am', for example         */

/*
 * 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 UNIX prog
  prog_time = time('e')
  say 'Time to run' prog ':' prog_time
end
exit

now = time('c', '17:17:00', 'n')
```

```

/* now = '5:17pm' */

plus45 = time('c', time('m') + 45, 'm')
/*
 * if it's currently 4:40pm, plus45 = '5:25pm'
 */

```

TRACE

The TRACE function returns the current setting of TRACE. It may also be used to change the TRACE setting.

TRACE([*option*])

If *option* is omitted, the function simply returns the current TRACE setting. The description of the TRACE instruction in *Chapter 4, Instructions* provides details on the possible settings.

option is one of the valid TRACE settings as described in *Chapter 4, Instructions*. Valid trace settings are A, C, E, F, I, L, N, O, R. *option* may also include the “?” prefix.

Using the TRACE function to change the setting differs from use of the TRACE instruction in the following ways:

- the function alters the trace action even if interactive tracing is in effect
- the setting specified by the function may not be a number

Examples:

```

setting = trace()
/* setting = 'N', for example */

/*
 * 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

The TRANSLATE function translates the characters in a string according to the specified translation tables.

TRANSLATE(*string* [, [*out_tbl*] [, [*in_tbl*] [, *pad*]]])

string is the string to be translated.

out_tbl is the set of characters used in the result string.

in_tbl is the set of characters from the original string that are to be translated in the result.

There is a one-to-one correspondence between characters in the two tables. Thus, the first character in the input table is translated to the first character in the output table and so forth. If *out_tbl* contains more characters than *in_tbl*, *out_tbl* is truncated to the same length as *in_tbl*. If *out_tbl* contains fewer characters than *in_tbl*, *out_tbl* is padded to the same length as *in_tbl*.

If *out_tbl* is omitted, the default is the null string. If *in_tbl* is omitted, the default is the range of characters from `00'x to `ff'x, inclusive. If both *out_tbl* and *in_tbl* are omitted, *string* is translated to uppercase. If *in_tbl* contains duplicate characters, the first occurrence of a character is used to determine the output.

pad is the character used to pad *out_tbl*, if necessary. If *pad* is omitted, the default is a blank.

Examples:

```

upper_str = translate('Hello')
/*
* upper_str = 'HELLO' ; this is a fully
* portable equivalent to the uni-REXX UPPER
* function, which may not be available in

```

```
* other REXX implementations
*/
```

```

/*
 * the following program fragment converts
 * a string to lowercase; this is a fully
 * portable equivalent to the uni-REXX LOWER
 * function, which may not be available in
 * other REXX implementations
 */
parse arg string
uppers = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ'
lowers = 'abcdefghijklmnopqrstuvwxyz'
lstring = translate(string, lowers, uppers)

intab = 'abcdefgh'
pattern = 'ef/gh/abcd'
reorder = translate(pattern, '19940704', intab)
/* reorder = '07/04/1994' */

```

TRUNC

The TRUNC function returns the integer portion of a number and, optionally, a specified number of decimal places.

TRUNC(*number* [, *n*])

number is the numeric value to be truncated.

n is the number of decimal positions in the result.

n must be non-negative. If *n* is omitted, the default value is 0.

The result of TRUNC is never expressed in exponential notation.

Examples:

```
x = trunc(3.1416)                                /* x = 3 */
```

```
y = trunc(3.1416, 2)                             /* y = 3.14 */
```

```
z = trunc(3.14, 3)                               /* z = 3.140 */
```

UPPER

The UPPER function converts characters in a string to uppercase.

UPPER(*string*)

string is the string of characters to be converted.
string may be upper-, lower-, or mixed case.

Examples:

```
up = upper('abcd')           /* up = 'ABCD' */

up = upper('Hello world')
/* up = 'HELLO WORLD'      */

/*
 * the following program fragment insures 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

The USERID function returns the UNIX userid. It is identical to the CUSERID built-in function.

USERID()

Examples:

```
say userid()
/*
 * displays the userid of the individual
 * running the program
 */
```

```

/*
 * the following program fragment changes the
 * current working directory to the user's
 * home directory and displays a directory
 * list
 */
cd_rc = chdir(`/home/'userid())
`ls'

```

VALUE

The VALUE function returns the value of a symbol and, optionally, assigns a new value to that symbol.

VALUE(*name* [, [*new_value*] [, *pool*]])

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.

new_value is the new value to be assigned to *name*. When VALUE is called with second argument specified, the function returns the original value of *name* prior to the assignment of the new value. Subsequent calls to VALUE return the new value.

pool identifies the external variable pool to be searched for *name*. The names of external variable pools are implementation-specific. uni-REXX supports `ENVIRONMENT' to identify UNIX environment variables. For purposes of application portability, uni-REXX recognizes `OS2ENVIRONMENT' and `DOSENVIRONMENT' as aliases for `ENVIRONMENT'.

Examples:

```

x = 10
say value(`x')
/* the output is "10" */

```



```

x = 10
y = 'x'
say value(y)
/* the output is "10" */

x = 10
say value(x)
/*
 * 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 = qqq
qqq = 10
y.10 = 'hello'
y.x = 'goodbye'
say value('y.x')
say value(y.x)
say value('y.' || x)
/*
 * the output is
 *      goodbye
 *      GOODBYE
 *      hello
 */

/*
 * get the current setting of the REXXPATH
 * environment variable - possibly
 * '/usr/local/bin:/home/user1/funcls'
 */
rpath = value('REXXPATH', , 'ENVIRONMENT')

/*
 * add the current working directory to the
 * setting of the REXXPATH environment variable;
 * the first argument is 'REXXPATH'; the second
 * argument (new value) is an expression com-
 * posed of the value function to retrieve the
 * current setting concatenated with the literal
 * string ':' concatenated with the qualify
 * function to retrieve the current working
 * directory; the third argument is the pool
 * name 'ENVIRONMENT'); the comma at the end of
 * each line in this example is for continuation
 */
nrp=value('REXXPATH', ,
          value('REXXPATH', , 'ENVIRONMENT'):',',
          qualify(), 'ENVIRONMENT')

```

VERIFY

The VERIFY function verifies whether or not a string is composed only of characters in a specified character list.

VERIFY(*string*, *char_list* [, [*option*] [, *start*]])

string is the string to be verified.

char_list is the list of acceptable characters.

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.

option controls whether the function verifies the presence or absence of characters in *char_list*. *option* may be one of the following:

Match the function returns the position of the first character in *string* that is present in *char_list*

Nomatch the function returns the position of the first character in *string* that is **not** present in *char_list*

option may be any string beginning with the character `M' or `N', in any case. If *option* is omitted, the default is Nomatch.

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.

Examples:

```
x = verify('abc', 'abcdefg')            /* x = 0 */
```

```

x = verify(abc, 'abcdefg')
/*
 * x = 1; the value of the symbol abc is "ABC",
 * and none of these characters is in "abcdefg"
 */

/*
 * the following program fragment verifies that
 * all date values in a file contain only num-
 * bers or slash before processing the file
 */
infile = '/home/shipping/orders'
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

/*
 * the following program fragment verifies that
 * employee numbers include a valid department
 * designator in position >=6 before proceeding
 */
infile = '/home/admin/personnel'
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
call lineout infile
if bad_data then do
    say 'Found some invalid employee numbers'
    exit
end
else call do_payroll

```

WORD

The WORD function returns a single word from a string of blank-delimited words.

WORD(*string*, *n*)

string is the string of blank-delimited words.

n is 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.

Examples:

```
x = word('Happy New Year', 2)    /* x = 'New' */

/*
 * 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 The **WORDINDEX** function returns the character position of the start of a specified word in a string of blank-delimited words.

WORDINDEX(*string*, *n*)

string is the string of blank-delimited words.

n is 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 0.

Examples:

```
x = wordindex('Happy New Year', 2) /* x = 7 */

/*
 * 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

The WORDLENGTH function returns the length of a specified word in a string of blank-delimited words.

WORDLENGTH(*string*, *n*)

string is the string of blank-delimited words.

n is the number of the word for which the 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.

Examples:

```
x = wordlength('Happy New Year', 2) /* x = 3 */

/*
 * 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

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

WORDPOS(*string1*, *string2* [, *start*])

string1 is the search string. *string2* is the string to be searched. Multiple blanks between words in both *string1* and *string2* are treated as single blanks 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.

start is 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.

Examples:

```
z = wordpos('time', 'time and time again')
/* z = 1 */

z = wordpos(time, 'Time flies') /* z = 0 */

a = 'the best of times'
b = 'It was the best of times'
c = wordpos(a, b) /* c = 3 */

a = 'the best of times, the worst of times'
b = 'times'
say wordpos(b, a, 5)
/* the output is "8" */

/*
 * the following program fragment uses
 * WORDPOS to verify user input
 */
prod_list = 'uni-REXX uni-XEDIT uni-SPF'
say 'Name a TWG product'
parse pull answer
if wordpos(answer, prod_list) = 0 then
    say "Sorry, that product's not from TWG"
```

WORDS

The WORDS function returns the number of words in a string of blank-delimited words.

WORDS(*string*)

string is the string of blank-delimited words.

Examples:

```
x = words('Hip, hip, hooray')          /* x = 3 */

/*
 * the following program fragment processes a
 * file, discarding all blank lines
 */
file = '/home/acctg/report.list'
do lines(file)
  line = linein(file)
  if words(line) \= 0 then call reports line
end
```

XRANGE

The XRANGE function returns a string of all the valid character encodings within a range.

XRANGE([*start*] [, *end*])

start is the beginning of the range. If *start* is omitted, the default value is ``00'x`.

end is the end of the range. If *end* is omitted, the default is ``ff'x`.

If *start* is greater than *end*, then the result will automatically wrap from ``ff'x` to ``00'x`.

Examples:

```
x = xrange('m', 'r')                  /* x = 'mnopqr' */
```



```

y = xrange('fa'x, '04'x)
say y
/*
 * the output is the character representation
 * of the hexadecimal string
 *      'fafbfcfdfeff01020304'x
 */

a = x2c(b2x('01100011'))
b = d2c(112)
say xrange(a, b)
/*
 * the output is
 *      cdefghijklmnop
 */

```

X2B

The X2B function converts a string of hexadecimal characters to a string of binary characters.

X2B(*string*)

string is a string of hexadecimal characters. It is not a hexadecimal string – that is, it is not represented in the form ``nnnn'x`.

You may use X2B in combination with other conversion functions to convert various formats to their equivalent binary value.

Examples:

```

x = x2b('63')           /* x = '01100011' */
y = x2b(c2x('a'))     /* y = '01100001' */

```

X2C

The X2C function converts a string of hexadecimal characters to character format.

X2C(*string*)

string is a string of hexadecimal characters. It is not a hexadecimal string – that is, it is not represented in the form ``nnnn'x`. *string* may contain embedded blanks, which are ignored, between pairs of characters. 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.

Examples:

```
x = x2c('616263')           /* x = 'abc' */
```

```
say x2c('f')
/*
 * the output is the character representation
 * of '0f'x
 */
```

```
z = x2c(d2x('112'))         /* z = 'p' */
```

X2D

The X2D function converts a string of hexadecimal characters to its decimal equivalent.

X2D(*string* [, *n*])

string is a string of hexadecimal characters. It is not a hexadecimal string – that is, it is not represented in the form ``nnnn'x`. *string* may contain embedded blanks, which are ignored, between pairs of characters. If *string* is null, the function returns 0.

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.

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, Error 40, Incorrect call to routine, results.

Examples:

```
x = x2d('76')                               /* x = '112' */
y = x2d(b2x('01100011'))                    /* y = 99 */
z = x2d(b2x('01100001'),1)                  /* z = 1 */
q = x2d('f063', 4)                          /* q = -3997 */
```


Chapter 6: uni-REXX Extensions

A number of types of language extensions are implemented in uni-REXX:

- built-in functions that allow a program to manage its environment
- uni-REXX specific functions, to implement system functions in an operating system independent manner
- implementations of UNIX-specific functions normally available only in the C library
- facilities that are available as part of the operating system in other environments where the REXX language is implemented

The first type of built-in functions are documented in *Chapter 5, Built-In Functions*. The other types of extensions are documented in this chapter.

uni-REXX Specific Functions

These functions are either specific to uni-REXX, or were drawn from the OS/2 implementation of the REXX language. They are intended to implement operating system level functionality, in an operating system independent manner. The dynamic definition and loading of user-written commands and functions is supported here. In addition, creation and control of external command processes, and other functions to assist in writing cross-platform rexx programs are defined.

RXFUNCADD RXFUNCADD registers a dynamically loaded function name, making it available to REXX programs. RXFUNCADD returns 0 if the function was registered successfully, 1 if not.

RXFUNCADD(*name*, *module*, *procedure*)

name is the name of the function to be registered.

module is the name of the library where the registered function may be found. The library name extension (.a, ..so, or .sl) should be omitted from the function argument. The library must be found in the concatenation defined by the REXXLOADPATH environment variable.

procedure for compatibility with OS/2 REXX, a duplicate of *name*.

RXFUNCDROP RXFUNCDROP deregisters a function name, previously added via RXFUNCADD. The function will no longer be available to REXX programs. RXFUNCDROP returns 0 if the function was deregistered successfully, 1 if not.

RXFUNCDROP(*name*)

name is the name of the function to be deregistered.

RXFUNCQUERY RXFUNCQUERY determines whether a function name is registered. It returns a value of 0 if the function is available, a value of 1 if it is not.

RXFUNCQUERY(*name*)

name is the function name which is to be checked.

RXXCOMMANDSPAWN

start an external command process

RXXCOMMANDSPAWN(*command*)

command is a literal value which represents a command to be passed to the default external command environment for execution.

Under UNIX systems, the return value is the process id of the process created by this command. This process id may be used by the functions RXXCOMMANDKILL and RXXCOMMANDWAIT as appropriate.

Returns:

the pid of the process spawned.

RXXCOMMANDKILL

kill a spawned process

RXXCOMMANDKILL(*pid*)

pid is the process id of the process to be terminated. The pid may be saved from the RXXCOMMANDSPAWN function, if the process was created by that function.

RXXCOMMANDWAIT

wait for process completion

RXXCOMMANDWAIT(*pid*)

pid is the process id of the process to wait on.

RXXOSENDOFLINESTRING

return end of line characters

RXXOSENDOFLINESTRING()

For UNIX systems, this will return a carriage return/line feed sequence.

RXXOSENVIROMENTSEPARATOR

return environment separator

RXXOSENVIROMENTSEPARATOR()

RXXOSPATHSEPARATOR

return file path separator

RXXOSPATHSEPARATOR()

RXXSLEEP

Sleep the program for a number of seconds

RXXSLEEP(*number*)

number is the decimal number of seconds to sleep.

RXXFUNCTIONPACKAGELOAD

The RXXFUNCTIONPACKAGELOAD function is used to load external native binary function packages from shared libraries (usually written in C) and make them available within uni-REXX.

RXXFUNCTIONPACKAGELOAD(*module* [,*package*])

module is the file name of the shared library function package to load. The file's suffix (ie .so) is optional. If not specified, uni-REXX will append the correct suffix type for the current platform. Since the suffix for shared library modules varies among different UNIX platforms, it's recommended that the suffix not be specified to enhance portability (see below).

package is the name of the function package table. If not specified the default is "irxpackt." In the C module, this function package table points to one or more function packages which in turn point to the functions themselves.

Returns:

0 if successful, non-0 otherwise.

Example:

```
if 0 /= rxxfunctionpackageload("funcpackage" , ,
    "FunctionPackageTable") then
    do
        say "Error - could not load function package."
        exit 1
    end
call function1 /* call the loaded function */
```

Comments:

Loaded function packages are available globally. They are accessible to any subroutine. They also remain available across API invocations of uni-REXX with `irxexec()`, `irxjcl()` and `RexxStart()`.

The environment variable `REXXLOADPATH` should be used to point to the paths where the shared library function packages(s) reside. Various platforms have their own environment variable used to locate shared libraries (`LD_LIBRARY_PATH` etc), but it is recommended that you use `REXXLOADPATH` since it is platform independent.

Unfortunately, the steps required to build a shared library differ from platform to platform. Here are a few suggestions, but be sure to consult your system documentation since the exact option flags used are known to change between releases.

```
#
# For Solaris shared libraries are '.so' files:
#
cc -KPIC -c funcpackage.c
ld -o funcpackage.so -G funcpackage.o

#
# For AIX shared libraries are '.a' files:
#
cc -c funcpackage.c
ld -bM:SRE -bnoentry -bE:funcpackage.exp
funcpackage.o -o funcpackage.a

#
# For HP-UX shared libraries are '.sl' files:
#
cc +z -c funcpackage.c
ld -b -bnoentry funcpackage.o -o funcpackage.sl
```

In the native C code, the function package table is declared as `FPCCKDIR *[]`, while the function packages are declared as `FPCCKDIR[]`. The functions themselves accept an `ARGLIST` and an `EVALBLOCK` as parameters. For example here are two function packages pointed to by a function package table:

```

/*
 * FunctionPackageTable ↔ FunctionPackage1 ↔ Function1
 *                               |
 *                               ↔ Function2
 *                               |
 *                               ↔ Function3
 *
 *                               ↔ FunctionPackage2 ↔ FunctionA
 *                                               |
 *                                               ↔ FunctionB
 *                                               |
 *                                               ↔ FunctionC
 */
static int Function1(ARGLIST *arglist,
                    EVALBLOCK *evalblock)
{
    ...
    /* function body */
    ...
}
...
/* Function2, Function3, FunctionA, FunctionB,
FunctionC */
...
FPCKDIR FunctionPackage1[] =
{
    Function1,
    Function2,
    Function3
    NULL
};

FPCKDIR FunctionPackage2[] =
{
    FunctionA,
    FunctionB,
    FunctionC
    NULL
};

FPCKDIR *FunctionPackageTable[] =
{
    FunctionPackage1,
    FunctionPackage2
    NULL
};

/* end of code */

```

Under AIX, the package name must be exported from the shared library.

RXXCOMMANDPACKAGELOAD

The function RXXCOMMANDPACKAGELOAD is used to load external native binary command packages from shared libraries (usually written in C) and make them available within uni-REXX.

RXXCOMMANDPACKAGELOAD(module [,package])

module is the file name of the shared library command package to load. The file's suffix (ie .so) is optional. If not specified, uni-REXX will append the correct suffix type for the current platform. Since the suffix for shared library modules varies among different UNIX platforms, it's recommended that the suffix not be specified to enhance portability (see below).

package is the name of the command package. If not specified the default is "irxpackt." In the C module, this command package points to the commands themselves.

Returns:

0 if successful, non-0 otherwise.

Example:

```
if 0 <> rxxcommandpackageload("cmdpackage", ,
"CommandPackageTable") then
  do
    say "Error - could not load command package."
    exit 1
  end
address command "command1" /* run the loaded
command */
```

Comments:

Unlike normal commands that are executed from the command shell or through the “ADDRESS UNIX ...” instruction, command packages can only be executed from the uni-REXX “ADDRESS COMMAND ...” instruction.

Commands loaded from command packages execute more quickly than normal commands since they are not run in a separate process or shell. In fact, command packages were first incorporated into uni-REXX to provide faster EXECIO and GLOBALV commands.

Loaded command packages are available globally. They are accessible to any subroutine. They also remain available across API invocations of uni-REXX with `irxexec()`, `irxjcl()` and `RexxStart()`.

The environment variable `REXXLOADPATH` should be used to point to the paths where the shared library command packages(s) reside. Various platforms have their own environment variable used to locate shared libraries (`LD_LIBRARY_PATH` etc), but we recommend that you use `REXXLOADPATH` since it is platform independent.

Unfortunately, the steps required to build a shared library differ from platform to platform. Here are a few suggestions, but be sure to consult your system documentation since the exact option flags used are known to change between releases.

```
#
# For Solaris shared libraries are '.so' files:
#
cc -KPIC -c cmdpackage.c
ld -o cmdpackage.so -G cmdpackage.o
#
# For AIX shared libraries are '.a' files:
#
cc -c cmdpackage.c
ld -bM:SRE -bnoentry -bE:cmdpackage.exp
cmdpackage.o -o cmdpackage.a
#
# For HP-UX shared libraries are '.sl' files:
#
cc +z -c cmdpackage.c
ld -b -bnoentry cmdpackage.o -o cmdpackage.sl
```

In the native C code, the command package is declared as `CPCCKDIR[]`. The commands themselves accept an `int` and a `char**` as parameters (just like a main program). For example here's a command package:

```
/*
 * CommandPackage +--> Command1
 *                |
 *                +--> Command2
 *                |
 *                +--> Command3
 */

static int Command1(int argc,
                    char **argv)
{
    ...
    /* command body */
    ...
}
...
/* Command2, Command3 */
...

CPCCKDIR CommandPackage[] =
{
    Command1,
    Command2,
    Command3
    NULL
};

/* end of code */
```

As noted above, the commands have the same calling sequence as a “main()” program. This makes converting standalone commands into command packages easy.

Note that commands in command packages don't have their static variables re-initialized between invocations. It's necessary to perform all required initializations in executable code and not in declaration initializers. For example:

```
static int x = 0; /* Is not initialized */
                /* on each entry.      */

static int x; /* Initializer not needed */
    ...
    x = 0; /* Initialize at start-up */
```

Under AIX, the package name must be exported from the shared library.

UNIX-Specific Functions

The UNIX-specific functions of uni-REXX provide a rich set of facilities for interaction with the operating system environment, both on a local host and across a network. They include functions for

- process management
- configuration management
- file and directory management
- system error processing
- regular expression processing
- interprocess communication

All UNIX-specific function names begin with the underscore character (_). Like the built-in functions, the syntax diagrams show the function name in uppercase; however, the name may appear in a program in any case.

The following table lists the UNIX-specific functions by category. Subsequent sections document the functions in alphabetical order.

Process Management

_EXIT
_FORK
_GETPID
_GETPPID
_KILL
_SETSID *
_SLEEP
_WAIT
_WAITPID *

Configuration Management

_GETEUID
_GETHOSTBYADDR
_GETHOSTBYNAME
_GETHOSTID *

*

Not available in all UNIX implementations

**_GETHOSTNAME
_GETSERVBYNAME
_GETUID
_SYSTEMDIR**

File and Directory Management

**_CLOSEDIR
_IOCTL
_OPENDIR
_READDIR
_STAT
_TRUNCATE
_UMASK**

System Error Processing

**_ERRNO
_SYS_ERRLIST**

Regular Expression Processing

_REGEX

Interprocess Communications

**_ACCEPT
_BIND
_CLOSESOCKET
_CONNECT
_FD_CLR
_FD_ISSET
_FD_SET
_FD_ZERO
_GETHOSTBYADDR
_GETHOSTBYNAME
_GETHOSTNAME
_GETPEERNAME
_GETSERVBYNAME
_GETSOCKNAME
_GETSOCKOPT
_LISTEN**

_RECV
_SELECT
_SEND
_SETSOCKOPT
_SOCKET

The documentation of these functions makes reference to both Internet domain and UNIX domain sockets. Some UNIX implementations do not support UNIX domain sockets. Refer to your operating system man page for “socket” to determine which domain types are supported on your system.

Unless noted otherwise, these functions return a negative value and set the appropriate system error number when an error occurs. The system error number and its associated text can be retrieved with the `_ERRNO` and `_SYS_ERRLIST` functions, respectively.

`_ACCEPT`

The `_ACCEPT` function accepts a connection on a socket. It provides the same service as the `accept(2)` system call.

`_ACCEPT(sockhandle, [sockaddr] [, addrlen])`

sockhandle is the handle of a socket that has been created with the `_SOCKET` function, has been bound to an address with the `_BIND` function, and is listening for connections by means of the `_LISTEN` function.

sockaddr is a stem that receives address information about the current connection when `_ACCEPT` is executed. When the function returns, one or more of the following compound variables is set:

sockaddr.SA_FAMILY

the type of socket; it is one of the following:

“AF_INET” if the connection is from an Internet domain socket

“AF_UNIX” if the connection is from a UNIX domain socket

sockaddr.SIN_ADDR

the binary Internet address of the socket; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SIN_PORT

the numeric Internet port; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SUN_PATH

the path name of the socket; set only if *sockaddr.SA_FAMILY* is “AF_UNIX”; on some UNIX implementations, this value may not be set

If *sockaddr* is omitted, the information returned by `_ACCEPT` is discarded.

addrlen is the name of a variable that receives the length of the socket address. If ***addrlen*** is omitted, the value is discarded.

`_ACCEPT` extracts the first pending connection from the queue. It creates a new socket with the same properties as ***sockhandle*** and assigns a new file descriptor to that socket. This new socket is used to read and write data from the connecting socket; it does not accept new connections. The socket identified by ***sockhandle*** continues to accept connections and create new sockets for each new connection.

The function is typically invoked as

```
var = _accept(sockhandle)
```

var contains the file descriptor associated with the newly created socket.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the _accept function; for simpli-
 * city of illustration, it does not include
 * the error checking that would normally be
 * present in a robust program
 */
s1 = _socket('AF_INET', 'SOCK_STREAM')
net.sa_family = 'AF_INET'
net.sin_port = 11111
bindrc = _bind(s1, 'net.')
listenrc = _listen(s1)
do forever
/*
 * accept the next pending connection; the new
 * socket created for read/write purposes is
 * "s2"
 */
    s2 = _accept(s1)
    :
    :
    closerc = _closesocket(s2)
end
crc = _closesocket(s1)
```

`_BIND`

The `_BIND` function binds a name to a socket. It provides the same service as the `bind(2)` system call.

`_BIND(sockhandle, sockaddr)`

sockhandle is the handle of a socket that has been created with the `_SOCKET` function.

sockaddr is the name of a stem used for socket address data. The following compound variables should be set as indicated before the `_BIND` function is invoked:

sockaddr.SA_FAMILY

the type of socket; it may be a literal string or an expression that evaluates to a literal string; the only permitted values are

“AF_INET” if the socket is an Internet domain socket

“AF_UNIX” if the socket is a UNIX domain socket

This value may be specified in any case. It must match the first argument used in the `_SOCKET` function call that created this socket.

sockaddr.SIN_ADDR

the binary Internet address; this variable must be set for Internet domain sockets; it must be a valid Internet address or “INADDR_ANY” ; typically, it is the value returned by the `_GETHOSTBYNAME` function in the variable *hostentry.H_ADDR*

sockaddr.SIN_PORT

the numeric Internet port; this variable must be set for Internet domain sockets; it may be a literal string or an expression that evaluates to a valid port number, such as the value returned by the `_GETSERVBYNAME` function in the variable *serverentry.S_PORT*

sockaddr.SUN_PATH

the path name of the socket; this variable must be set for UNIX domain sockets; it may be a literal string or an expression that evaluates to a valid path name

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the _bind function; for simpli-
 * city of illustration, it does not include
 * all of the error checking that would normal-
 * ly be present in a robust program nor does
 * it include the processing that normally
 * follows a bind
 */
domain = 'AF_INET'
s1 = _socket(domain, 'SOCK_STREAM')
net.sa_family = domain
net.sin_port = 11111
bindrc = _bind(s1, 'net.')
if bindrc < 0 then call error1
```

_CLOSEDIR

The `_CLOSEDIR` function closes an open directory descriptor. It provides the same service as the `closedir(3)` library function.

_CLOSEDIR(*dirhandle*)

dirhandle is the handle returned by a previous call to `_OPENDIR`.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of _opendir, _readdir, and _closedir
 * to obtain directory information; it prints
 * the first 5 directory entries
 */
dir = _opendir('/home/user1')
if dir \= 0 then do 5
    line = _readdir(dir)
    say line
end
call _closedir dir
```

_CLOSESOCKET

The `_CLOSESOCKET` function deletes a socket descriptor.

_CLOSESOCKET(*sockhandle*)

sockhandle is the handle of a socket that has been created with the `_SOCKET` function or the `_ACCEPT` function.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the _closesocket function; for
 * simplicity of illustration, it does not
 * include the error checking that would
 * normally be present in a robust program
 */
s1 = _socket('AF_INET', 'SOCK_STREAM')
net.sa_family = 'AF_INET'
net.sin_port = 11111
bindrc = _bind(s1, 'net.')
listenrc = _listen(s1)
do forever
/*
 * accept the next pending connection; the new
 * socket created for read/write purposes is
 * "s2"
 */
    s2 = _accept(s1)
    :
    :
    closerc = _closesocket(s2)
end
crc = _closesocket(s1)
```

_CONNECT

The `_CONNECT` function initiates a connection on a socket. It provides the same service as the `connect(2)` system call.

`_CONNECT(sockhandle, sockaddr)`

sockhandle is the name of a socket that has been created with the `_SOCKET` function.

sockaddr is the name of a stem used for socket address data for the socket to which you wish to connect. The following compound variables should be set as indicated before the `_CONNECT` function is invoked:

sockaddr.SA_FAMILY

the type of socket; it may be a literal string or an expression that evaluates to a literal string; the only permitted values are

“AF_INET” if the socket is an Internet domain socket

“AF_UNIX” if the socket is a UNIX domain socket

This value may be specified in any case. It must match the first argument used in the `_SOCKET` function call that created this socket.

sockaddr.SIN_ADDR

the binary Internet address; this variable must be set for Internet domain sockets; it must be a valid Internet address or “INADDR_ANY” ; typically, it is the value returned by `_GETHOSTBYNAME` function in the variable *hostentry.H_ADDR*

sockaddr.SIN_PORT

the numeric Internet port; this variable must be set for Internet domain sockets; it may be a literal string or an expression that evaluates to a valid port number, such as the value returned by `_GETSERVBYNAME` function in the variable *serverentry.S_PORT*

sockaddr.SUN_PATH

the path name of the socket; this variable must be set for UNIX domain sockets; it may be a literal string or an expression that evaluates to a valid path name

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the _connect function; for
 * simplicity of illustration, it does not
 * include all of the error checking that
 * would normally be present in a robust
 * program
 */
net.sa_family = domain
net.sin_port = 11111
net.sin_addr = `INADDR_ANY`
socket1 = _socket("AF_INET", "SOCK_STREAM")
crc = _connect(socket1, `net.`)
```

_ERRNO

The `_ERRNO` function returns the UNIX error number from the last invocation of a UNIX-specific function. It is equivalent to the `_errno` external system variable.

_ERRNO()

The value returned by `_ERRNO` can be used as an index into the table of system error messages to retrieve the text associated with this error number.

Examples:

```
s1 = _socket(`AF_INET`, `SOCK_STREAM`)
if s1 < 0 then
  say `Error` _errno()':': _sys_errlist(_errno())
/*
 * if the socket creation fails, the system
 * error number and its associated message is
 * displayed
 */
```

_EXIT

The `_EXIT` function causes the current process to terminate with the specified status code. It provides the same service as the `exit(2)` system call.

_EXIT(status)

status is the value to be returned from the process that is being terminated.

Examples:

```
/* the following program fragment illustrates
 * using _exit in an error trapping routine to
 * terminate the entire process if an error
 * occurs
 */
frc = _fork()
if frc < 0 then call error1 'fork'
s1 = _socket('AF_INET', 'SOCK_STREAM')
if s1 < 0 then call error1 'socket'
:
:
exit
error1:
say arg(1) 'error' _errno()''
say _sys_errlist(_errno())
call _exit(4)
```

_FD_CLR

The `_FD_CLR` function returns a `_SELECT` mask with the handle's bit turned off.

_FD_CLR(handle, mask)

handle is the file handle used with this mask.

mask is the mask to be modified. It must have been previously created using the `_FD_ZERO` function.

_FD_ISSET

The `_FD_ISSET` function determines whether the handle's bit is turned on in a specified `_SELECT` mask.

`_FD_ISSET(handle, mask)`

handle is the file handle used with this mask.

mask is the mask to be tested. It must have been previously created using the `_FD_ZERO` function and may have been modified using either the `_FD_CLR` or `_FD_SET` function.

_FD_SET

The `_FD_SET` function returns a mask with the handle's bit turned on.

`_FD_SET(handle, mask)`

handle is the file handle used with this mask.

mask is the mask to be modified. It must have been previously created using the `_FD_ZERO` function.

_FD_ZERO

The `_FD_ZERO` function creates an empty or zero-filled mask for use with the `_SELECT` function.

`_FD_ZERO()`

_FORK

The `_FORK` function creates a new process. It provides the same service as the `fork(2)` system call.

`_FORK()`

The new process (child) is identical to the calling process (parent) with the following exceptions:

- the child process has a unique process id
- the child process has a unique parent process id (the process id of the calling process)

- the child process has its own copy of the parent's descriptors

Examples:

```

/*
 * the following program fragment illustrates
 * the use of _fork to spawn a new process
 */
frc = _fork()
if frc < 0 then call error1
if frc \= 0 then do
    say 'Child process started:  PID =' frc
    exit
end

```

_GETEUID

The `_GETEUID` function returns the effective userid of the current process. It provides the same service as the `geteuid(2)` system call.

_GETEUID()

The effective userid may not be the userid under which the user who started the process logged in. If the user has executed an “su” command, the effective userid results from execution of that “su” command. The user’s real userid remains the login id. Use the `_GETUID` function to retrieve the real userid.

The function returns the user’s numerical id. On some systems, this is the value of the third field in the `/etc/passwd` file.

Examples:

```

say _geteuid()
/* for a user logged in as "user1" whose
 * numeric id is "1010", if no UNIX
 * commands are executed before running this
 * program, the output is "1010"; if user1
 * first types "su root" and gives the proper
 * password so that he is now effectively
 * superuser, the output is "0"
 */

```

_GETHOSTBYADDR

The `_GETHOSTBYADDR` function retrieves information about the specified host. It provides the same service as the `gethostbyaddr(3)` network function.

`_GETHOSTBYADDR(hostaddr, type, hostentry)`

hostaddr is the binary host address string.

type is either `AF_INET` or `AF_UNIX`.

hostentry is a stem that receives the data from the network host database. When the function returns, the following compound variables are set

hostentry.H_NAME

the official name of the host

hostentry.H_ALIASES

a list of aliases for this host name

hostentry.H_ADDRTYPE

the type of address being returned

hostentry.H_LENGTH

the length, in bytes, of the address

hostentry.H_ADDR

the address of the host

_GETHOSTBYNAME

The `_GETHOSTBYNAME` function retrieves information about the specified host. It provides the same service as the `gethostbyname(3)` network function.

`_GETHOSTBYNAME(name, hostentry)`

name is the hostname for which the host entry information is sought. *name* may be a literal string or an expression that evaluates to a valid hostname on the network.

hostentry is a stem that receives the data from the network host database. When the function returns, the following compound variables are set

hostentry.H_NAME

the official name of the host

hostentry.H_ALIASES

a list of aliases for this host name

hostentry.H_ADDRTYPE

the type of address being returned; this is always "AF_INET"

hostentry.H_LENGTH

the length, in bytes, of the address

hostentry.H_ADDR

the address of the host

The `_GETHOSTBYNAME` function may be used to obtain address information required by the `_CONNECT` function.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of _gethostbyname to retrieve
 * data in preparation for invoking a _connect
 * function
 */
call _gethostbyname('pluto', 'chost.')
net.sa_family = chost.h_addrtype
net.sin_addr = chost.h_addr
net.sin_port = 11111
s = _socket('AF_INET', 'SOCK_STREAM')
crc = _connect(s, 'net.')
```

```

/*
 * the following program fragment modifies the
 * previous example to run on the current host;
 * it uses the _gethostname function to
 * retrieve the name of the current host
 */
call _gethostbyname(_gethostname(), 'chost.')
net.sa_family = chost.h_addrtype
net.sin_addr = chost.h_addr
net.sin_port = 11111
s = _socket('AF_INET', 'SOCK_STREAM')
crc = _connect(s, 'net.')

```

_GETHOSTID The `_GETHOSTID` function retrieves a unique identifier for the current host. It provides the same service as the `gethostid(2)` system call.

_GETHOSTID()

The function returns a 32-bit identifier for the current host in a binary string. This identifier should be unique for each host.

Some UNIX implementations do not support `gethostid(2)`. On these systems, the `_GETHOSTID` function is unavailable. Check your operating system man pages to determine if it is supported on your system.

Examples:

```

say d2x(_gethostid())
/*
 * the output is, possibly, "57123379"
 */

```

_GETHOSTNAME

The `_GETHOSTNAME` function retrieves the name of the current host. It provides the same service as the `gethostname(2)` system call.

_GETHOSTNAME()

Examples:

```
say _gethostname()
/*
 * if this program is executed on a host named
 * 'pluto', the output is "pluto"
 */

/*
 * the following program fragment illustrates
 * the use of _gethostbyname to retrieve
 * data in preparation for invoking a _connect
 * function; since this program will be run on
 * the current host, _gethostname is used to
 * provide the host name in the _gethostbyname
 * call
 */
call _gethostbyname(_gethostname(), 'chost.')
net.sa_family = chost.h_addrtype
net.sin_addr = chost.h_addr
net.sin_port = 11111
s = _socket('AF_INET', 'SOCK_STREAM')
crc = _connect(s, 'net.')
```

_GETPEERNAME

The `_GETPEERNAME` function returns the name of the peer connected to a specified socket. It provides the same services as the `getpeername(2)` system call.

_GETPEERNAME(*sockhandle*, *sockaddr*)

sockhandle is the handle of a socket that has been created with the `_SOCKET` function.

sockaddr is a stem that receives address information about the current connection when `_GETPEERNAME` is executed. (This is the same information returned in the *sockaddr* stem on a call to `_ACCEPT`.) When the function returns, one or more of the following compound variables is set:

sockaddr.SA_FAMILY

the type of socket; it is one of the following:

“AF_INET” if the connection is from an Internet domain socket

“AF_UNIX” if the connection is from a UNIX domain socket

sockaddr.SIN_ADDR

the binary Internet address of the socket; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SIN_PORT

the numeric Internet port; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SUN_PATH

the path name of the socket; set only if *sockaddr.SA_FAMILY* is “AF_UNIX”; on some UNIX implementations, this value may not be set

_GETPID

The `_GETPID` function returns the process id for the current process. It provides the same service as the `getpid(2)` system call.

_GETPID()**Examples:**

```
say _getpid()
/* the output is, possibly, '11914' */

/*
 * the following program fragment retrieves the
 * process id to create a unique filename for
 * a temporary file used by the program
 */
temp_file = '/tmp/mprog.'_getpid()
do i = 1 to list.0
  call lineout temp_file, list.i
end
call lineout temp_file
```

_GETPPID

The `_GETPPID` function returns the process id of the current process's parent. It provides the same service as the `getppid(2)` system call.

_GETPPID()

Examples:

```
say _getppid()
/* the output is, possibly, '10853' */

/*
 * the following program fragment retrieves the
 * parent process id to create a unique file-
 * name for a temporary file used by the
 * program
 */
temp_file = '/tmp/mprog.'_getppid()
do i = 1 to list.0
  call lineout temp_file, list.i
end
call lineout temp_file
```

_GETSERVBYNAME

The `_GETSERVBYNAME` function returns information about the specified network service. It provides the same service as the `getservbyname(3)` network function.

_GETSERVBYNAME(name, proto, serventry)

name is the network service for which data is to be retrieved. ***name*** must be a valid service name found in the network services database, `/etc/services`. ***name*** may be a literal string or an expression that evaluates to a valid service name.

proto is the name of the protocol for this network service.

serventry is a stem that receives the data from the network services database. When the function returns, the following compound variables are set:

serventry.S_NAME

the official name of the service

serventry.S_ALIASES

a list of aliases for this service name

serventry.S_PORT

the port number at which the service resides

serventry.S_PROTO

the protocol to use when contacting the service

Examples:

```
/*
 * the following program fragment illustrates
 * the use of _getservbyname to get the port
 * number for a service to which you wish to
 * connect
 */
call _getservbyname('myserv', 'tcp', 'cserv.')
net.sa_family = chost.h_addrtype
net.sin_addr = chost.h_addr
net.sin_port = cserv.s_port
socket = _socket('AF_INET', 'SOCK_STREAM')
connect_rc = _connect(socket, 'net.')
```

_GETSOCKNAME

The **_GETSOCKNAME** function returns the name of the specified socket. It provides the same service as the **getsockname(2)** system call.

_GETSOCKNAME(sockhandle, sockaddr)

sockhandle is the handle of a socket that has been created with the **_SOCKET** function.

sockaddr is a stem that receives address information about the current connection when **_GETSOCKNAME** is executed. (This is the same information returned in the ***sockaddr*** stem on a call to **_ACCEPT**.) When the

function returns, one or more of the following compound variables is set:

sockaddr.SA_FAMILY

the type of socket; it is one of the following:

“AF_INET” if the connection is from an Internet domain socket

“AF_UNIX” if the connection is from a UNIX domain socket

sockaddr.SIN_ADDR

the binary Internet address of the socket; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SIN_PORT

the numeric Internet port; set only if *sockaddr.SA_FAMILY* is “AF_INET”

sockaddr.SUN_PATH

the path name of the socket; set only if *sockaddr.SA_FAMILY* is “AF_UNIX”; on some UNIX implementations, this value may not be set

_GETSOCKOPT

The `_GETSOCKOPT` function is used to determine the current options set for a specified socket. It provides the same service as the `getsockopt(2)` system call.

`_GETSOCKOPT(sockhandle, level, option, opt_value)`

sockhandle is the handle of a socket that has been created with the `_SOCKET` function.

level is the level of the option to be queried. Currently, only `SOL_SOCKET` is supported.

option is the name of the socket option for which the current setting is to be returned. It may be any of the following:

SO_ACCEPTCONN	SO_RCVLOWAT
SO_BROADCAST	SO_RCVTIMEO
SO_DEBUG	SO_REUSEADDR
SO_DONTROUTE	SO_SNDBUF
SO_ERROR	SO_SNDLOWAT
SO_KEEPALIVE	SO_SNDTIMEO
SO_LINGER	SO_TYPE
SO_OOINLINE	SO_USELOOPBACK
SO_RCVBUF	

opt_value is the name of a variable that receives the current option setting. If *option* is specified as “SO_LINGER”, then *opt_value* must be a stem and the element returned is either *opt_value.L_ONOFF* or *opt_value.L_LINGER*. For all other options, the value returned is 1 if the option is enabled or 0 if the option is disabled.

_GETUID

The `_GETUID` function returns the real userid of the current process. It provides the same service as the `getuid(2)` system call.

_GETUID()

The real userid is the userid under which the user who started the process logged in, regardless of any “su” command(s) that may have been executed. Use the `_GETEUID` function to retrieve the effective userid.

The function returns the user’s numerical id. On some systems, this is the value of the third field in the `/etc/passwd` file.

Examples:

```
say _getuid()  
/*  
 * for a user logged in as "user1" whose  
 * numeric id is "1010"  
 */
```

_IOCTL

The `_IOCTL` function performs one of a set of special functions on the specified file descriptor. It provides the same service as the `ioctl(2)` system call.

_IOCTL(file_handle, request, argument)

file_handle is the handle or file descriptor on which the operation is to be performed

request is the special function to be performed. It may be any one of the following:

FIOCLEX

FIONBIO

FIONCLEX

argument is a numeric argument for the request.

_KILL

The `_KILL` function terminates a process. It provides the same service as the `kill(2)` system call.

_KILL(pid, signal)

pid is the process id of the process to be terminated. *pid* must be a positive whole number and must be the process id of a currently executing process.

signal is the signal to be sent. *signal* must be a positive whole number that is a valid signal on the system where it is executed.

Unless the real or effective userid of the process executing `_KILL` is superuser, the real or effective userid of the process executing `_KILL` must be the same as the real or effective userid of the process to be terminated. The `_GETUID` and `_GETEUID` functions, respectively, provide access to the real and effective userids.

Examples:

```
/*
 * the following program, named "clobber",
 * kills all processes in the argument list
 */
parse arg who_to_kill
do while who_to_kill \= ''
  parse var who_to_kill next who_to_kill
  if \datatype(next, 'W') then call error1 next
  kill_rc = _kill(next, 9)
  if kill_rc < 0 then call error2 next
end
exit
error1:
parse arg proc_id
say 'Invalid process id:' proc_id
return
error2:
parse arg proc_id
say 'Error killing process:' proc_id
say 'System error:' _errno()
say _sys_errlist(_errno())
say ''
say 'Process not killed'
return
```


_LISTEN

The `_LISTEN` function listens for connections on a socket. It provides the same service as the `listen(2)` system call.

_LISTEN(*sockhandle*, [*limit*])

sockhandle is the handle of a socket that has been created with the `_SOCKET` function and has been bound to an address with the `_BIND` function. The `_LISTEN` function is used only with sockets of type `SOCK_STREAM`.

limit specifies the maximum number of pending connections that this socket will accept. *limit* must be a positive whole number. If a connection request would cause the number of pending connections to exceed the value specified by *limit*, the client program attempting the connection receives an error return from the `_CONNECT` function. If *limit* is omitted, the default value is 5.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of the _listen function; for simpli-
 * city of illustration, it does not include
 * the error checking that would normally be
 * present in a robust program
 */
domain = 'AF_INET'
server = 'myservice'
call _getservbyname(server, 'sdata.')
a = _socket(domain, 'SOCK_STREAM')
net.sa_family = domain
net.sin_port = sdata.s_port
brc = _bind(a, 'net.')
lrc = _listen(a)
```

`_OPENDIR`

The `_OPENDIR` function opens a directory. It provides the same service as the `opendir(3)` library function.

`_OPENDIR(directory_name)`

directory_name is the name of the directory to be opened. It may be a single directory name if the directory is a sub-directory of the current working directory; or it may be a full directory path name. It may also be specified as “.” to refer to the current working directory. Since a shell is not invoked to process this function, *directory_name* may not include references to environment variables (such as \$HOME) or shell-specific shorthand (such as “~”) as these notations are expanded only by a shell. If *directory_name* does not exist, is not a directory, or does not have read permission for the real or effective userid of the process executing the `_OPENDIR`, the function returns 0.

The function is typically invoked as

```
var = _opendir(directory_name)
```

var contains the file descriptor associated with the opened directory.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of _opendir, _readdir, and _closedir
 * to obtain directory information; it prints
 * the first 5 directory entries
 */
dir = _opendir('/home/user1')
if dir \= 0 then do 5
    line = _readdir(dir)
    say line
end
call _closedir dir
```

_READDIR

The `_READDIR` function reads the next directory entry. It provides the same service as the `readdir(3)` library function.

_READDIR(*dirhandle*)

dirhandle is the handle returned by a previous call to `_OPENDIR`.

If a read error occurs or if there are no more directory entries, `_OPENDIR` returns a null string.

Examples:

```
/*
 * the following program fragment illustrates
 * the use of _opendir, _readdir, and _closedir
 * to obtain directory information; it prints
 * the first 5 directory entries
 */
dir = _opendir('/home/user1')
if dir \= 0 then do 5
  line = _readdir(dir)
  say line
end
call _closedir dir
```

_RECV

The `_RECV` function receives a message from a socket. It provides the same services as the `recv(2)` system call.

_RECV(*sockhandle*, *buffer*, *length* [, *flagnames*])

sockhandle is the handle of a socket that has been created as a result of the `_SOCKET` or the `_ACCEPT` function.

buffer is the name of a variable that will contain the message when the function is executed.

length specifies the amount of data to be read. *length* must be a positive whole number.

flagnames specify alternative behavior of the `_RECV` function. *flagnames* may be a literal string or an expression which evaluates to one of the following:

MSG_OOB

read “out-of-band” data rather than the normal “in-band” data

MSG_PEEK

preview the data; the data is read but is not removed from the socket; a subsequent call to `_RECV` without the “MSG_PEEK” flag reads the same message again

If *flagnames* is omitted, `_RECV` reads *length* data from the socket.

If the socket is blocking, then if no data is available on the socket, `_RECV` waits for a message to arrive. If the socket is non-blocking, then if no data is available, `_RECV` returns a value less than 0.

Examples:

```
/*
 * the following program fragment illustrates
 * two ways to use _recv; the first call simply
 * previews the first 4 bytes of data, which
 * contain the length of the message; the
 * second call actually reads all the data
 */
recvbuf: procedure
parse arg socket
/*
 * get the first 4 bytes of data (which tell
 * us how long the data is) into the variable
 * "len"; do not remove data from the socket
 */
recvrc = _recv(socket, 'len', 4, 'MSG_PEEK')
/*
 * get "len" bytes of data from the socket and
 * store it in the variable 'buf'; this call
 * actually removes "len" bytes from the
 * socket
 */
recvrc = _recv(socket, 'buf', len)
```

_REGEX

The `_REGEX` function compares a regular expression pattern to a string.

`_REGEX(pattern, string)`

pattern is the regular expression to be matched. *string* is the string to be compared to the regular expression.

`_REGEX` returns 1 if the pattern matches the string and 0 if there is no match.

Examples:

```
/* the following program compares a file
   pattern to all directory entries to find
   all file names that match the pattern; it
   searches for all files of the form
   "???.rex" - 3-character filename with the
   ".rex" extension */
file = '???.rex'
file2reg = file2reg(file) /* convert to reg exp */
dir = _opendir('.') /* open directory */
do while entry \= '' /* for each dir entry */
  entry = _readdir(dir)
  if _regex(file2reg, entry) then
    say entry
  end
end
call _closedir(dir) /* close directory */
exit
/* convert filename to regular expression */
file2reg: procedure
  parse arg pattern
  if pattern \= '' then do
    reg = '^'
    do index = 1 to length(pattern)
      char = substr(pattern, index, 1)
      select
        when char == '.' then reg = reg || '\.'
        when char == '?' then reg = reg || '\?'
        when char == '*' then reg = reg || '\*'
        when char == '[' & lchar == '[' then
          reg = reg || '[' & lchar & ']'
        otherwise reg = reg || char
      end
      lchar = char
    end index
  else
    reg = '^.*$'
  end
  return reg
end
```

`_SELECT`

The `_SELECT` function determines if an I/O stream is ready for reading or writing or is in an exception state. It provides the same service as the `select(2)` system call.

`_SELECT`(*width*, *r_mask*, *w_mask*, *e_mask*, *timeout*)

width is the maximum number of bits in the I/O descriptor selection masks.

r_mask is the name of a variable containing the read selection mask. This mask must have been previously created using the `_FD_ZERO` function and modified with `_FD_CLEAR` or `_FD_SET`.

w_mask is the name of a variable containing the write selection masks. This masks must have been previously created using the `_FD_ZERO` function and modified with `_FD_CLEAR` or `_FD_SET`.

e_mask is the name of a variable containing the exception selection mask. This mask must have been previously created using the `_FD_ZERO` function and modified with `_FD_CLEAR` or `_FD_SET`.

timeout is the name of a stem that specifies the timeout value to wait for this `_SELECT` to complete. You may specify the timeout value using one of the following elements:

`timeout.TV_SEC`
timeout value in seconds

`timeout.TV_USEC`
timeout value in milliseconds; this may not be reliable on some platforms

_SEND

The `_SEND` function sends a message to a socket. It provides the same service as the `send(2)` system call.

`_SEND(sockhandle, buffer [, [length] [, flagnames]])`

sockhandle is the handle of a socket that has been created as a result of the `_ACCEPT` function.

buffer is the name of a variable that contains the message to be sent.

length specifies the amount of data to send. *length* must be a positive whole number that is less than or equal to the length of *buffer*. If *length* is omitted, the default is the length of *buffer*.

flagnames specify alternative behavior of the `_SEND` function. *flagnames* may be a literal string or an expression which evaluates to one of the following:

MSG_OOB

send “out-of-band” data rather than the normal “in-band” data; only Internet domain sockets of type `SOCK_STREAM` support “out-of-band” data

MSG_DONTROUTE

normally used only for diagnostic purposes

If no space is available on the socket for the message, `_SEND` normally blocks unless the socket has specifically been placed in non-blocking mode. If the message cannot be sent, a system error is set.

Examples:

```
/*
 * the following program fragment illustrates
 * use of the _SEND function
 */
parse arg socket
msg = 'Hello world'
call _send socket, msg, length(msg)

/*
 * the following program fragment sends a
 * length-prefixed message so that the _RECV
 * function can look at the first 4 bytes of
 * the message to determine how much data to
 * read
 */
parse arg socket
msg = 'Hello world'
/*
 * pad length of message on right with zeros
 * to be exactly 4 characters long
 */
length = right(length(msg), 4, 0)
packet = length||msg
call _send socket, msg
```

_SETSID

The `_SETSID` function makes the current process the group leader of a new process group. It provides the same service as the `setsid(2)` system call.

_SETSID()

The new process group has no controlling terminal. This is useful when starting a daemon to avoid having the daemon affected by the job control and other process relationships of the shell that started it.

Some UNIX implementations do not support `setsid(2)`. On these systems, the `_SETSID` function is unavailable. Check your operating system man pages to determine if it is supported on your system.

_SETSOCKOPT The `_SETSOCKOPT` function is used to set options for a specified socket. It provides the same service as the `setsockopt(2)` system call.

`_SETSOCKOPT(sockhandle, level, option, opt_value)`

sockhandle is the handle of a socket that has been created with the `_SOCKET` function.

level is the level of the option to be queried. Currently, only `SOL_SOCKET` is supported.

option is the name of the socket option for which the current setting is to be returned. It may be any of the following:

<code>SO_ACCEPTCONN</code>	<code>SO_RCVLOWAT</code>
<code>SO_BROADCAST</code>	<code>SO_RCVTIMEO</code>
<code>SO_DEBUG</code>	<code>SO_REUSEADDR</code>
<code>SO_DONTROUTE</code>	<code>SO_SNDBUF</code>
<code>SO_ERROR</code>	<code>SO_SNDLOWAT</code>
<code>SO_KEEPALIVE</code>	<code>SO_SNDTIMEO</code>
<code>SO_LINGER</code>	<code>SO_TYPE</code>
<code>SO_OOINLINE</code>	<code>SO_USELOOPBACK</code>
<code>SO_RCVBUF</code>	

opt_value is the name of a variable that receives the current option setting. If *option* is specified as “`SO_LINGER`”, then *opt_value* must be a stem and the two elements to be set are *opt_value.L_ONOFF* and *opt_value.L_LINGER*. For all other options, the value

set should be 1 to enable the option or 0 to disable the option.

`_SLEEP`

The `_SLEEP` function suspends execution of a program for a specified interval. It provides the same service as the `sleep(3)` library function.

`_SLEEP(time)`

time is the number of seconds that the program remains suspended. *time* must be a positive whole number.

Examples:

```
call _sleep 5
/*
 * causes the program to pause for 5 seconds
 * before the next instruction is executed
 */
```

`_SOCKET`

The `_SOCKET` function creates a socket – a point for communication between processes. It provides the same service as the `socket(2)` system call.

`_SOCKET(family, type, [protocol])`

family specifies communications domain for this socket. *family* may be a literal string or an expression that evaluates to a literal string. The only permitted values are

- “AF_INET” to create an Internet domain socket
- “AF_UNIX” to create a UNIX domain socket

type specifies the type of socket. *type* may be a literal string or an expression that evaluates to a literal string. The only permitted values are:

SOCK_STREAM
SOCK_DGRAM

protocol specifies the communication protocol to be used. Since there is normally only one protocol defined

for each socket type, *protocol* is normally specified as 0. If *protocol* is omitted, the default value is 0.

The function is typically invoked as

```
var = _socket(family, type)
```

var contains the file descriptor associated with the newly created socket.

Examples:

```
/*
 * the following code fragment illustrates
 * creation of a socket
 */
mysocket = _socket('AF_INET', 'SOCK_STREAM')
if mysocket < 0 then
say 'Error:' errno() '-' _sys_errlist(_errno())
```

_STAT

The **_STAT** function retrieves status information about a file. It provides the same service as the **stat(2)** system call.

_STAT(*file_name*, *stataddr*)

file_name is the name of the file for which status information is requested. It may be a simple file name or a full path to the file. The **_STAT** function can access the file regardless of its permissions; however, all directories in the path to the file must exist and must have read and execute permission for the process in which the uni-REXX program is running.

stataddr is a stem that receives information about the file when **_STAT** is executed. When the function returns, the following compound variables are set:

<i>stataddr</i>.ST_DEV	device number
<i>stataddr</i>.ST_INO	inode number
<i>stataddr</i>.ST_MODE	numeric file mode

<i>stataddr.ST_NLINK</i>	number of lines
<i>stataddr.ST_UID</i>	numeric userid of the owner
<i>stataddr.ST_GID</i>	numeric group id of the owner
<i>stataddr.ST_RDEV</i>	device number
<i>stataddr.ST_SIZE</i>	file size in bytes
<i>stataddr.ST_ATIME</i>	last access time, in seconds since 1 Jan 1970
<i>stataddr.ST_MTIME</i>	last modification time, in seconds since 1 Jan 1970
<i>stataddr.ST_CTIME</i>	creation time, in seconds since 1 Jan 1970

Examples:

```

/*
 * the following program fragment uses _stat
 * to determine the last access time of all
 * files in a directory and removes those older
 * than the specified expiration date; the
 * get_expir_time function, not shown, would
 * calculate the expiration date in seconds
 * since 1/1/70
 */
dir = '/home/user1'
call popen('ls -l' dir)
exp_date = get_expir_time()
do queued()
  pull file
  rc = _stat(file, 'st.')
  if st._atime < exp_date then 'rm' file
end

```

_SYS_ERRLIST The `_SYS_ERRLIST` function provides access to the text of system error messages.

_SYS_ERRLIST(*error_number*)

error_number is the system error number for which the message text is to be retrieved. *error_number* must be a positive whole number.

The `_ERRNO` function provides access to the system error message number set when any function call returns an error.

Examples:

```
/*
 * the following program fragment illustrates
 * a subroutine that might be used to process
 * system errors
 */
error1:
parse arg caller
say 'A system error was detected in:' caller
say 'System error number:' _errno()
say _sys_errlist(_errno())
return
```

_SYSTEMDIR The `_SYSTEMDIR` function returns the “system directory”. For UNIX, this is always “/”.

_SYSTEMDIR()

_TRUNCATE

The `_TRUNCATE` function sets a file to a specified length. It provides the same service as the `truncate(2)` system call.

`_TRUNCATE(file, size)`

file is name of the file to be truncated. *file* may be a simple file name, if the file is in the current directory, or a full path name.

size is the size to which the file is to be set. *size* must be a non-negative whole number. If *size* is less than the current size of *file*, file is truncated to the new size and data is lost. If *size* is greater than the current size of the file, additional null characters are added to the file to create the desired size.

Examples:

```
/*
 * the following program is named "truncit";
 * the file to be truncated contains the single
 * line "Good morning to you" ; the output
 * from "ls -l" on this file shows a size of
 * 20
 */
call _truncate 'myfile', 12
/*
 * after the program is run, the file contains
 * the single line "Good morning" ; the
 * output from "ls -l" shows a size of 12
 */
```

_UMASK

The `_UMASK` function sets the file creation mode mask. This affects the permissions that are assigned to newly created files. The function provides the same service as the `umask(2)` system call.

`_UMASK([mask])`

mask is the mask to be used to alter the permissions for new files created by this program. *mask* must be a

valid mask number. The man page for the `umask` command contains details on the setting and use of masks. If *mask* is omitted, the function returns the current UMASK setting.

Examples:

```
call _umask(020)
call lineout 'newfile', 'hello'
call lineout 'newfile'
/*
 * on a system where new files are normally
 * created with -rw-rw-r- permission, the
 * program causes new files to be created with
 * -rw-r-rw- permission
 */
```

_WAIT

The `_WAIT` function waits for a process to terminate. It provides the same service as the `wait(2)` system call.

_WAIT([status])

status is the name of a variable to receive the status code returned by the `_WAIT` function. If *status* is omitted, the code is discarded.

`_WAIT` returns the process id of the process that terminated.

Examples:

```
/* the following program fragment illustrates
 * the use of wait to suspend the parent
 * process until the child process terminates
 */
pid = _fork()          /* create child process */
/*
 * if this is the child process, sleep for 1
 * second and then exit; typically, the child
 * process would do some useful work and then
 * exit
 */
if pid = 0 then do
  call _sleep 1
end
```



```

/*
 * if this is the parent, wait for the child to
 * complete
 */
wid = _wait()
/*
 * when we get a termination signal, be sure it
 * is for the child
 */
do while wid \= pid
  say wid d2x(wid)
  wid = _wait()
end

```

_WAITPID

The `_WAITPID` function waits for a specified process to terminate. It provides the same service as the `waitpid(2)` system call. The `_WAITPID` function may not be available on all UNIX implementations.

`_WAITPID(pid [, status] [, options])`

pid is the process id of the process that must terminate before execution of the program can continue. *pid* must be the process id of a process that is a current child of the one executing the `_WAITPID` function.

status is the name of a variable to receive the status code from the `_WAITPID` function. If *status* is omitted, the code is discarded.

option specifies the action to be taken by `_WAITPID`. It must be one of the following:

WNOHANG	the process does not actually wait but only returns a status code if it is already complete
WUNTRACED	the function returns the status of any child processes that are stopped

If *option* is omitted, the function waits normally.

Some UNIX implementations do not support `waitpid(2)`. On these systems, the `_WAITPID` function is unavailable. Check your operating system man pages to determine if it is supported on your system.

Examples:

```
/*
 * the following program fragment creates a
 * child process & waits for it to complete
 */
pid = _fork()
if pid = 0 then do
    :                               /* do some useful work */
    end
wid = _waitpid()
```

Client/Server Sample Appli- cation

This sample application illustrates the use of some of the UNIX-specific functions in a client/server application to provide information about available modems on the system to any user who requests it. The modem server (`mods`) runs on a host named ``zeus'`. At startup, it reads a modem configuration file to determine what modems have been installed on various network nodes and some configuration information about each modem. A client program (`modc`), which can be run on any workstation in the network, sends modem requests to the server and gets back the hostname and modem initialization string for the first free modem that satisfies the request. For simplicity of illustration, this example does not attempt to connect the user to a modem – it merely reports back the information required for the user to do this manually.

The programs that comprise this sample application are included in the uni-REXX Sample Library, which is on the product distribution media.

The Modem Configuration File

The modem configuration file contains a list of all modems installed on the network. Each line of the file is in the following format:

modem_speed hostname device_name init_str

where

<i>modem_speed</i>	the maximum baud rate of this modem
<i>hostname</i>	the hostname of the workstation to which the modem is attached
<i>device_name</i>	the UNIX device name by which the modem is addressed
<i>init_str</i>	the modem initialization string

Sample lines from the file look like

```
19200 athena /dev/cua0 +++ato
2400 artemis /dev/ttyd0 +++ato
9600 poseidon /dev/cua0 None
14400 apollo /dev/cua0 \e129++
```

The client program sends requests to the server in the one of the following forms:

GET *speed*
FREE *hostname device_name*

The client program is executed by typing

modc *request*

where *request* is one of the message forms shown above.

Sending and Receiving Messages

Since both the client and the server need to send and receive messages, these processes are written as separate functions that each can call. The send function creates a length-prefixed packet of data. The receive function expects the data in this format and checks the length of the message before actually reading it from the socket.

The message send function (sendbuf) and the message receive function (recvbuf) are shown on the following pages.

```

/*
 * sendbuf - send a length-prefixed packet
 *
 * This routine builds a message packet in the form
 *
 *      _____
 *      | length | message |
 *      _____
 *
 * where "length" is always exactly 4 bytes.
 *
 * The recipient (recvbuf) can then read the first 4 bytes to
 * determine the length of the message waiting on the socket
 * stream.
 *
 */
sendbuf: procedure
parse arg socket, buffer

/*
 * Get buffer length, make it 4 characters long, padded with
 * zeros
 */
bufferlength = right(length(buffer), 4, '0')

/*
 * Get length of whole packet (buffer plus length field), make
 * it 4 characters, padded with zeros
 */
bufferlength = right(bufferlength+length(bufferlength), 4, '0')

/*
 * Concatenate length and buffer into a message packet
 */
packet = bufferlength||buffer

/*
 * Send the message packet
 */
call _send socket, packet, , ""
if sendrc < 0 then call error 'send'
return

```

```

/*
 * recvbuf - recv a length-prefixed packet
 *
 * This routine receives a message packet in the form
 *
 *      _____
 *      | length | message |
 *      _____
 *
 * where "length" is always exactly 4 bytes.
 *
 * It uses the "MSG_PEEK" flag of _recv to obtain the length
 * field, placing it in the specified variable. "MSG_PEEK"
 * allows you to retrieve information from the message without
 * removing it from the stream. The routine then invokes
 * _recv again, using the length variable to specify how much
 * data to read from the socket stream
 *
 */
recvbuf: procedure
parse arg socket

/*
 * Peek at the message to get its length - put that value in the
 * variable bufferlength
 */
recvrc = _recv(socket, "bufferlength", 4, "MSG_PEEK")
if recvrc < 0 then call error "recv"

/*
 * Actually receive the full message packet, including its
 * length prefix
 */
recvrc = _recv(socket, "buffer", bufferlength, "")
if recvrc < 0 then call error "recv"

/*
 * Remove the length prefix from the message and return only the
 * message
 */
message = substr(buffer,5)
return message

```

The Server

The modem server (mods) can be run on any host in the network. In this example, it is assumed that the server is running on a host named `zeus`. The client program will send its requests to `zeus` for processing.

The modem server program is shown below.

```
#!/usr/local/bin/rxx
/*
 * mods - modem allocation server
 *
 * Program runs as a daemon on the designated host.  Accepts
 * requests from clients across the network for allocation of
 * modems.  Uses modem configuration file to determine which
 * modem satisfies the particular specifications requested.
 *
 * At this time, the modem server does not actually connect the
 * requestor to the modem - it simply returns the device name
 * and hostname of an available modem that satisfies the cur-
 * rent request.
 *
 */

/*
 * Fork process to continue running as a daemon & exit parent
 * process
 */

forkrc = _fork()
if forkrc < 0 then call error 'fork'
if forkrc \= 0 then do
    say 'Modem server daemon started:  PID =' forkrc
    exit
end

/*
 * Process modem configuration file to get all modems into the
 * modem. stem
 */
modem_list = './modem.list'
modem. = ''
address command 'execio * diskr' modem_list '(stem modem.'

/*
 * Open a socket
 */
sock1 = _socket('AF_INET', 'SOCK_STREAM')
if sock1 < 0 then call error 'socket'
```

```

/*
 * Socket address structure
 */
net.sa_family = 'AF_INET'
net.sin_port = 11111

/*
 * Bind socket to port
 */
bindrc = _bind(sock1, 'net.')
if bindrc < 0 then call error 'bind'

/*
 * Listen for connections from clients
 */
listenrc = _listen(sock1)
if listenrc < 0 then call error 'listen'

/*
 * Main communications/processing loop
 */

do forever

/* Accept client connections */

    sock2 = _accept(sock1)
    if sock2 < 0 then call error 'accept'

/* Use the recvbuf runction to receive length-prefixed message
 * packet */

    request = recvbuf(sock2)
    request = lower(request) /* this app needs msg in lowercase */
    parse var request get_free rest
    select
        when get_free = 'get' then do /*For a "get" request, see*/
            parse var rest req_baud rest /*what kind of modem needed*/
            got_one = 0 /* & go through all not busy */
            do i = 1 to modem.0 /* until you find one */
                parse var modem.i max_baud hostname device init_str busy
                if req_baud <= max_baud then
                    if busy = '' then do
                        reply = hostname device init_str
                        modem.i = modem.i 'busy'
                        got_one = 1
                    end
                end
            end
            if got_one then leave
        end
    if \got_one then
        reply = 'No' req_baud 'modem available at this time'

```



```

    end
when get_free = 'free' then do /*For a "free" request, see*/
    parse var rest free_host free_dev rest /*which host/dev*/
    freed = 0 /* & go through until found. */
    do i = 1 to modem.0 /* If marked busy, free it. */
        parse var modem.i max_baud hostname device init_str busy
        if hostname = free_host then
            if device = free_dev then
                if busy = 'busy' then do
                    modem.i = max_baud hostname device init_str
                    reply = 'Modem free request processed'
                    freed = 1
                end
            if freed then leave
            end
        if \freed then
            reply = free_host free_dev 'not allocated - not freed'
        end
    otherwise /*Reqs not starting with get or free are invalid */
        reply = get_free 'invalid; must begin with "get" or "free"'
    end

/*
* Call sendbuf to create a length-prefixed message packet and
* send the reply back to the client
*/
call sendbuf sock2, reply

/*
* Close client connection
*/
closerc = _closesocket(sock2)
if closerc < 0 then call error 'close'
end /* end of do forever loop */

/*
* Close accepting connection
*/
closerc = _closesocket(sock1)
if closerc < 0 then call error 'close'
exit

error:
/*
* Print function that failed, error number, and system error
* text. Then call system exit function to really exit this
* process
*/
say arg(1) 'error' _errno()':' _sys_errlist(_errno())

    call _exit(1)

```

The Client

The client program (modc) can be run on any host in the network. In this example, it is assumed that the server is running on a host named `zeus`. The client program will send its requests to `zeus` for processing.

The client program is shown below.

```
#!/usr/local/bin/rxx
/*
 * modc - modem request client
 *
 * Program requests modem allocation from modserver (modem
 * allocation server) running on the network.
 */

arg request

/*
 * Get host configuration
 */
call _gethostbyname('zeus', 'chost.')

/*
 * Socket address structure
 */
net.sa_family = chost.h_addrtype
net.sin_addr = chost.h_addr
net.sin_port = 11111
net.sin_addr = 'INADDR_ANY'

/*
 * Create the socket
 */
socket = _socket('AF_INET', 'SOCK_STREAM')
if socket < 0 then call error 'socket'

/*
 * Connect to the server
 */
connectrc = _connect(socket, 'net.')
if connectrc < 0 then call error 'connect'

/*
 * Send the request to the server
 */
call sendbuf socket, request
```

```

/*
 * Get reply back from server
 */
reply = recvbuf(socket)

say 'Reply from modem allocation server'
say ''
say reply

/*
 * Close the connection
 */
closerc = _closesocket(socket)
if closerc < 0 then call error 'close'

exit

error:
/*
 * Print function that failed, error number, and system error
 * text. Then call system exit function to really exit this
 * process.
 */
say arg(1) 'error' _errno()':' _sys_errlist(_errno())
call _exit(1)

```

Operating System Facilities

For convenience in porting applications among a variety of platforms, uni-REXX includes certain facilities that are provided as part of the operating system in other environments where the REXX language is implemented. These include

EXECIO

alternative method of performing file I/O

GLOBALV

management of global variables

MAKEBUF, DROPBUF, DESBUF

management of buffers within the program stack

RXQUEUE

pipe command or program output to the program stack

SENTRIES

return the number of entries in the program stack

While these facilities make it easier to move applications between mainframe environments and uni-REXX, they are not fully portable because they are not present on all platforms where REXX is implemented.

These facilities are implemented as external commands and not as part of the uni-REXX interpreter. They are delivered as separate modules on the uni-REXX distribution media. The uni-REXX Developer's Kit includes a redistribution license for these modules. If you have licensed the Developer's Kit and your program uses any of these external commands, you may distribute the necessary binaries along with your program.

Like the uni-REXX instructions, the syntax diagrams for these facilities show the command name in uppercase. The name may appear in a program in upper- or lower-case but not in mixed case.

DESBUF

The DESBUF command clears from the program stack all buffers created by the MAKEBUF command.

DESBUF

Examples:

```
/*
 * the following program fragments illustrate
 * the effect of using DESBUF
 */
push zero
`makebuf`
push `one`
`makebuf`
push `two`
`makebuf`
push `three`
pull next
say next
pull next
say next
/*
 * the output is
 *     THREE
 *     TWO
 */
```

```
push zero
`makebuf`
push `one`
`makebuf`
push `two`
`makebuf`
push `three`
pull next
say next
`desbuf`
pull next
say next
/*
 * the output is
 *     THREE
 *     ZERO
 */
```

DROPBUF

The DROPBUF command clears from the program stack one or more specific buffers created with the MAKEBUF command.

DROPBUF [*n*]

n is the buffer number of the first buffer to drop. DROPBUF drops buffer *n* and all buffers created after it. If *n* is omitted, the default value is the buffer number of the last buffer created.

DROPBUF sets one of the following return codes if an error occurs:

- 1 *n* is not a valid number
- 2 the specified buffer does not exist

Examples:

```
/*
 * the following program fragment illustrates
 * dropping the buffer created most recently
 */
push `zero`
`makebuf`
push `one`
`makebuf`
push `two`
`dropbuf`
pull next
say next
/* the output is "ONE" */

/*
 * the following program fragment illustrates
 * dropping buffer 2; any buffers created after
 * buffer 2 are automatically dropped, also
 */
push `zero`
`makebuf`; push `one`
`makebuf`; push `two`
`makebuf`; push `three`
`dropbuf 2`
pull next
say next
/* the output is "ONE" */
```

EXECIO

The EXECIO command reads or writes lines from a disk file to the program stack or to one or more program variables.

EXECIO *n* **DISKR** *file* [*lnum*] ([FINIs] *options*)
DISKW

n is the number of lines to read or write. *n* must be a non-negative whole number. If *n* is 0, then no lines are processed. If *n* is 0 and the FINIs keyword is specified, then the only action taken is to close the file.

If *n* is specified as “*”, all available lines are processed. For a read operation (DISKR) all lines in the file are read. For a write operation (DISKW) all lines on the program stack or in the specified stem are written to the file. Specifying *n* as “*” in conjunction with the VAR option (discussed below) is not permitted since only one line is processed when VAR is used.

DISKR and **DISKW** specify the type of I/O operation to be performed. **DISKR** indicates that lines are to be read from a file. **DISKW** indicates that lines are to be written to a file. One of these keywords **must** be specified.

file is the name of the disk file for this I/O operation. *file* may be any valid UNIX file name. It will probably be necessary to modify the filename specification for applications being migrated to or from other environments. For ease of portability, consider using a variable to store the name of the file.

lnum is supported only for **DISKR**. It is the position in the file where the first read operation occurs. *lnum* must be a non-negative whole number. If *lnum* is specified as 0, read begins at the first line of the file. If *lnum* is omitted, the default value is 0.

The keyword **FINIs** controls the state in which the file is left after an I/O operation is performed. If **FINIs** is specified, the file is closed after the I/O occurs. If **FINIs** is omitted, the file may be left in an open state and it may be necessary to close it with **EXECIO** before subsequent I/O is performed.

options may be one or more of the following:

FInd */string/*

finds the first line that contains *string* beginning in the first character position and writes the following to the program stack:

the contents of the line

the line number where *string* was found; for **DISKR**, both the absolute and relative line numbers are reported

the range of characters searched may be limited with the *Zone* option described below

LOcate */string/*

finds the first line that contains *string* in any position and writes the following to the program stack

the contents of the line

the line number where *string* was found; for **DISKR**, both the absolute and relative line numbers are reported

the range of characters searched may be limited with the *Zone* option described below

Avoid */string/*

finds the first line that does **not** contain *string* in any position and writes the following to the program stack:

the contents of the line

the line number; for **DISKR**, both the absolute and relative line numbers are reported

the range of characters searched may be limited with the *Zone* option described below

Zone *c1 c2*

used to restrict the range of the input line that is searched by the FInd, LOcate, or Avoid options; if Zone is not specified, these operations search the entire line; *c1* and *c2* indicate the beginning and ending character positions, respectively, for the search; *c1* and *c2* must be positive whole numbers, except that *c2* may be specified as “*” to indicate the last column of the line

LIFO**FIFO**

these keywords specify the order in which lines are written to the program stack; only one may be specified on an EXECIO command; if this option is omitted, FIFO (first-in-first-out) is the default except when a search option (FInd, LOcate, or Avoid) is specified; in this case, LIFO (last-in-first-out) is the default because line numbers are placed on the stack also; this option is not valid with the VAR or STEM options since it affects only the program stack

SKip

valid only with DISKR; lines read with SKip in effect are not written to the program stack

Margins *c1 c2*

restricts the portion of a line from the stack or a variable that is to be processed; if Margins is not specified, the entire line is processed; *c1* and *c2* indicate the beginning and ending character positions, respectively, of the data to be processed; *c1* and *c2* must be positive whole numbers, except that *c2* may be specified as “*” to indicate the last column of the line

STRIP

specifies that trailing blanks are to be removed from all lines read or written

NOTYPE

this option has no effect in uni-REXX; it is provided for portability purposes only

STEM *stem*.

for DISKR, specifies that lines read from a file are to be placed in compound variables beginning with the stem *stem.*; the variable *stem.0* is automatically set to the number of lines read; the variables *stem.1* through *stem.n* contain the lines from the file

for DISKW, specifies that lines written to a file are to be taken from compound variables beginning with the stem *stem.*; the indices of *stem.* must be numeric and sequential; EXECIO automatically writes to the output file the contents of variables *stem.1* through *stem.n*, stopping at the first occurrence of a gap in the indices or a non-numeric index

VAR *var*

for DISKR, specifies that a line read from the file is to be placed in the variable *var*; for DISKW, specifies that the line to be written is taken from the variable *var*; if VAR is specified, *n* must be 1; in addition, if VAR is specified, the search options (FInd, Locate, Avoid) and the FIFO and LIFO keywords are not permitted

Because a number of the operands used with EXECIO may be specified as “*”, it is recommended EXECIO commands be addressed to the “command” host command environment. This precludes attempts by the shell to expand the special characters, which could have undesirable results. You may, if you prefer, specify the shell escape character (\) before the asterisk, but this is

not portable to other environments and may be confusing to someone reading the program if they are not familiar with UNIX shell processing.

EXECIO sets the following return codes:

0	successful completion
1	truncated
2	end of file encountered before the specified number of lines were read
3	number of lines to process was reached before a successful pattern match occurred
24	bad parameter list; the specified file does not exist or a keyword or option is misspelled
41	insufficient memory to run EXECIO
100	an I/O error occurred
2008	the name following STEM or VAR is an invalid stem or variable name

Examples:

```
/*
 * the following program fragment reads all
 * lines in a disk file and places them on
 * the program stack
 */
address command
`execio * diskr data.file`
```

```
/*
 * the following program fragment is similar
 * to the previous example but places the
 * lines read into the stem "lines." and closes
 * the file when the read is complete
 */
address command
`execio * diskr data.file (finis stem lines.`
```

```

/*
 * the following program fragment is similar to
 * the previous example but reads only 4 lines
 * beginning at line 10; in this example,
 * "lines.0" will have the value 4
 */
address command
'execio 4 diskr data.file 3 (finis stem lines.'

```

```

/*
 * the following program fragment writes to a
 * file all lines currently on the program
 * stack, stripping trailing blanks; it closes
 * the file when the write is complete
 */
address command
'execio * diskw out_file (finis strip'

```

```

/*
 * the following program fragment writes one
 * line to an output file; the data is taken
 * from the variable abc
 */
abc = 'Hello world'
address command 'execio 1 diskw ofile (var abc'

```

```

/*
 * the following program fragment illustrates
 * the use of execio in a portable program
 */
parse source sys .
select
  when sys = 'UNIX' then fn = 'june.data'
  when sys = 'CMS' then fn = 'JUNE DATA A'
  when sys = 'TSO' then fn = 'SALE.DATA(JUNE)'
  otherwise do
    say 'Please enter filename'
    parse pull fn
  end
end
address command
'execio * diskr' fn '(finis stem sales.'

```

```

/*
 * the following program fragment illustrates
 * using the LLocate search option to read
 * the desired line from a file
 */
string = 'June'
address command
'execio * diskr sales.data (lo '/'string''
parse pull .
parse pull . . june_commissions_paid .

/*
 * the following program fragment is similar to
 * the previous example but restricts the
 * search to columns 10-25 of each line and
 * limits the lines read to lines 8-12 of the
 * file
str = 'January'
fn = 'sales.data'
address command
'execio 4 diskr' fn '8 (lo '/'str'/ zone 10 25'

/*
 * the following program fragment reads every
 * line in the file *except* those for the
 * "East" region; data is stored in the stem
 * "noteast."
 */
str = 'East'
fn = 'sales.data'
address command
'execio * diskr' fn '(a '/'str'/ stem noteast.'
```

GLOBALV

The GLOBALV command provides a method for sharing variables among uni-REXX programs and for retaining variable values either temporarily or permanently for subsequent use.

GLOBALV allows you to store variables

- temporarily
- for the current session (session globalv)
- permanently (lasting globalv)

Variables that are related or used together may be associated in groups. This provides for more efficient retrieval and selective use. Group names are specified by the user. The default group name is “unnamed”.

The first use of GLOBALV creates a directory named .GLOBALV in the user’s \$HOME directory. Within this directory are files that correspond to each of the variable groups that have been defined. You may access another user’s global variables by including their \$HOME directory in your \$PATH.

GLOBALV	INIT	
	SELECT <i>group</i>	
<i>value_list</i>	SELECT <i>group</i>	SET
		SETS
		SETP
		SETL <i>var value</i>
		SETLS
		SETSL
		SETLP
		SETPL
		PUT <i>var_list</i>
		PUTS
		PUTP
		GET [<i>var_list</i>]
		LIST [<i>var_list</i>]
		STACK [<i>var_list</i>]
	PURGE	
	GRPLIST	
	GRPSTACK	

INIT initializes global variables. **GLOBALV** sets variables from the values stored. It sets variables from the lasting, session, and initialization files, in that order. If a variable name appears more than once in any of these files, the subsequent values override previous definitions so that the last value encountered is the value used.

SELECT specifies the name of the variable group to be used.

group is the name of the group to be accessed. **group** must be a literal string of 8 characters or fewer. Any characters are permitted in a group name. However, if the group name includes special characters, it is recommended that the GLOBALV command be addressed to the “command” host command environment to bypass any shell expansions that might occur.

If only **group** follows the **SELECT** keyword, this has the effect of setting the variable group for all subsequent GLOBALV commands.

SET, **SETS**, and **SETP** set the value of one or more variables for temporary, session, or lasting GLOBALV, respectively. Each of these keywords assigns a value to a variable and stores that variable with the group specified in the preceding **SELECT**.

value_list is a series of blank-delimited words in the form

var [*value*] [*var* [*value*]] ...

var may be any valid REXX variable name. **value** is the value to be assigned to **var**. **value** must be a constant or a literal string and may not contain any embedded blanks. If **var** is already defined, its value is replaced by the one specified in the current command.

SETL, **SETLS**, **SETSL**, **SETLP**, and **SETPL** set the value of a single variable. Each of these keywords assigns a value to a variable and stores that variable in the group specified in the preceding **SELECT**. **SETL** affects temporary GLOBALV; **SETLS** and **SETSL** affect session GLOBALV; and **SETLP** and **SETPL** affect lasting GLOBALV.

var may be any valid REXX variable name. **value** is the value to be assigned to **var**. **value** must be a constant

or a literal string. In this case, however, *value* may contain embedded blanks since only one variable may be set using these keywords.

PUT, **PUTS**, and **PUTP** assigns one or more variables for temporary, session, or lasting GLOBALV, respectively. The value stored for each variable is its current value at the time the GLOBALV command is executed.

var_list is a blank-delimited list of one or more valid REXX variable names.

GET retrieves one or more variables from global storage. *var_list* is a blank-delimited list of one or more valid REXX variable names.

LIST lists the variable name and its current values for one or more variables in the group specified. *var_list* is a blank-delimited list of one or more valid REXX variable names. If *var_list* is omitted, LIST lists all variables in the specified group.

The format of the variable listing is

```
SELECTED TABLE IS: group  
  var1=value  
  var2=value  
  :  
  :
```

If a PURGE has been executed on the specified group name prior to the LIST, only the first line shown above appears in the output.

STACK places the value of one or more variables on the uni-REXX program stack. Values are stacked in LIFO (last-in-first-out) order. Use the PULL or PARSE PULL instruction, described in *Chapter 4, Instructions*, to retrieve values from the program stack. The number of elements currently on the stack is accessible with the

QUEUED built-in function described in *Chapter 5, Built-In Functions*.

var_list is a blank-delimited list of one or more valid REXX variable names. If *var_list* is omitted, STACK has no effect.

PURGE removes all variables in the selected group from global storage. If no group has been selected, PURGE removes all variables in all groups.

GRPLIST displays a list of all existing group names.

GRPSTACK places the names of all existing groups on the uni-REXX program stack. A null line is written to the stack to indicate the end of the group names.

Examples:

```
/*
 * the following program fragments illustrate
 * the results of some GLOBALV commands
 */
#!/usr/local/bin/rxx
'globalv init'
'globalv select commvars' /* default group */
base_comm = 0.06
accel_level = 10000
accel_incr = 0.01
max_comm = 0.20
'globalv putp base_comm accel_level accel_incr'
'globalv putp max_comm'
'run_commissions' /* run commissions program */
:
:
exit

#!/usr/local/bin/rxx
thismo = date('m')
'globalv select commvars'
'globalv get base_comm accel_level accel_incr'
'globalv get max_comm'
'globalv select thismo'
'globalv get east west north south foreign'
: /* perform calculations */
:
```

MAKEBUF

The MAKEBUF command creates a new buffer within the uni-REXX program stack.

MAKEBUF

The return code from MAKEBUF is the number of the buffer just created. This value can be used with DROPBUF to clear one or more buffers when their contents are no longer needed.

All data PUSHed or QUEUED onto the stack after a MAKEBUF command is associated with this buffer until a subsequent MAKEBUF, DROPBUF, or DESBUF command is executed.

Use DROPBUF to remove the contents of one or more specific buffers from the stack. Use DESBUF to clear all buffers from the stack.

Examples:

```
/*
 * the following program fragments illustrate
 * the use of MAKEBUF, DROPBUF, and DESBUF
 */
push zero
'makebuf'; push 'one'
'makebuf'; push 'two'
'makebuf'; push 'three'
'makebuf'; push 'four'
'makebuf'; push 'five'
pull next; say next
pull next; say next
'dropbuf 3'
pull next; say next
'desbuf'
pull next; say next
/*
 * the output is
 *     FIVE
 *     FOUR
 *     TWO
 *     ZERO
 */
```

RXQUEUE

RXQUEUE redirects output to the uni-REXX program stack. It may be used as a command or as a filter.

RXQUEUE

When used as a command, RXQUEUE accepts input from the default character input stream (usually the keyboard) and places it on the program stack. Use CTL-D to terminate keyboard input mode and return control to the uni-REXX program.

When used as a filter, RXQUEUE redirects STDOUT from a command or program to the uni-REXX program stack. The syntax for this use is

```
cmd_or_prog `| rxqueue`
```

cmd_or_prog is the name of the command or program (including its calling arguments) for which output is to be placed on the stack.

SENTRIES

The SENTRIES command determines the number of entries in the uni-REXX program stack.

SENTRIES

The return code from SENTRIES is the number of items on the stack excluding any data currently pending in the STDIO stream.

Chapter 7: Application Programming Interfaces

The uni-REXX application programming interfaces (APIs) provide mechanisms to

- embed uni-REXX as a macro language within a larger application
- extend the capabilities of the uni-REXX interpreter by adding external function packages written in a compiled language

This chapter documents the APIs and provides examples of their use. Most of the examples illustrate using two or more APIs in combination. All of the examples are included in uni-REXX Sample Library, which is delivered on the product distribution media.

The uni-REXX APIs are modelled after the IBM implementations under TSO/E. The following APIs are supported:

IRXJCL

execute a uni-REXX program (simple interface)

IRXEXEC

execute a uni-REXX program (complete interface)

IRXSUBCM

manage host command environments

IRXEXCOM

access to uni-REXX program variables

IRXSTK

access the uni-REXX program stack

IRXEXITS

specify user-supplied exits

IRXSTOP

terminate the uni-REXX program

The APIs are function calls that can be included in a C language program. They use a number of control blocks (structures) which are defined in the header file `irx.h`, supplied on the distribution media. The header file `irx.h` must be included in any programs that use these APIs. Definitions and use of each of these control blocks are included at the end of this chapter.

The section entitled “Building an Embedded Application” in this chapter provides instructions on building an executable module of a C program that uses the uni-REXX APIs.

For portability between UNIX and OS/2, uni-REXX also includes support for OS/2-style APIs. These function substantially as they do in the OS/2 environment and further documentation is not provided here. An example of their use is included in the uni-REXX Sample Library in the file `func.tar.Z`.

IRXEXCOM

The IRXEXCOM interface provides access to uni-REXX program variables. It uses the shared variable block structure (SHVBLOCK) for communication between the uni-REXX program and the compiled language program.

The function declaration for IRXEXCOM in irx.h is:

```
int ORXXCDecl ORXXLoadDS irxexcom(SHVBLOCK *);
```

The following declarations are required:

```
SHVBLOCK varname;
```

varname may be any variable name of your choosing.

Before setting values in the shared variable block, you **must** initialize to 0 the memory locations to be used for the shared variable block. Use `memset` as in

```
(void) memset(&varname, 0, sizeof(SHVBLOCK));
```

varname is the name of your shared variable block.

The syntax for invoking the function in a C language application is:

```
irxexcom(&varname)
```

varname is the name of your shared variable block.

Multiple requests to IRXEXCOM may be made by chaining multiple instances of SHVBLOCK using the SHVNEXT field. In this case, the value returned by IRXEXCOM is set by ORing together the values returned by each separate request.

Return codes that may be set by IRXEXCOM are the following (the corresponding value of `shvret` is shown in parentheses):

-1	an internal error occurred; contact The Workstation Group
0	successful completion (SHVCLEAN)
1	a new variable was accessed (SHVNEWV)
2	last variable from SHVNEXTV (SHVLVAR)
4	variable truncated (SHVTRUNC); in this case, shvval contains the length required, before truncation
8	bad variable name (SHVBADN)
16	value too long (obsolete in uni-REXX V2.00 and later releases) (SHVBADV)
20	no uni-REXX program active
128	bad function passed in shvcode (SHVBADV)

Since IRXEXCOM may often be used to set the value of a uni-REXX program variable that did not previously exist, it may be most useful to check return codes by invoking the function as

```
if (irxexcom(&varname) <= SHVNEWV)
```


Examples:

The IRXEXCOM interface might typically be used in an initialization or termination exit, to set variables for use by the uni-REXX program or to retrieve values set by it. This example illustrates the setting of variables in an initialization exit. The first program is “excom.c”.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"
/*forward reference */
static void rxinit();

int main(argc, argv)
int argc;
char **argv;
{ int rc = 0;
/* declare an EXITBLK structure */
EXITBLK myexit;
(void) &argc;
(void) &argv;
/* set up the EXITBLK */
myexit.initialization = rxinit;
/* setup the exits */
irxexits(&myexit);
/* execute the uni-REXX program */
rc = irxjcl("excom.rex");
if (rc != 0)
    printf("Return code from irxjcl=%d\n", rc);
}

static void rxinit()
{
    int rc = 0;
/* declare variables for REXX */
static char v1[10];
static char v2[10];
/* declare a shared variable block */
SHVBLOCK rxvar;
/* initialize variables for REXX ..... */
strcpy(v1, "this is v1");
strcpy(v2, "this is v2");
/* clear memory of shared variable block */
(void) memset(&rxvar, 0, sizeof(SHVBLOCK));
/* data into shared variable block */
rxvar.shvcode = shvstore;
rxvar.shvnama = "v1";
rxvar.shvnaml = (int) strlen(rxvar.shvnama);
rxvar.shvvala = v1;
}
```

```

    rxvar.shvval = (int) strlen(rxvar.shvvala);
/* set this variable for uni-REXX */
    rc = irxexcom(&rxvar);
    if (rc > SHVNEWV)
        printf("irxexcom error, rc=%d\n", rc);
/* do it again for the next variable */
    memset(&rxvar, 0, sizeof(SHVBLOCK))
    rxvar.shvcode = shvstore;
    rxvar.shvnama = "v2";
    rxvar.shvnaml = (int) strlen(rxvar.shvnama);
    rxvar.shvvala = v2;
    rxvar.shvval = (int) strlen(rxvar.shvvala);
    rc = irxexcom(&rxvar);
    if (rc > SHVNEWV)
        printf("irxexcom error, rc=%d\n", rc);
}

```

The following program is “excom.rex”.

```

/*
 * excom.rex - demonstrate IRXEXCOM in an
 *             initialization exit
 */
say 'Now in the program excom.rex'
say 'v1 is:  ` v1
say 'v2 is:  ` v2

```

When you build and execute this application, the output is:

```

Now in the program excom.rex
v1 is:  this is v1
v2 is:  this is v2

```

IRXEXEC

The IRXEXEC interface provides a method for calling a uni-REXX program and facilities for passing calling arguments, accessing data returned by the program, or executing an in-storage block of REXX code.

The function declaration for IRXEXEC in irx.h is:

```
int ORXXCDecl ORXXLoadDS irxexec(EXECBLK *,
ARGLIST *, int, INSTBLK *, char **,
EVALBLOCK *, char *);
```

One or more of the following declarations may be required:

```
ARGLIST argvar;
EXECBLK exvar;
EVALBLOCK valvar;
INSTBLK stvar;
```

argvar, *exvar*, *valvar*, and *stvar* may be any variable names of your choosing. It is not necessary to declare a structure that is to be specified as NULL on the IRXEXEC call.

Before setting values in any of these control blocks, you **must** initialize to 0 the memory locations to be used for the block. Use `memset` as in

```
(void) memset(&varname, 0, sizeof(TYPE));
```

varname is the name of your control block (*argvar*, *exvar*, *valvar*, or *stvar*). *TYPE* is the structure type, such as ARGLIST or EXECBLK.

The syntax for invoking the function in a C language application is:

```
irxexec(&exvar, argvar, flag, &stvar, envptr, valvar,  
usrptr)
```

exvar is the name of the structure containing information about the uni-REXX program to be executed (EXECBLK). This block must always be defined if the program to be executed. If IRXEXEC is used to execute an in-storage block of REXX code, then the fields *dsnptr* and *pathptr* within the EXECBLK should be set to NULL.

argvar is the name of the structure containing the argument list. If no arguments are to be passed to the program, this may be specified as NULL.

flag indicates how the program to be executed is to be processed. It should be specified as one of the following:

0

program is a command

REXXCOMMAND

program is a command

REXXFUNCTION

program is a function; in this case the program **must** return a value

REXXSUBROUTINE

program is a subroutine

REXXSTICKY

program is sticky – that is, it should remain in memory after execution is complete

The flag values may be specified in upper- or lower-case but not in mixed case.

stvar is the name of the structure containing an in-storage block of REXX code. If the program to be executed is a disk file, the pointer to the in-storage block should be specified as NULL.

envptr is a pointer to the environment area or NULL. This field is not usually specified.

valvar is the name of the structure in which the data returned from the uni-REXX program is stored. This must always be specified.

usrptr is a user pointer or NULL. It is included for compatibility with TSO/E and should normally be specified as NULL.

Return codes that may be set by IRXEXEC are:

- 0 successful execution
- 20 program not executed; this may result if the uni-REXX program file is not found

Examples:

This example illustrates the use of IRXEXEC to invoke a uni-REXX program (“myexec.rex”) as a subroutine, passing it two arguments and printing the data returned.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

int main()
{
    int rc = 0;
    EXECBLK execut;
    ARGLIST ag[3];
    int fl = REXXFUNCTION;
    EVALBLOCK *values;
    char evaldata[100];

    /* initialize the execution block */
    (void) strcpy(execut.subcom, "rex");
    execut.dsnptr = "myexec.rex";
    execut.pathptr = NULL;

    /* initialize the argument list */
    memset(ag, 0, sizeof(ARGLIST));
    ag[0].argstring_ptr = "arg1";
    ag[0].argstring_length = strlen(ag[0].argstring_ptr);
    ag[1].argstring_ptr = "arg2";
    ag[1].argstring_length = strlen(ag[1].argstring_ptr);
    ag[2].argstring_ptr = NULL;

    /* initialize the evaluation block */
    memset(evaldata, 0, sizeof(evaldata));
    values = (EVALBLOCK *) evaldata;
    values->evsize = sizeof(evaldata);

    /* invoke the program */
    rc = irxexec(&execut, ag, fl, NULL, NULL, values, NULL);
}
```

```

    if (rc != 0)
    {
        printf("irxexec error, rc = %d\n", rc);
        return rc;
    }
    else

/* display the returned value
    printf("Value back from REXX = \"%s*\n",
        values->evlen, values->evdata);
*/
}

```

The following program is “myexec.rex”:

```

parse arg one, two
say 'First arg is: ' one
say 'Second arg is:' two
say ''
return_value = 'Hello world'
return return_value

```

When you build and execute this application, the output is:

```

First arg is:  arg1
Second arg is: arg2

```

```

Value back from REXX = "Hello world"

```

IRXEXITS

The IRXEXITS interface provides a facility for user-written exits to intercept certain uni-REXX operations before they are executed. The exit routine determines whether to perform its own processing or to let uni-REXX continue as if no exit were taken.

The function declaration for IRXEXITS in irx.h is

```
void ORXXCDecl ORXXLoadDS irxexits(EXITBLK *);
```

The following declarations are required:

```
EXITBLK varname;
```

varname is the name of your exit block.

The syntax for invoking the function in a C language application is:

irxexits(&varname)

varname is the name of your exit block.

The following types of exits are supported:

initialization

exit is entered before any instructions in the uni-REXX program are executed

termination

exit is entered after the last instruction in the uni-REXX program has been executed

command

exit is entered when the uni-REXX program encounters a host command

termin

exit is entered when the uni-REXX program encounters an instruction that pauses for terminal input; such instructions include PULL or PARSE PULL when there is no data on the program stack, PARSE LINEIN, and calls to CHARIN or LINEIN using the default input stream

termout

exit is entered when the uni-REXX program encounters an instruction that performs output to the terminal; such instructions include SAY, calls to CHAROUT or LINEOUT using the default input stream or specifying “stderr:” as the output stream, and TRACE; the exit is also entered prior to output of a uni-REXX diagnostic or error message

Only the initialization and termination exits do not set a return code for use by the uni-REXX program. For all the other exits, a return code of 0 indicates that the exit handled the processing and that uni-REXX should not

execute the instruction that triggered the exit. Any non-zero return code indicates that, regardless of any processing performed by the exit, uni-REXX should execute the instruction that triggered the exit.

Examples:

This example illustrates the use of the terminal input and output exits. Other examples in this chapter also include illustrations of user-written exits. Refer to the examples for IRXEXCOM for an initialization exit and the examples for IRXSTOP for a command exit.

For termin, the exit handles PULL, PARSE PULL PARSE LINEIN, and CHARIN() invocations but permits uni-REXX to handle LINEIN() invocations. For termout, the exit handles LINEOUT invocations by prefixing a message identifier to the output string; it permits uni-REXX to handle all other output operations.

This program is named “termex.c”.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"
static int trapin();          /* terminal input exit */
static int trapout();        /* terminal output exit */

int main(argc, argv)
int argc;
char **argv;
{
    EXITBLK termexits;
    int rc = 0;
    (void) &argc;
    (void) &argv;

    /* set up exit block */
    (void) memset(&termexits, 0, sizeof(EXITBLK));
    termexits.termin = trapin;
    termexits.termout = trapout;
    irxexits(&termexits);

    /* invoke the uni-REXX program */
    rc = irxjcl("termex.rex");
    if (rc != 0)
        printf("demoexit() REXX rc = %d\n", rc);
}

static int trapin(type, len, str)
int type;
int *len;
char *str;
{
```



```

(void) &str;
(void) strncpy(str, "Data from exit", *len);
*len = strlen(str);
printf("In the input exit\n");
if (type == IOLINEIN) /* if instr is LINEIN */
{
    printf("Press enter to continue\n");
    return 1; /* tells REXX to handle instr */
}
else /*for all other terminal input instrs */
    return 0; /* tell REXX we took care of it */
}

static int trapout(type, length, string)
int type;
int *length;
char *string;
{
    if (type == IOLINEOUT ) /* if instr is LINEOUT */
    {
        printf("Output exit prefixing lineout strings\n");
        printf("EXITMSG: %.*s\n", length, string);
        return 0; /* tells REXX we took care of it */
    }
    else /* for all other terminal output intrs */
        return 1; /* tell REXX to handle it */
}

```

The following program is named “termex.rex”.

```

say 'Now entering "termex.rex"'
say 'About to invoke linein()'
x = linein()
call lineout , 'Back from exit'
say 'Input string:' x
say 'About to invoke pull'
pull y
say 'Input string:' y
say 'About to invoke charin()'
x = charin(, ,14)
call lineout , 'Back from exit'
say 'Input string:' x

```

When you build and execute this application, the output is:

```
Now entering "termex.rex"  
About to invoke linein()  
In the input exit  
Press enter to continue
```

```
Output exit prefixing lineout strings  
EXITMSG: Back from exit  
Input string:  
About to invoke pull  
In the input exit  
Input string: DATA FROM EXIT  
About to invoke charin()  
In the input exit  
Output exit prefixing lineout strings  
EXITMSG: Back from exit  
Input string: Data from exit
```

IRXJCL

The IRXJCL interface provides a simple method for calling a uni-REXX program. It does not require the use of any control blocks.

The function declaration for IRXJCL in `irx.h` is

```
int ORXXCDecl ORXXLoadDS irxjcl(char *);
```

There are no required declarations.

The syntax for invoking the function in a C language application is:

```
irxjcl(program)
```

program is the name of the uni-REXX program to be executed, optionally followed by a space and calling arguments for the program. It must be specified as a null-terminated string. *program* may either be a literal string enclosed in double quotes or the name of a C language variable that has been set to a null-terminated string.

Because only a single parameter may be passed to IRXJCL, *program* must also include any calling arguments required by the uni-REXX program.

Return codes that may be set by IRXJCL are

0	successful completion
20	an unspecified error occurred

The returned value may also be set by specifying a value on the RETURN or EXIT instruction in the uni-REXX program.

Examples:

The first example illustrates the simplest use of IRXJCL. It simply calls a uni-REXX program and tests the return code. This program might be named “jcltest.c”.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

main()
{
    int rc = 0;
    rc = irxjcl("jcltest.rex");
    if (rc != 0)
        printf("IRXJCL error. rc = %d\n", rc);
}
```

The uni-REXX program “jcltest.rex” might look like the following:

```
say 'Hello world'
say 'I was called by the jcltest application'
```

When you build and execute this application, the output is:

```
Hello world
I was called by the jcltest application
```

A slightly more complex example of “jcltest.c” might include calling arguments.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

main()
{
    int rc = 0;
    rc = irxjcl("argtest.rex abc xyz");
    if (rc != 0)
        printf("IRXJCL error. rc = %d\n", rc);
}
```

The uni-REXX program “argtest.rex” might look like the following:

```
parse arg first second
say 'First arg is' first
say 'Second arg is' second
```

When you build and execute this application, the output is:

```
First arg is abc
Second arg is xyz
```

An alternative to hard-coding the program argument is to use a variable as in the following example of “jcltest.c”:

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

main()
{
    int rc = 0;
    char args[9];
    char prog[20];
    strcpy(args, " abc xyz");
    strcpy(prog, "argtest.rex");
    strcat(prog, args);
    rc = irxjcl(prog);
    if (rc != 0)
        printf("IRXJCL error.  rc = %d\n", rc);
}
```

This application uses the same uni-REXX program as in the previous example. When you build and execute the application, the output is also identical to the previous example.

IRXSTK

The IRXSTK interface provides an application with access to the uni-REXX program stack. It allows you to place data on the stack, retrieve data from the stack, or query the size of the stack.

The function declaration for IRXSTK in `irx.h` is:

```
int ORXXLoadDS irxstk(char *, char *, int, int *);
```

There are no required control block declarations.

The syntax for invoking the function in a C language application is:

```
irxstk(op, data, data_len, qvar)
```

op is the stack operation to be performed. It must be specified as a null-terminated string and must be one of the following, which correspond to uni-REXX instructions and built-in functions:

pull	retrieve a line from the stack
push	place a new line at the top of the stack
queue	place a new line at the bottom of the stack
queued	request the number of lines currently on the stack

data is the name of the variable that contains the data. If *op* is “push” or “queue”, *data* contains the value to be placed on the stack. If *op* is “pull”, the value of *data* is set to the line retrieved from the stack. If *op* is “queued”, this argument should be specified as “NULL”.

data_len is the length of the data. If *op* is “queued”, this argument is ignored and should be specified as 0.

qvar is the address of the variable in which the number of lines currently on the stack is stored.

Examples:

The example shown here illustrates manipulation of the uni-REXX program stack from within a special host command environment. The following program is named "stackem.c".

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

/* required forward reference */
int doit();

static char *stkinfo="Put on stack by C prg";

int main(argc, argv)
int argc;
char **argv;
{
    int rc = 0;

    /*Set up host command environment: staktst */
    IRXSUBCT staktst;
    (void) strcpy(staktst.name, "staktst");
    staktst.routine = doit;
    rc = irxsubcm("add", &staktst);
    if (rc != 0)
    {
        printf("irxsubcm add error, rc = %d\n", rc);
        exit(rc);
    }

    /* Run the uni-REXX program */
    rc = irxjcl("testit.rex");
    if (rc != 0)
    {
        printf("irxjcl error, rc = %d\n", rc);
        exit(rc);
    }

    /* Remove staktst host command environment */
    rc = irxsubcm("delete", &staktst);
    if (rc != 0)
    {
        printf("irxsubcm delete error, rc = %d\n", rc);
        exit(rc);
    }
}
```

```

    }
    return rc;
}

/* uni-REXX host command interface: */
/* routine executed whenever the REXX program*/
/* executes "address staktst" */

int doit(sr);
char *sr;
{
    int rc = 0;
    int sz;
    char stk[100];

/* Anything on the stack? */
rc = irxstk("queued", NULL, 0, &sz);
if (rc != 0)
{
    printf("irxstk queued error, rc=%d\n", rc);
    exit(rc);
}
else
{
    printf("Elements on stack = %d\n", sz);

/* Retrieve a value from the stack */
rc = irxstk("pull", stk, sizeof(stk), &sz);
if (rc != 0)
{
    printf("irxstk pull error, rc=%d\n", rc);
    exit(rc);
}
else
    printf("Data from stack: \"%s\"\n", stk);
}

/* Put data on stack for uni-REXX to read */
strcpy(stk, stkinfo);
rc = irxstk("push", stk, strlen(stk), &sz);
if (rc != 0)
{
    printf("irxstk push error, rc=%d\n", rc);
    exit(rc);
}
}
}

```


The following uni-REXX program is named "testit.rex":

```
say ''
say 'Now entering "testit.rex" program'
say ''
x = "Placed here by REXX program"
say 'Data to stack is:'
say '  ' x
push x
say ''
say 'Next command is "address staktst go"'
say ''
say 'Back from host command environment'
say 'Is there anything on the stack?'
say ''
say 'QUEUED() is:' queued()
pull whatigot
say ''
say 'Data from C program is:'
say '  ' whatigot
```

When you build and execute this application, the output is:

```
Now entering "testit.rex" program
```

```
Data to stack is:
```

```
  Placed here by REXX program
```

```
Next command is "address staktst go"
```

```
Elements on stack = 1
```

```
Data from stack = "Placed here by REXX program"
```

```
Back from host command environment
```

```
Is there anything on the stack?
```

```
QUEUED() is: 1
```

```
Data from C program is:
```

```
  PUT ON STACK BY C PRG
```

IRXSTOP

The IRXSTOP interface stops the currently executing uni-REXX program. The program stops when execution of the current clause is complete.

The function declaration for IRXSTOP in irx.h is:

```
int ORXXLoadDS irxstop(void);
```

There are no required control block declarations.

The syntax for invoking the function in a C language application is:

```
irxstop()
```

Return codes that may be set by IRXSTOP are

0	successful termination of the program
20	no program active

Examples:

The example that follows illustrates the use of IRXSTOP in a command exit. For each host command executed in the uni-REXX program, the exit examines the command string. If it is “quit”, the exit executes IRXSTOP to terminate the uni-REXX program. If the command string is not “quit”, the exit sets the proper return code to insure that the command is processed by the host command environment. The following program is named “stopit.c”.

```

#include <stdio.h>
#include <string.h>
#include "irx.h"
/* forward references */
static int cmd();
static int doit();

int main()
{
/* declarations for control blocks */
EXITBLK cmdexit;
IRXSUBCT test;
int rc = 0;

/* set up exit block; enable exit routines */
memset(&cmdexit, 0, sizeof(EXITBLK));
cmdexit.command = cmd;
irxexits(&cmdexit);

/* set up host command environment */
(void) strcpy(test.name, "test");
test.routine = doit;
if ((rc = irxsubcm("add", &test)) != 0)
{
printf("irxsubcm error, rc = %d\n", rc);
return rc;
}

/* invoke the uni-REXX program */
if ((rc = irxjcl("quitit.rex")) != 0)
{
printf("irxjcl error, rc = %d\n", rc);
return rc;
}

/* delete the host command environment */
if ((rc = irxsubcm("delete", &test)) != 0)
{
printf("irxsubcm error, rc = %d\n", rc);
return rc;
}
}

/*
* the command exit routine
*/
static int cmd(host_env, string, rc)
char *host_env;
char *string;
int *rc;
{
if (strcmp(string, "quit") == 0)

```

```

    {
        irxstop();
        return;
    }
    else
    {
        printf("Command exit - string not quit\n");
        *rc = 1;
        return;
    }
}

/*
 * the host command processing routine
 */
static int doit(sr)
char *sr;
{
    int rc = 0;
    printf("Command string is: %s\n", sr);
    return;
}

```

The following program is “quitit.rex”.

```

say 'Now beginning "quitit.rex"'
say 'Sending "hello" to test'
address test 'hello'
say 'Back from host environment'
say 'Sending "quit" to test'
address test 'quit'
say 'If this line prints, irxstop failed'

```

When you build and execute this application, the output is:

```

Now beginning "quitit.rex"
Sending "hello" to test
Command exit - string not quit
Command string is: hello
Back from host environment
Sending "quit" to test

```

IRXSUBCM

The IRXSUBCM interface is used to manage host command environments for uni-REXX programs.

The function declaration for IRXSUBCM in irx.h is:

```
int ORXXLoadDS irxsubcm(char *, IRXSUBCT *);
```

The following declarations are required:

```
IRXSUBCT varname;
```

varname may be any variable name of your choosing.

The syntax for invoking the function in a C language application is:

```
irxsubcm(function, &varname)
```

function is the function to perform. It must be specified as a null-terminated string and must be one of the following:

add	add a new host command environment
delete	delete an existing host command environment
update	update an existing host command environment

varname is the name of your host command block.

Return codes that may be set by IRXSUBCM are

0	function successful
20	an error occurred, such as an invalid function request
28	host command environment not found

Examples:

The example which follows illustrates creating, modifying, and deleting a host command environment. It also illustrates the effects of the modification on subsequent executions of the uni-REXX program. The following program is named "subcom.c".

```
#include <stdio.h>
#include <string.h>
#include "irx.h"

/* forward references */
static int cmd();
static int doit();

int main()
{
/* declare host command environment block */
  IRXSUBCT test;
  int rc = 0;

/* set up host command environment */
  (void) strcpy(test.name, "test");
  test.routine = doit;
  rc = irxsubcm("add", &test)
  if (rc != 0)
  {
    printf("irxsubcm error, rc = %d\n", rc);
    return rc;
  }

/* execute the uni-REXX program */
  rc = irxjcl("hostcmd.rex CALL1")
  if (rc != 0)
  {
    printf("irxjcl error, rc = %d\n", rc);
    return rc;
  }

/* change the host command environment */
  test.routine = cmd;
  rc = irxsubcm("update", &test);
  if (rc != 0)
  {
    printf("irxsubcm error, rc = %d\n", rc);
    return rc;
  }
}
```

```

/* execute the uni-REXX program again          */
rc = irxjcl("hostcmd.rex CALL2")
    if (rc != 0)
    {
        printf("irxjcl error, rc = %d\n", rc);
        return rc;
    }

/* remove the host command environment        */
rc = irxsubcm("delete", &test);
    if (rc != 0)
    {
        printf("irxsubcm error, rc = %d\n", rc);
        return rc;
    }
}

static int doit(sr)
char *sr;
{
    int rc = 0;
    printf("Message from C routine 'doit'\n");
    printf("Command string is: %s\n", sr);
    return;
}

static int cmd(sr)
char *sr;
{
    int rc = 0;
    printf("Message from C routine 'cmd'\n");
    printf("Command string is: %s\n", sr);
    return;
}

```

The following program is “hostcmd.rex”.

```

parse arg parm
say 'Called with' parm
say ''
say 'Now executing host command'
address test parm
say ''
say 'Back from host command'

```

When you build and execute this application, the output is:

```
Called with CALL1
```

```
Now executing host command  
Message from C routine 'doit'  
Command string is: CALL1
```

```
Back from host command  
Called with CALL2
```

```
Now executing host command  
Message from C routine 'cmd'  
Command string is: CALL2
```

```
Back from host command
```


Building Embedded Applications

The UNIX make facility provides the capabilities needed to build applications that embed uni-REXX as a macro language. This section describes the requirements for building the application and some example make files.

Building the application is a two step process involving

- compiling the C language program
- generating an executable binary

The UNIX “cc” command (or its equivalent) can be used for both steps because “cc” automatically invokes the “ld” (load) command to create a binary unless explicitly instructed to do otherwise.

Typically, the command name “cc” is used to invoke the C compiler. At some sites this may be an alias for or a symbolic link to a differently named compiler module. Some sites may prefer to use a different compiler (such as “gcc” – GNU C) and choose to invoke it directly by the name under which it is installed.

The “cc” command is most often located in /usr/bin; or a symbolic link in /usr/bin points to the actual location. Since /usr/bin is typically in each user’s PATH, access to the command is automatic. If this is not the case, it may be necessary to add the location of the compiler to your PATH environment variable before attempting to build your application.

If you are having difficulty accessing the C compiler at your site, contact your system administrator for assistance.

The load phase requires access to one or more libraries to resolve function calls and system calls that appear in your program. The C and system libraries are typically found in a location known to the compiler and are automatically searched. On some systems, it may be necessary to set a special environment variable to access the right version of these libraries. As an example, Sun’s SUNWsprow compiler does not use the libraries in

/usr/lib and requires that the environment variable LD_LIBRARY_PATH point to the correct libraries for successful loading. If you are having difficulty successfully creating an executable binary, contact your system administrator for assistance in finding the correct libraries.

In addition to these system facilities, building an embedded application requires access to the following files, delivered on the uni-REXX distribution media:

irx.h	the uni-REXX header file that contains the API function and control block declarations
irxproto.h	additional header file used internally
librx.a	the uni-REXX archive library that contains the interpreter object code

Be sure that you are using these files from the same version of uni-REXX. A mismatch between the version levels of irx.h and librx.a will cause unpredictable and undesirable results. A call to ORXXVersionCheck() in your C program will generate an error if a mismatch exists.

If you have only a single source program, a single “cc” command can be used to compile your program and generate a binary executable. Such a “cc” command would look something like

```
cc -I/usr/local/rexx app.c -L/usr/local/rexx -lrx -lm -o app
```

The components of this command are

<code>cc</code>	the command to invoke the compiler; it automatically invokes the “ld” command because the “-c” flag is not present
<code>-I/usr/local/rexx</code>	specifies the directory to search for the <code>irx.h</code> header file – <code>/usr/local/rexx</code> in this example, but this may differ at your site
<code>app.c</code>	the name of the source program to use as input to this entire process
<code>-L/usr/local/rexx</code>	specifies the directory to search for the <code>librx.a</code> archive library – <code>/usr/local/rexx</code> in this example, but this may differ at your site
<code>-lrx</code>	instructs the loader to search the file “ <code>librx.a</code> ” to resolve external references
<code>-lm</code>	instructs the loader to search the file “ <code>libm.a</code> ” to resolve external references; this is the math library and is required for uni-REXX
<code>-o app</code>	specifies the name to be assigned to the executable binary that is created; if this flag and a name are not specified, the default name is “ <code>a.out</code> ”

It is also possible to use environment variables to reference the locations for the header and library files. If you set the environment variable `REXXLIB` to include the directory in which `librx.a` and `irx.h` are found, the “`cc`” command on the preceding page could be modified to read

```
cc -I$(REXXLIB) myapp.c -L$(REXXLIB) -lrx -lm -o myapp
```

For ease of maintenance, it is probably preferable to create a makefile that can be executed whenever you need to rebuild the application. A makefile to automate the example illustrated above would look like:

```
REXXLIB=/usr/local/rexx
myapp: myapp.c $(REXXLIB)
       cc -I$(REXXLIB) myapp.c -L$(REXXLIB) -lrx -lm -o myapp
```

The first character on the line that contains the “cc” command **must** be a tab character. The lines in this file do the following:

- line 1 set the environment variable REXXLIB to specify the location of irx.h and librx.a
- line 2 specify the target of this make process; also specify the dependencies for the make
- line 3 compile the source and generate the executable binary

If the name of this file is “makefile” or “Makefile”, you execute it simply by typing the command “make”. You may give it a more descriptive name, if you prefer, such as “mkmyapp”. In this case, execute it by typing

```
make -f mkmyapp
```

If your application requires more than one source file, the make facility is the best approach for managing the build. In the example make file that follows, three source programs are required:

```
REXXLIB=/usr/local/rexx
myapp: app1.o app2.o app3.o $(REXXLIB)
       cc -L$(REXXLIB) -lrx -lm -o myapp app1.o app2.o app3.o
.c.o:
       cc -c -I$(REXXLIB) *.c
```

As in the previous example, the first character on each line that contains the “cc” command **must** be a tab character.

Your source program may require additional include files and access to other libraries. It may also require additional compiler or loader flags. Refer to your system man pages for “cc” and “ld” to determine what ad-

ditional flags may be necessary. Your system administrator may also be an excellent resource for researching these topics.

Input files to “make” can also be far more sophisticated and complex, depending on the requirements of your application. Refer to your system man pages for “make” or to other topical references for guidance in creating your make files.

External Function Packages

uni-REXX supports the use of external function packages. Such functions are often written in C, but may also be written in any language that may be called from C and that supports the control block structures required by uni-REXX. This section gives details for writing external function packages in C.

You may use an external function package in one of two ways:

- as part of an embedded application, where uni-REXX programs are executed from a C language application using one of the APIs described earlier in this chapter
- within a stand-alone uni-REXX program

For embedded applications, the function package is linked directly into the application. For stand-alone uni-REXX programs, the -G option of the rxc command permits you to generate a version of the interpreter that includes the function package. The control blocks and variables required are identical for either type of application.

Control Blocks

The use of external function packages involves control blocks that use the following types:

FPCKDIR

the function package directory. You must provide directory entries for each function in the package.

ARGLIST

the function argument list. uni-REXX uses an array of this type to pass arguments to the function. This array is the first argument for each function in the package.

EVALBLOCK

values returned by the program. uni-REXX uses this structure to retrieve returned values from each function. It is the first argument of each function in the package.

Writing a Function Package

Each function in the package should be defined as in the example below for a function named "loc1".

```
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    :
    function code
    :
}
```

The examples used in this document use the `#ifdef/#else/#endif` syntax to illustrate coding alternatives between ANSI-compliant C compilers and non-ANSI-compliant compilers. The ANSI-compliant syntax is shown first (following `#ifdef __STDC__`) with the older syntax shown as the `#else` alternative. You need use only one of these options; and most compilers will process the older syntax correctly.

The function code may retrieve and process function arguments. As an example, the following code prints, for each argument:

```
from loc1: argstring_ptr = arg-n
```

where *arg-n* is the function argument.

```
while (ag->argstring_ptr != NULL)
{
    printf("from loc1: argstring_ptr = %.s\n",
        ag->argstring_ptr, ag->argstring_ptr);
    ++ag;
}
```

The function code may also set a return value for use by uni-REXX. This may be a simple numeric return code or any appropriate return value. In a simple example, the function named "loc1" returns its name as the return value using the following code:

```
strncpy(ev->evdata, "loc1", ev->evsize - 1);
ev->evlen = strlen(ev->evdata);
```

Thus, if the uni-REXX program contains the simple statement

```
call loc1 hello
```

the value of the special variable RESULT is "loc1".

Alternatively, if the program contains the statement

```
say loc1(anything)
```

the output of the uni-REXX "say" statement is "loc1".

By combining all these elements, you have a simple function named "loc1" which

- receives one or more arguments from the function
- prints the string 'from loc1: argstring_ptr = arg-n' for each argument
- sets the return value "loc1" for use by the uni-REXX program

The combined code looks like the following:

```

#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc1: argstring_ptr = %.s\n",
            ag->argstring_ptr, ag->argstring_ptr);
        ++ag;
    }
    strncpy(ev->evdata, "loc1", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}

```

To complete the function package, you must add an entry for `loc1` in a function package directory. For convenience, use the existing directory named `irxfloc`. This definition looks like

```

FPCKDIR irxfloc[] =
{
    {"loc1", loc1},
    {NULL, NULL}
};

```

To avoid problems with forward references, you may want to place this definition after the function code.

You must also include appropriate headers at the beginning of the package. For this example, you need only

<stdio.h>	for input/output processing
<string.h>	for the <code>strncpy</code> function
"irx.h"	for uni-REXX compatibility

Your complete function package should therefore look like:

```
#include <stdio.h>
#include <string.h>
#include "irx.h"
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc1: argstring_ptr = %.s\n",
            ag->argstring_ptr);
        ++ag;
    }
    strncpy(ev->evdata, "loc1", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}
FPCKDIR irxfloc[] =
{
    {"loc1", loc1},
    {NULL, NULL}
};
```

You may place your function package directory definition at the beginning of the package, but this requires that you first define each function, then define the function package directory, and finally include the executable code for each function. The result would look something like the following:

```

#include <stdio.h>
#include <string.h>
#include "irx.h"
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1();
#endif
FPCKDIR irxfloc[] =
{
    {"loc1", loc1},
    {NULL, NULL}
};
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#endif
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc1: argstring_ptr = %s\n",
            ag->argstring_ptr);
        ++ag;
    }
    strncpy(ev->evdata, "loc1", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}

```

Typically, an external function package will contain a number of functions for use by uni-REXX programs. In such cases, the function package code includes a definition of each function, and there is an entry in one of the function package directories (irxfloc, irxuser, or your own directory name) for each function in the package.

Execution Options

The next step depends on whether this function package is to be part of an embedded application or if it is to be used in stand-alone uni-REXX programs.

Embedded Applications. In an embedded application, the function package would normally be part of a larger application that includes other processing. Such an ap-

plication would already contain code for one of the uni-REXX APIs to invoke a uni-REXX program from within the C program.

The simplest case uses IRXJCL to invoke the uni-REXX program. The code necessary to call the program “demofunc.rex” from the sample function package is as follows:

```
main()
{
    int rc = irxjcl("demofunc.rex");
    printf("demofunc() rc = %d\n", rc);
}
```

Add it to the end of the example function package as shown below.

```
#include <stdio.h>
#include <string.h>
#include "irx.h"
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc1: argstring_ptr = %.s\n",
            ag->argstring_ptr, ag->argstring_length);
        ++ag;
    }
    strncpy(ev->evdata, "loc1", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}
FPCKDIR irxfloc[] =
{
    {"loc1", loc1},
    {NULL, NULL}
};
main()
{
    int rc = irxjcl("demofunc.rex");
    printf("demofunc() rc = %d\n", rc);
}
```

You then build the application as described in the section “Building Embedded Applications” in this chapter.

Stand-alone uni-REXX Programs. If you want to use your function package in stand-alone uni-REXX programs, you must create a new version of the uni-REXX interpreter that includes your function package. Several options of the `rxcc` command are provided for this purpose:

- G** required to generate a new interpreter
- L** used optionally to specify the location of object libraries
- l** used optionally to specify the name of user-supplied object libraries
- o** to specify the name for the new interpreter; if omitted, the default name “`rxcc`” is used

The `-L`, `-l`, and `-o` options function like their `cc(1)` counterparts.

Generation of a new interpreter requires access to the uni-REXX library `librx.a`. This access is provided through the environment variable `REXXLIB`, which must point to the directory where `librx.a` resides.

The steps to generate a new interpreter are the following:

1. Compile your function package (“`fpack`” for this example), using appropriate `cc` options including `-c` and `-o` to create `fpack.o`.

2. Set the environment variable REXXLIB to point to the directory containing librx.a. If librx.a is located in /usr/local/bin, you would type

C Shell: **setenv REXXLIB /usr/local/rexx**

Bourne Shell: **REXXLIB = /usr/local/rexx**
export REXXLIB

Korn Shell: **REXXLIB = /usr/local/rexx**
export REXXLIB

3. Generate a new interpreter using rxc. For the simple example here, it is sufficient to type

```
rxc -G fpack.o
```

This produces a new version of “rxx” that contains the functions in fpack. If you wish to have a separate version of the interpreter for use with these functions, use

```
rxc -G fpack.o -o frxx
```

You would then use frxx to run any uni-REXX program that uses the functions defined in fpack.

In this case, you do NOT include the API code in your function package code.

Examples:

An example of a function package for use in an embedded application follows. The application (“xmpl.c”) defines two functions (“loc1” and ”loc2”) that are defined as part of the “irxfloc” package. This application uses the API IRXJCL to invoke the uni-REXX program “xmpl.rex”. Listings of “xmpl.c” and “xmpl.rex” appear below along with sample output. This example is included in the uni-REXX Sample Library provided on the distribution media.

The listing which follows is “xmpl.c”.

```

#include <stdio.h>
#include <string.h>
#include "irx.h"
#ifdef __STDC__
void loc1(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc1(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc1: argstring_ptr = \"%.*s\\n\"",
            ag->argstring_length, ag->argstring_ptr);
        ++ag;
    }
    strncpy(ev->evdata, "loc1", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}
#ifdef __STDC__
void loc2(ARGLIST *ag, EVALBLOCK *ev)
#else
void loc2(ag, ev)
ARGLIST *ag;
EVALBLOCK *ev;
#endif
{
    while (ag->argstring_ptr != NULL)
    {
        printf("from loc2: argstring_ptr = \"%.*s\\n\"",
            ag->argstring_length, ag->argstring_ptr);
        ++ag;
    }
    strncpy(ev->evdata, "loc2", ev->evsize - 1);
    ev->evlen = strlen(ev->evdata);
}
FPCKDIR irxfloc[] =
{
    {"loc1", loc1},
    {"loc2", loc2},
    {NULL, NULL}
};
main()
{
    int rc = irxjcl("xmpl.rer");
    printf("xmpl() rc = %d\\n", rc);
}

```

The listing which follows is “xmpl.rex”.

```
say loc1(hello)
say loc1(hello,goodbye)
say loc2(wrkgrp)
call loc2 1,2,3
say result
call loc1 test
if result \= 'loc1' then say 'loc1 call failed'
    else say 'loc1 call successful'
```

This is the uni-REXX program that is executed through the IRXJCL API in “xmpl.c”.

Build the example application, naming the executable binary “xmpl”. To run the example program, type

```
xmpl
```

The output should look like the following:

```
from loc1: argstring_ptr = "HELLO"
loc1
from loc1: argstring_ptr = "HELLO"
from loc1: argstring_ptr = "GOODBYE"
loc1
from loc2: argstring_ptr = "WRKGRP"
loc2
from loc2: argstring_ptr = "1"
from loc2: argstring_ptr = "2"
from loc2: argstring_ptr = "3"
loc2
from loc1: argstring_ptr = "TEST"
loc1 call successful
xmpl() rc = 0
```

The lines that begin with “from” result from the printf statements in the functions “loc1” and “loc2”. The lines that begin with the function name result from the “say” statements in “xmpl.rex”. The last line of output is from the printf statement following the execution of IRXJCL in “xmpl.c”.

If you wish to use “xmpl.rex” as a stand-alone uni-REXX program, you must modify “xmpl.c” and generate a special version of the interpreter that contains your function package.

1. Remove the last five lines (beginning with “main()”) from xmpl.c.

2. Recompile xmpl.c using

```
cc xmpl.c -c -o xmpl.o
```

3. Set the environment variable REXXLIB to the location where the file “librx.a” resides. This is normally where uni-REXX is installed.

4. Generate the special version of the interpreter using

```
rxcc -G xmpl.o -o myrxx
```

To run “xmpl.rex” using your specially generated interpreter, type

```
myrxx xmpl.rex
```

The output from this execution is identical to that shown above except that the last line (the result of the IRXJCL execution) is not included.

Control Blocks The remaining sections of this chapter discuss the control blocks used by the APIs. Previous sections have made reference to the use of these control blocks. Detailed documentation is provided here.

ARGLIST ARGLIST is the structure used by uni-REXX to pass argument strings. ARGLIST contains the following fields:

argstring_ptr a pointer to the location where the argument string is stored

argstring_length the length of the argument string

The final element in the array of ARGLIST structures has `argstring_ptr` set to `NULL` and `argstring_length` set to zero.

The definition for ARGLIST in `irx.h` is

```
typedef struct arglist
{
    char *argstring_ptr;
    int argstring_length;
}
ARGLIST;
```

CPCKDIR CPCKDIR is the structure that is used to create a directory of all command packages available in uni-REXX.

The definition of CPCKDIR in `irx.h` is:

```
typedef struct
{
    const char *cmdname;
    int (*cmdaddr)(int argc,
                  char **argv);
}
CPCKDIR;
```

EVALBLOCK

EVALBLOCK is the structure used by uni-REXX to store values returned by the uni-REXX program. EVALBLOCK contains the following fields:

evsize	the maximum size of the return value
evlen	the length of the value actually returned
evdata	the location of the returned data; this field is declared as a one-byte character array but is actually an array of evsize length; for external functions, the size of evdata is generally limited to about 250 characters, though this limitation may vary among different platforms

The definition for EVALBLOCK in irx.h is

```
typedef struct evalblock
{
    int evsize;
    int evlen;
    char evdata[1];
}
EVALBLOCK;
```

EXECBLK

EXECBLK is the structure used by uni-REXX to describe the program name, program location, and default host command environment for IRXEXEC. EXECBLK contains the following fields:

subcom	the name of the default host command, specified as a null-terminated string
dsnptr	a pointer to the location that contains the uni-REXX program file name
pathptr	a pointer to the location that contains the name of the environment variable which identifies the search path for uni-REXX programs

If `pathptr` is `NULL`, a default path is created by concatenating the default host command environment name with the string “`PATH`”. Thus, if the default host command environment name is “`TPSYS`” and `pathptr` is `NULL`, the pathname that will be used to locate uni-REXX programs is “`TPSYSPATH`”.

The definition for `EXECBLK` in `irx.h` is

```
typedef struct execblk
{
    char *dsnptr;
    char *pathptr;
    char subcom[ORXXSubComMaxLength + 1];
}
EXECBLK;
```

EXITBLK

`EXITBLK` is the structure used by uni-REXX to contain the exit addresses that are passed to the interpreter. Each exit address should point to a routine of the correct type for that exit. This routine is then called by uni-REXX at the point of execution which corresponds to the exit.

The definition of `EXITBLK` in `irx.h` is

```
typedef struct exitblk
{
    int (ORXXCDecl ORXXLoadDS * termout)(int, int, char *);
    int (ORXXCDecl ORXXLoadDS * termin)(int, int *, char *);
    int (ORXXCDecl ORXXLoadDS * command)(char *, char *, int *);
    void (ORXXCDecl ORXXLoadDS * termination)(int, int);
    void (ORXXCDecl ORXXLoadDS * initialization)(void)
        : /* defines for terminal I/O types */
        : /* and termination types */
}
EXITBLK;
```

A typical exit block definition would look something like

```
EXITBLK myexits:
:
:
myexits.initalization = init;
myexits.command = cmd;
myexits.termination = atend;
irxexits(&myexits);
```

The parameters for each exit routine are as follows:

termout

I/O type – from the list below; this is passed to the exit and is determined by the I/O instruction that triggered the exit

output buffer length – passed to the exit

output buffer – passed to the exit

termin

I/O type – from the list below; this is passed to the exit and is determined by the I/O instruction that triggered the exit

pointer to the input buffer length – the buffer length is passed to the exit; the exit should return the length of the data this it returns

input buffer – passed to the exit

I/O types

IOCHARIN	set by charin() from STDIN
IOCHAROUT	set by charout() to STDOUT
IODIAGNOSTIC	set by a diagnostic message
IOERROR	set by an error message
IOLINEIN	set by LINEIN() from STDIN
IOLINEOUT	set by LINEOUT() to STDOUT
IOPARSE	set by PARSE PULL from terminal or PARSE LINEIN
IOPULL	set by PULL from terminal
IOSAY	set by SAY
IOTRACE	set by trace output or interactive trace input
IOCHARERR	set by CHAROUT(“stderr:”)
IOLINEERR	set by LINEOUT(“stderr:”)

command

host command environment name – passed to the exit

command buffer – passed to the exit

pointer to command return code – set by the exit

termination

termination reason

return code – set by the exit; must be one of the following:

TERMNORMAL normal termination

TERMERROR an error occurred

TERMHALT halt condition is raised

initialization

none

FPCKDIR

FPCKDIR is a structure used to create function directories for each function package available to uni-REXX. FPCKDIR contains the following fields:

funcname	the name by which this function is referenced in a uni-REXX program
funcaddr	the name of the C routine that performs the processing

The definition for FPCKDIR in irx.h is

```
typedef struct
{
    char *funcname;
    int (*funcaddr)(ARGLIST *arglist,
                   EVALBLOCK *evalblock);
}
FPCKDIR;
```

There are two default function directories:

```
irxfloc  
irxfuser
```

Each directory is actually an array of arbitrary length that contains any number of funcname/funcaddr pairs. The last entry in the array must have both funcname and funcaddr set to `NULL` to indicate the end of the array. A typical function directory definition should look something like

```
FPCKDIR irxfloc[] =  
{  
    {"loc1", loc1},  
    {"loc2", loc2},  
    {NULL, NULL}  
};
```

This defines two functions, `loc1` and `loc2`. Note the required quotation marks around the character variable funcname. Note also that it is not necessary to use the same name for funcname and funcaddr. Further, you may define aliases by using more than one name for the same function.

The two default function directories are defined in `irx.h` by

```
FPCKDIR *irxpackt[] =  
{  
    irxfuser,  
    irxfloc,  
    NULL  
};
```

`irxpackt` is an array of pointers to structures of type `FPCKDIR` and is a global variable in `uni-REXX`. If `irxpackt` is defined in your function package code, that definition overrides the `uni-REXX` default. You may therefore add your own function package directories by including a definition of `irxpackt` in your function pack-

age code. As an example, to add a directory named “myfp”, include the following definition in your code:

```
FPCKDIR *irxpackt[] =
{
    irxfuser,
    irxfloc,
    myfp,
    NULL
};
```

Note that `irxpackt` is also an array of arbitrary length and that the last element must be `NULL` to indicate the end of the array.

There are no requirements that you use a specific directory name for a specific purpose. You may include the functions in your function package in either of the two default directories or in a new directory that you have added. You may use the different directory names to logically organize the functions in your application, or you may include all your functions in a single directory.

INSTBLK

`INSTBLK` is the structure used by uni-REXX to describe a uni-REXX program that is currently in memory. It contains the following fields:

address	a pointer to an array of structures of type <code>STMT</code> ; the <code>STMT</code> structure describes a line of a uni-REXX program and is described in detail in this chapter
usedlen	the length of the <code>STMT</code> array in bytes
dsname	a pointer location that contains the name of the uni-REXX program; this is used by the <code>PARSE SOURCE</code> instruction

The definition of INSTBLK in irx.h is

```
typedef struct instblk
{
    STMT *address;
    int usedlen;
    char *dsname;
}
INSTBLK;
```


IRXSUBCT

IRXSUBCT is the structure used by uni-REXX to pass information about the host command environment to IRXSUBCM. It contains the following fields:

name	the name of the host command environment, specified as a null-terminated string
routine	the address of the routine that processes host commands; the host command is passed to the routine as a null-terminated string

The definition of IRXSUBCT in irx.h is

```
typedef struct irxsubct
{
    int (ORXXDecl ORXXLoadDS * routine)(char *)
    char name[ORXXSubComMaxLength + 1]
}
IRXSUBCT;
```

A typical host command environment definition would look something like this

```
IRXSUBCT tpsys;
:
:
(void) strcpy(tpsys.name, "tpsys");
tpsys.routine = tape_sys;
:
:
int tape_sys(nm)
char *nm;
:
:
```

SHVBLOCK

SHVBLOCK is the structure used to share variables between an application that embeds uni-REXX as a macro language and the uni-REXX program(s) that it executes. It contains the following fields:

shvnext	a pointer to the next shared variable block; this field should be specified as NULL if there are no more shared variable blocks
shvuser	the length of the variable name for the variable returned by the shvnextv function
shvcode	the shared variable function to be performed; this must be specified as a null-terminated string and must be one of the function names in the following list; the function may be specified in upper- or lowercase but not in mixed case

SHVSTORE

store the value of the specified variable; no symbolic substitution for compound variable tails is performed

SHVFETCH

fetch the value of the specified variable; no symbolic substitution for compound variable tails is performed

SHVDROPV

drop the specified variable

SHVSYSET

store the value of the specified variable; perform symbolic substitution on compound variable tails

SHVSYFET

fetch the value of the specified variable; perform symbolic substitution on compound variable tails

SHVSYDRO

drop the specified variable; perform symbolic substitution on compound variable tails

SHVNEXTV

fetch the value of the next variable

SHVPRIV

fetch private information

shvret

the return code flag set by the shared variable function; it will be one of the following; when testing the value of shvret, the return flag may be specified in upper- or lowercase, but not in mixed case

SHVCLEAN

function completed successfully

SHVNEWV

the specified variable did not exist

SHVLVAR

the shvnextv function retrieved the last available variable

SHVTRUNC

truncation occurred during a fetch function

SHVBADN

the specified variable name is invalid

SHVBADV

the value of the specified variable is too long

SHVBADV

the function code specified is invalid

shvbufl

the length of the fetch value buffer

shvnama	the address of the variable name
shvnaml	the length of the variable name
shvvala	the address of the value buffer
shvval	the length of the value

The definition of SHVBLOCK in irx.h is

```
typedef struct shvblock
{
    struct shvblock *shvnext;
    int shvuser;
    char shvcode;
    char shvret;
    int shvbufl;
    char *shvnama;
    int shvnaml;
    char *shvvala;
    int shvval;
    : /* defines for function codes and */
    : /* return codes */
}
SHVBLOCK;
```

A typical shared variable block definition would look something like

```
SHVBLOCK rxvar;
:
:
(void) memset(&rxvar, 0, sizeof(SHVBLOCK))
rxvar.shvcode = shvstore;
rxvar.shvnama = "v2";
rxvar.shvnaml = (int) strlen(rxvar.shvnama);
rxvar.shvvala = v2;
rxvar.shvval = (int) strlen(rxvar.shvvala);
rc = irxexcom(&rxvar);
```

STMT

STMT is the structure used by uni-REXX to contain lines of the uni-REXX program currently in memory. It contains the following fields:

stmt	a pointer to a string containing a line from the program currently in memory
stmtlen	the length of the line pointed to by stmt

The definition of STMT in irx.h is

```
typedef struct
{
    char *stmt;
    int stmtlen;
}
STMT;
```

Interprocess Communication with uni-REXX

You may wish to create your own stand-alone C language programs that communicate directly with uni-REXX. The uni-REXX archive library `librxi.a` is provided for this purpose.

The type of communication between your stand-alone application and uni-REXX is selected from the application programming interfaces described earlier in this chapter. For example, if you wish to use the uni-REXX program stack to share data, use the `IRXSTK` API in your stand-alone program. Your application must include the uni-REXX header file “`irx.h`”.

The only difference in creating such a stand-alone program and building an embedded application is the inclusion of the `librxi.a` library in the library search path. A simple “`cc`” command to build a stand-alone program that communicates with uni-REXX is

```
cc -I$(REXXLIB) app.c -L$(REXXLIB) -lrxi -lrx -lm -o app
```

This assumes, of course, that you have set the environment variable `REXXLIB` to point to the location of `irx.h`, `librx.a`, and `librxi.a` on your system. The library `librxi.a` **must** be searched before `librx.a`. The order of the “`-l`” flags on the “`cc`” command is critical.

Refer to the section entitled “Building Embedded Applications” earlier in this chapter for a detailed discussion of compiling and building C language applications.

Examples:

The following program, named “`rxqueue.c`”, is an example of a stand-alone application that communicates with uni-REXX through the program stack.

```

/*
** Copyright(C) ix Corporation 1993-1994.
** All rights reserved.
**
** Module =
**
**     rxqueue.c
**
** Abstract =
**
**     This is the rxqueue routine.  It's a demonstra-
**     tion of how inter-process APIs work (just like
**     the ones that are linked into an application, only
**     slower due to IPC overhead).
**
**     This program works like the OS/2 rxqueue command.
**     It copies it's
**     standard input to Open-REXX's stack.  This is
**     useful for capturing output of a command.
**     For example:
**
**     "ls | rxqueue"
**
**     would place the ls command's output in the
**     uni-REXX stack.
**
** History =
**
**     20-Apr-92 nfnm 0.00 added this comment
**
** Possible future enhancements =
**
**     Demonstrate other APIs, such as irxexcom()
**     updating variables.
**
*/
/*
* buffer size increments
*/
#define RXQUBFSZ 100
/*
* includes
*/
#include <stdio.h>
#include "irx.h"
/*
** Routine =
**
**     main
**
** Abstract =
**
**     main() for rxqueue
**
** Parameters =
**
**     1) argc
**     2) argv
**
** Returns =
**
**     void
**
** Possible future enhancements =

```

```

**
*/

#ifdef ORXXPrototype
int ORXXCDecl main(int argc, char **argv)
#else
int ORXXCDecl main(argc, argv)
int argc;
char **argv;
#endif
{
/*
 * input character
 */
int ch;
/*
 * return code
 */
int rqrc = 0;
/*
 * input buffer size and length remaining
 */
unsigned srsz = RXQUBFSZ, srle = srsz;
/*
 * input buffer and input buffer pointer
 */
char *sr, *srpt;
/*
 * create the buffer
 */
if ((sr = malloc(srsz)) == NULL)
{
(void) printf("rxqueue - out of memory, rc = 20\n");
rqrc = 20;
}
srpt = sr;
/*
 * there shouldn't be any arguments
 */
if (argc != 0)
{
/*
 * read characters until EOF
 */
do
{
/*
 * if the character is newline or EOF, send the line
 * to the REXX queue
 */
if ((ch=getchar())=='\n' || (ch==EOF && (srpt - sr)!=0))
{
/*
 * if adding to the queue fails, exit with the
 * irxstk rc
 */
if ((rqrc = irxstk("QUEUE",
sr,
srpt - sr,
NULL)) != 0)
{
(void) printf("rxqueue - irxstk failed,
rc = %d\n", rqrc);
break;
}
}
}
}
}
}

```



```

        }
    /*
    * reset the buffer pointer and remaining size
    */
        else
        {
            srpt = sr;
            srle = srsz;
        }
    /*
    * if it's a normal character ...
    */
        else
        {
    /*
    * if there isn't room in the buffer ...
    */
            if (srle == 0)
            {
    /*
    * new buffer
    */
                char *nwsr;
    /*
    * new buffer size
    */
                unsigned nwsz;
    /*
    * if creating the new buffer fails, exit rc = 20
    */
                if ((nwsr=malloc(nwsz=srsz+(srle=RXQUBFSZ)))==NULL)
                {
                    (void) printf("rxqueue-out of memory, rc=20\n");
                    rqrc = 20;
                    break;
                }
    /*
    * copy the old buffer into the new buffer
    */
                (void) memcpy(nwsr,
                               sr,
                               srsz);
    /*
    * free the old buffer
    */
                free(sr);
    /*
    * set the new buffer pointers
    */
                srpt = (sr = nwsr) + srsz;
    /*
    * set the new buffer size
    */
                srsz = nwsz;
            }
    /*
    * put the character in the buffer
    */
            -srle;
            *srpt++ = (char) ch;
        }
    }
    while (ch != EOF);

```

```
    }
    else
        (void) printf("too many arguments to RXQUEUE\n");
/*
 * free the buffer
 */
free(sr);
/*
 * return with the rxqueue return code
 */
return rqrc;
}
```

Appendix A: Message Summary

This appendix lists all the messages that may be generated by uni-REXX. Each message is followed by a brief description of its meaning.

3 Program is unreadable

uni-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 specified by the environment variable REXXPATH.

4 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, uni-REXX immediately terminates execution when an interrupt occurs.

5 Machine resources exhausted

The uni-REXX program was not able to obtain the system resources required to continue execution of this program. This may indicate insufficient memory, swap space, or other system resources.

6 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 “*/” may be reported at the end of the program rather than on the line where the opening “/*” appears. Unmatched quotes may be reported at the end of the line on which the opening quote appears.

7 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 may occur if the OTHERWISE clause has been omitted and none of the WHEN conditions is satisfied. It may also occur if a list of instructions follows a WHEN without the necessary DO and END.

8 Unexpected THEN or ELSE

A THEN or an ELSE was encountered in the program for which a matching IF or WHEN is not found. This may occur if the instruction following THEN is DO and its matching END is omitted.

9 Unexpected WHEN or OTHERWISE

A WHEN or OTHERWISE keyword was encountered outside the scope of a SELECT construct. This may 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 may also occur if an attempt is made to branch to the WHEN or OTHERWISE clause using SIGNAL.

10 Unexpected or unmatched END

An END was encountered in the program for which a matching DO or SELECT is not found. This may occur if the END is badly located so that it does not match the DO or SELECT for which it was intended. It may also 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 for avoiding and/or identifying this type of error.

This error may 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 may occur with deeply nested DO-END or IF-THEN-ELSE constructs. It may 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 may 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 and/or identifying this type of error.

15 **Invalid hexadecimal constant**

Hexadecimal constants may contain only the digits 0-9 and the letters a-f and A-F. They may not have leading or trailing blanks, and embedded blanks may occur only at byte boundaries (between pairs of hexadecimal digits).

Binary strings may contain only the digits 0 and 1. They may not have leading or trailing blanks, and embedded blanks may 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 is invoked. This error may 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 which 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 *Chapter 4, Instructions* for a list of valid TRACE settings.

25 **Invalid sub-keyword found**

A unexpected token was in the position where an instruction expected a specific keyword. This may occur if the token following NUMERIC is not DIGITS, FORM, or FUZZ. It may 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 may 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 encountered unexpectedly during execution. Either no loop is active or the control variable name specified on the instruction does not match that of any active loop. This may occur when attempt-

ing 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 may 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 may 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 may 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 may occur if no symbol follows a left parenthesis or if one of the parentheses 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 may also occur if the routine is not compatible with the uni-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 im-

plementation. This may occur in 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 may occur as 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 should immediately follow the variable name is missing.

48 Failure in system service

An operating system service called by uni-REXX resulted in an error. Execution of the program therefore terminated.

49 Interpretation error

A uni-REXX internal error occurred during execution of the program. Please contact The Workstation Group for assistance.

100 I/O Error with intermediate object code file

An I/O error has occurred when trying to execute a program created by the rxc command. The intermediate code version may have been

corrupted in some way. Regenerate the intermediate code version from the source program.

101 Only one input file may be specified

It is not valid to specify more than one program name on the rxx or rxc command.

102 Cannot find rxx for implicit execution

The rxc command requires access to the rxx binary to build the proper implicit execution string in its output file. This error occurs when rxc and rxx are not in the same directory and when rxx cannot be found using the PATH environment variable. It is good practice to install rxx and rxc in the same directory.

103 Invalid command switch

The command switch specified with the rxx or rxc command is not supported. The only valid command switches are

for rxx -c, -v

for rxc -m, -G, -L, -I, -o

104 Program intermediate code version error - recompile program

The version of the uni-REXX interpreter (rxx) that you are currently using does not match the version of the compiler (rxc) used to create the intermediate code version of your program. Because of the possibility of changes to internal data structures in uni-REXX, it is necessary that the version of the interpreter used with intermediate code at run time match the version of the compiler used to generate the intermediate code. Recompile your source program with the version of rxc that you now have installed.

105 Environment variable REXXLIB not set

When you use rxc -m to create an executable binary of a uni-REXX program or rxc -G to rebuild the interpreter, the compiler must have ac-

cess to the uni-REXX archive library, `librx.a`. The environment variable `REXXLIB` must specify the directory where `librx.a` is located. `REXXLIB` is not defined.

106 Cannot find REXX library

When you use `rxcc -m` to create a binary executable of a uni-REXX program or `rxcc -G` to rebuild the interpreter, the compiler must have access to the uni-REXX archive library, `librx.a`. The environment variable `REXXLIB` is set, but the compiler cannot find `librx.a` in that location.

107 Output file would overwrite input file

When you use `rxcc` or `rxcc -m` to compile a source program, the source must have a different name from the output. Typically, the source program has some extension to the filename, such as `“.rex”`. The compiler creates an output file of the same name as the source file but without the extension. This error occurs if you attempt to compile a source program that does not have an extension to the filename.

Appendix B: Common Pitfalls in uni-REXX Programs

This appendix offers assistance in avoiding common pitfalls in uni-REXX programs. The more common programming mistakes are identified, and the correct uni-REXX usage is shown. If you are new to the REXX language, you may want to review these hints in preparation for writing your uni-REXX programs.

Invoking a Built-in Function Like an Instruction

When a built-in function call is the only clause on a line, as in

```
LINEOUT('myfile', 'new data')
```

the function returns a value. This value is then passed to the external environment where it is interpreted as a command. This results in an “Invalid command” message from the operating system. To avoid this, use `CALL` to invoke the function, as in

```
CALL LINEOUT 'myfile', 'new data'
```

or use the built-in function as the expression on the right hand side of an assignment clause, as in

```
x = LINEOUT('myfile', 'new data')
```

Failure to Use Commas with CALL and PARSE ARG

When you CALL a routine or function, the arguments of the CALLED routine must be separated by commas:

```
CALL SUB X, Y
```

passes two arguments to the routine SUB. But,

```
CALL SUB X Y
```

passes one argument to SUB. This argument is the result of concatenating X with Y.

Commas must also be used between arguments in the template of a PARSE ARG instruction.

```
PARSE ARG a1, a2
```

assigns all of the first argument to a1 and all of the second argument to a2.

```
PARSE ARG a1 a2
```

assigns the first word of the first argument to a1 and the rest of the first argument to 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)
```

Conversion of UNIX Commands or Filenames to Uppercase

REXX automatically converts to uppercase the value of an unassigned symbol. To preserve case sensitivity, you must enclose the case-sensitive string in quotes. Since UNIX commands are generally entered in lowercase, be sure to enclose UNIX commands in your uni-REXX programs in quotes.

Also, UNIX filenames are case-sensitive. A file named “ABC” is different from a file named “abc” or “Abc”. When you use filenames as arguments of functions or operands of instructions, you must use quotes if you intend to reference other than an uppercase filename. For example:

```
var = LINEOUT(acctfile, 'accounting data')
```

causes output to be written to the file “ACCTFILE”, which is different from the file “acctfile”.

Using “cd” Instead of CHDIR()

To change directories, use the uni-REXX function CHDIR instead of the UNIX command “cd”. When uni-REXX executes UNIX commands, it spawns a new process for execution of the command and releases that process on command completion. Since the current directory is a separate attribute for each process, a “cd” command affects only the spawned process. The current working directory for the uni-REXX program’s process is unchanged.

Failure to Enclose Command Arguments Within Quotes

Consider the example of attempting to set the REXXPATH environment variable within the UNIX environment:

```
dir = '/home/user1'  
putenv(REXXPATH=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

```
dir = /home/user1  
rc = putenv('REXXPATH='dir)
```

Similar pitfalls exist in the use of host commands that include strings which might be interpreted as operators. In uni-XEDIT macros, for example, the EXTRACT command requires the use of the “/” character as in

```
extract /curline
```

If this command is not enclosed in quotes, uni-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 uni-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) may leave the file in an open state. It may therefore be necessary to close the file with CHAROUT, LINEOUT, STREAM, or EXECIO (FINIs before subsequent attempts to read from or write to the file.

“Incorrect” Procedure Syntax

Both *The REXX Language* and ANSI standard X3.274:1996 state that the PROCEDURE instruction, if present, must be the first instruction executed after sub-routine initialization. *The REXX Language* explicitly states that this instruction “must be the first instruction following the label”.

Some implementations of REXX allow what appears to be an incorrect internal procedure syntax to operate properly. The following example using INTERPRET to access a stem works on the mainframe but does **not** work in uni-REXX

```
label:
  arg stem .
  interpret 'procedure expose' stem'.'
```

The following example does the same thing and works both in uni-REXX and on the mainframe:

```
label:  
  interpret 'procedure expose' arg(1)'.'
```

Note that both of these examples imply that the INTERPRET instruction is really not an instruction between the label and the PROCEDURE instruction. The supposition is that the INTERPRET instruction is not actually executed but that only its operand(s) are.

Appendix C: Bibliography

This appendix contains a listing of additional reference books on the REXX language. It is not intended to be complete but merely to offer suggestions for additional reading.

The REXX Language, A Practical Approach to Programming, M. F. Cowlishaw, Prentice Hall, Second Edition, 1990.

Modern Programming using REXX, Bob O'Hara and Dave Gomberg, Prentice Hall, 1985, 1988. (Now available from REXXPress, REXXPress@wcf.com)

REXX in the TSO Environment, Gabriel F. Gargiulo, QED Information Systems, Inc., Revised Edition, 1993.

Practical Usage of REXX, Anthony Rudd, Ellis Horwood Limited, 1990.

Using ARexx on the Amiga, Chris Zamara and Nick Sullivan, Abacus, 1991.

Amiga Programmers Guide to AREXX, Eric Giguere, Commodore-Amiga, Inc., 1991.

REXX Handbook, edited by Gabe Goldberg and Phil Smith, McGraw-Hill, Inc., 1991.

The AREXX Cookbook, Merrill Callaway, Whitestone, 1992.

Programming in REXX, Charles Daney, McGraw-Hill, Inc., 1992.

REXX Tools and Techniques, Barry K. Nirmal, QED Publishing Group, 1993.

REXX: Advanced Techniques for Programmers, Peter Kiesel, McGraw-Hill, Inc., 1993.

OS/2 2.1 REXX Handbook - Basics, Applications, and Tips, Hallett German, Van Nostrand Reinhold, 1993.

Application Development Using OS/2 REXX, Anthony Rudd, Wiley-QED, 1994.

Mastering OS/2 REXX, Gabriel F. Gargiulo, Wiley-QED, 1994.

Teach Yourself REXX in 21 Days, William F. Schindler & Esther Schindler, SAMS, 1994.

Writing OS/2 REXX Programs, Ronny Richardson, McGraw-Hill, 1994

REXX Procedursprak—hur du programmerar din PC med OS/2, Bengt Kynning, Studentlitteratur (Sweden), 1994.

The REXX Cookbook, Merrill Callaway, Whitestone, 1995.

Object REXX by Example, Gwen L. Veneskey, Will Trosky, and John J. Urbaniak, Ph.D., Aviar, Inc., 1996.

Programming Language REXX, Document Number X3.274:1996, American National Standards Institute, 1996.

The following papers, published in technical journals, also provide reference material on the REXX language.

The Design of the REXX Language, M. F. Cowlshaw, **IBM Systems Journal**, Vol. 23, No. 4, 1984.

REXX on TSO/E, G. E. Hoernes, **IBM Systems Journal**, Vol. 28, No. 2, 1989.

Partial Compilation of REXX, R. Y. Pinter, P. Vortman, and Z. Weiss, **IBM Systems Journal**, Vol. 30, No. 3, 1991.

The programming language standards scene, ten years on, Paper 20: REXX, Brian Marks, **Computer Standards & Interfaces** 16, 1994 (Elsevier).

The Early History of REXX, Mike Cowlshaw, **IEEE Annals of the History of Computing** (ISSN 1058-6180) Vol. 16, No. 4, Winter 1994.

In addition to these references, published proceedings of the annual REXX Symposium are available from Stanford Linear Accelerator Center.

Appendix D: System Limitations

There are very few implementation-specific limitations in uni-REXX. Those that exist are documented in this appendix.

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

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 therefore likely that your system memory will be exceeded before you approach these limits.

Appendix E: uni-REXX Environment Variables

This appendix lists all of the environment variables used by uni-REXX. Each variable is documented in context in the regular chapters of this manual.

The commands used to set an environment variable are shell-specific. Choose the appropriate commands from the table below:

Shell	Command
C	setenv <i>var-name value</i>
Bourne or Korn	<i>var-name=value</i> export <i>var-name</i>

REXXCOMMANDNOTQUOTED

For backward compatibility with early versions of uni-REXX, setting this variable to “1” causes commands sent to an external command environment to execute as if they were not enclosed by quotes.

REXXENVPREFIX

the prefix used for all other uni-REXX environment variables

Default: REXX

Set this environment variable to change the initial characters in the names of all other uni-REXX environment variables to something other than `REXX`.

The setting of REXXENVPREFIX and its use in the names of other environment variables is case sensitive. This environment variable would most likely be used by an application that embeds uni-REXX as a scripting language to easily identify configuration settings for scripts within that application.

Examples:

REXXENVPREFIX set to “ABC”

uni-REXX searches the path specified by the environment variable “ABCPATH” to locate external programs

REXXENVPREFIX set to “Inv_Prog_”

uni-REXX searches the path specified by the environment variable “Inv_Prog_PATH” to locate external programs

REXXEXECIOCASELESS

For backward compatibility, if set to ‘1’ the Locate and Avoid options to EXECIO will no longer honor case in text strings.

REXXEXTERNALUPPERCASEFILENAME

If set to ‘1’, external filenames are searched for as uppercase names, rather than lowercase, for systems with case-sensitive filenames.

REXXIOPERSISTENTNAMEDFIFOS

when defined as ‘1’, allows a REXX program acting as a server on a FIFO to accept input from multiple clients without getting an EOF indication, and raising the NOTREADY condition.

REXXLIB

when using rxc -m or rxc -G, the location of the uni-REXX libraries and header files (librx.a, librx.i.a, irx.h, irxproto.h, irxsaa0.h, rexxsaa.h)

Default: none; the location where these files are installed is site-specific

REXXLOADPATH

If defined, dynamic loading will use this path instead of the default search path. This is intended to improve portability between the various UNIX systems.

REXXMODULECC

the command or full path name to execute the C compiler on your system; used only when generating platform-specific binaries with the uni-REXX Developer's Kit

Default: cc

REXXMODULECFLAGS

the C compiler flags required on your system; used only when generating platform-specific binaries with the uni-REXX Developer's Kit

Default: platform-specific

REXXMODULECOMMANDINTERPRETER

If specified, rxc will use this value as the command interpreter and arguments to insert as the first line of compiled output.

Default: #!/usr/local/bin/rxx, or the actual install location

REXXMODULEEXTRALIBS

For rxc -m, allows additional archive libraries to be specified when linking a standalone executable module. Applies to uni-REXX Developer's Kit only.

REXXMODULELD_FLAGS

the loader flags required to generate a binary on your system; used only when generating platform-specific binaries with the uni-REXX Developer's Kit

Default: platform-specific

At a minimum, you will require access to the math library and the socket library. Additional libraries may also be required.

REXXMODULELDSYMBOLS

indicates if loader symbol tables should be removed (stripped) from a binary; used only when generating platform-specific binaries with the uni-REXX Developer's Kit

Default: module is stripped

Set this environment variable to any value to disable stripping of the binary.

REXXNUMERICDIGITS

When this variable is set to a numeric value, it establishes the default value for the NUMERIC DIGITS setting within all uni-REXX programs.

REXXPATH

the path searched for external uni-REXX programs (functions or subroutines) called by a program

Default: current working directory

Set this environment variable in the same way you would set PATH to specify all the directories to be searched for your external programs.

REXXPARSESOURCEACTIVEENVIRONMENT

For backward compatibility, when set to '1', the fifth word of output from the PARSE SOURCE instruction will be the active external command environment, instead of the default environment when the program was started.

REXXSTACKSHARED

enables sharing of the uni-REXX program stack (external data queue) between programs running in separate processes

Default: stack sharing disabled

Set this environment variable to any value to enable sharing of the program stack. The default behavior is a change from previous releases of the product.

REXXSTREAMCOMMANDNOCOLON

For compatibility with older releases of uni-REXX, setting the variable to '1' will cause the stream function to delete the trailing ":" character on its "READY:" response string.

REXXSUFFIXSOURCE

the file extension (suffix) used to locate a uni-REXX program for execution or compilation

Default: rex

Set this environment variable to the extension that identifies all uni-REXX programs. Both the interpreter (rxx) and the compiler (rxc) use this setting to locate source programs.

REXXTEMP

the directory where uni-REXX temporary files are placed; these may be disk files or special files created by sockets for interprocess communications

Default: /tmp

Set this environment variable to the full directory path to be used for all uni-REXX temporaries.

REXXTEMPDISK

the directory where uni-REXX temporary disk files are placed; such temporary files are normally created only when generating a platform-specific binary using the uni-REXX Developer's Kit

Default: current setting of REXXTEMP, or its default if it is unset

Set this environment to the directory in which you wish disk files to be placed. This variable does not affect the location for temporaries associated with interprocess communications.

REXXTEMPIPC

the directory for uni-REXX temporary files associated with interprocess communications

Default: current setting of REXXTEMP, or its default if it is unset

Set this environment to the directory in which you want special IPC files to be placed. This variable does not affect the location for disk files.

REXXTRACE

when set to any of the TRACE instruction options, provides a default TRACE setting for all uni-REXX programs.

REXXTRACELOGFILE

When set to the name of a disk file, uni-REXX writes extensive internal debugging trace output to the file. For TWG Technical Support use only, not intended for end users.

!	<ul style="list-style-type: none"> _ACCEPT function225 _BIND function227 _CLOSEDIR function.....229 _CLOSESOCKET function.....229 _CONNECT function.....230 _ERRNO function232 _EXIT function233 _FD_CLR function.....233 _FD_ISSET function234 _FD_SET function234 _FD_ZERO function234 _FORK function234 _GETEUID function235 _GETHOSTBYADDR function236 _GETHOSTBYNAME function.....236 _GETHOSTID238 _GETHOSTNAME238 _GETPEERNAME function239 _GETPID function240 _GETPPID function241 _GETSERVBYNAME function241 _GETSOCKNAME function.....242 _GETSOCKOPT function.....243 _GETUID function.....244 _IOCTL function245 _KILL function.....246 _LISTEN function.....247 _OPENDIR function248 _REaddir function.....249
---	--

_RECV function	249
_REGEX function	251
_SELECT function	252
_SEND function.....	253
_SETSID function	254
_SETSOCKOPT function.....	255
_SLEEP function	257
_SOCKET function	257
_STAT function	258
_SYS_ERRLIST function	260
_SYSTEMDIR function	260
_TRUNCATE function	261
_UMASK function	261
_WAIT function.....	262
_WAITPID function.....	263

A

ABBREV function.....	114
Abbreviations.....	114
ABS function.....	115
Absolute value	115
Abuttal concatenation	12
ADDRESS built-in function	42
ADDRESS function	115
ADDRESS instruction.....	15, 35, 39, 42
command I/O redirection	39-41
Altering the flow of a DO loop	67
AND operation.....	118
API	
SEE Application Programming Interfaces	
Application Programming Interfaces	291
ARGLIST structure.....	335
building the application.....	319
control blocks	335
EVALBLOCK structure	336
EXECBLK structure	336
executing a uni-REXX program.....	291, 297, 305
EXITBLK structure.....	337
FPCKDIR structure.....	339
host command environments	291, 315
INSTBLK structure.....	341
IRXEXCOM	292-293

IRXEXEC	291, 297
IRXEXITS	292, 300
IRXJCL	291, 305
IRXSTK	292, 308
IRXSTOP	292, 312
IRXSUBCM	315
IRXSUBCOM	291
IRXSUBCT structure	343
sharing variables	292-293
SHVBLOCK structure	344
STMT structure	347
terminating a uni-REXX program.....	292, 312
uni-REXX archive libraries	320
uni-REXX header file	320
uni-REXX program stack.....	292, 308
user exits	292, 300
ARG function	117
ARG instruction	44
ARGLIST structure (control block).....	335
Argument strings	44, 74, 117
Arithmetic operators	11
Arithmetic precision	70
determining current setting	144

B

B2X function	121
Bibliography	371
Binary string	6
Binary to hexadecimal conversion.....	121
BITAND function	118
BITOR function	119
BITXOR function	120
Blank concatenation	12
blanks	5
Bourne shell	35, 42
Built-in functions	46, 111
ABBREV	114
ABS	115
ADDRESS.....	42, 115
ARG	117
B2X	121

BITAND	118
BITOR	119
BITXOR	120
C2D	135
C2X	135
CENTER.....	122
CHARIN.....	16, 124
CHAROUT	16, 126
CHARS	16, 128
CHDIR.....	18, 129
COMPARE.....	130
CONDITION	131
COPIES	133
CUSERID.....	18, 134
D2C	145
D2X	146
DATATYPE.....	136
DATE.....	138
DELSTR	142
DELWORD	143
DIGITS	144
ERRORTXT	146
FIND	147
FORM	149
FORMAT.....	150
FUZZ.....	154
GETCWD.....	18, 155
GETENV	18, 156
INDEX.....	157
INSERT	158
JUSTIFY	159
LASTPOS.....	159
LEFT	160
LENGTH	162
LINEIN.....	16, 163
LINEOUT	16, 165
LINES	16, 167
list of.....	112
LOWER	168
MAX	169

MIN	170
OVERLAY	170
POPEN	18, 36, 172
POS	173
PUTENV	18, 174
QUALIFY	175
QUEUED.....	17, 176
RANDOM	177
REVERSE	178
RIGHT	178
SIGN	179
SOURCELINE.....	180
SPACE	181
STREAM.....	17, 183
STRIP	185
SUBSTR	186
SUBWORD	187
SYMBOL.....	188
TIME.....	190
TRACE	193
TRANSLATE	194
TRUNC	196
UPPER	197
USERID	197
VALUE.....	198
VERIFY	200
WORD	202
WORDINDEX	203
WORDLENGTH	204
WORDPOS	205
WORDS	206
X2B	207
X2C	208
X2D	209
XRANGE.....	206
Built-in functions	13

C

C compiler	379
C shell	35, 42
C2D function.....	135

C2X function.....	135
CALL instruction.....	46
CENTER function	122
Changing directories	18, 129
Character to decimal conversion.....	135
Character to hexadecimal conversion.....	135
CHARIN function	16, 124
CHAROUT function.....	16, 126
CHARS function	16, 128
CHDIR function	18, 129
Clauses	5-12
continuation	8
instruction.....	7
label.....	8
null clause	8
Clearing stack buffers.....	275
Client/Server Sample Application	265
Closing a directory	229
Closing a file.....	126, 165, 184, 278
cms host command environment	42
command host command environment.....	35, 42
Comment.....	8
Comparative operators	11-12
normal	12
strict	12
COMPARE function	130
Comparing strings	130
Compiler.....	21, 27-28
Compound symbol	9
Concatenation	12
abuttal	12
blank.....	12
Concatenation operators	12
CONDITION function	131
Condition traps	14-15, 48, 99, 131
Conditional processing	54, 63, 96
Conditions.....	14-15
ERROR	15
FAILURE	15
HALT	15
information about current trapped.....	131

LOSTDIGITS.....	15
NOTREADY	15
NOVALUE.....	15
SYNTAX	15
Configuration management functions	222
Constant symbol.....	8
Constants.....	8
Continuation	8
Continuation character.....	8
Control variables	53
Conversion	
binary to hexadecimal.....	121
character to decimal	135
character to hexadecimal.....	135
decimal to character	145
decimal to hexadecimal	146
hexadecimal to binary.....	207
hexadecimal to character.....	208
hexadecimal to decimal	209
to lowercase	168, 194
to uppercase.....	110, 194, 197
Converting to lowercase	168, 194
Converting to uppercase	110, 194, 197
COPIES function	133
Creating a child process	234
Creating stack buffers.....	289
csh host command environment.....	35, 42
Current date.....	138
Current directory.....	18, 155
CUSERID function	18, 134

D

D2C function.....	145
D2X function.....	146
Data type.....	136
DATATYPE function	136
Date formats.....	138-139
DATE function.....	138
Debugging a program.....	103-104, 365
Decimal to character conversion.....	145
Decimal to hexadecimal conversion.....	146
Deleting a file.....	184

Deleting characters	142
Deleting words	143
DELSTR function	142
DELWORD function	143
DESBUF command	19, 274-275
Developer's Kit	21, 27-28, 30
Diagnostic messages	353
DIGITS function	144
DO instruction	53
DO loop	53
altering the flow	67
control variable	53
leaving	68
Documentation conventions	2-4
DROP instruction	59
DROPBUF command	19, 274, 276
Dropping stack buffers	276
Dummy instruction	69
Duplicating strings	133

E

Effective userid	235
Embedded applications	291
SEE ALSO Application Programming Interfaces	
building	319
End of file	15, 49, 100
Environment variables	18, 377
current setting	18, 156
defining	18, 32, 174
PATH	24
REXXCOMMANDNOTQUOTED	377
REXXENVPREFIX	377-378
REXXEXECIOCASELESS	378
REXXEXTERNALUPPERCASEFILENAME	378
REXXIOPERSISTENTNAMEDFIFOS	378
REXXLIB	31, 321, 378
REXXLOADPATH	379
REXXMODULECC	30, 379
REXXMODULECFLAGS	31, 379
REXXMODULECOMMANDINTERPRETER	379
REXXMODULEEXTRALIBS	379
REXXMODULEELDFLAGS	31, 379

REXXMODULELDSYMBOLS	31, 380
REXXNUMERICDIGITS	380
REXXPARSESOURCEACTIVEENVIRONMENT ...	380
REXXPATH	22, 25, 33, 380
REXXSTACKSHARED	381
REXXSTREAMCOMMANDNOCOLON	381
REXXSUFFIXSOURCE	24-25, 29-30, 381
REXXTEMP	30, 381
REXXTEMPDISK	382
REXXTEMPIPC	382
REXXTRACE	382
REXXTRACELOGFILE	382
setting	31-32, 377
ERROR condition	15
Error messages	353
ERRORTXT function	146
EVALBLOCK structure (control block)	336
Exclusive OR operation	120
EXECBLK structure (control block)	336
EXECIO command	274, 277
Executable binary	28, 30-31
C compiler	30-31
loader symbol tables	380
temporary space	30
uni-REXX libraries	30-31
unresolved references	379
Executing dynamically-created instructions	66
Executing uni-REXX programs	
SEE Program execution	
Executing Unix commands	18
EXIT instruction	61
EXITBLK structure (control block)	337
Exits	292, 300
command	301
initialization	301
terminal input	301
terminal output	301
termination	301
Explicit execution	22
Exponential notation	70-71
determining current setting	149

Exposing variables to a subroutine	47, 85
Expression	10-12
Extensions.....	2, 18, 34
External data queue	
SEE uni-REXX program stack	
External function packages	323
ARGLIST structure	324, 335
control blocks	323
EVALBLOCK structure.....	324, 336
FPCKDIR structure	323, 339
in embedded applications	328
in stand-alone uni-REXX programs	330
External functions	13, 33
External subroutines	33

F

FAILURE condition	15
File and directory management functions.....	223
File permissions	22-23
File status.....	258
FIND function	147
Finding a string	147, 157, 159, 173, 202-203, 205
FORM function.....	149
FORMAT function	150
Formatting numeric output	150
Formatting string output.....	159-160, 178, 181, 194
FPCKDIR structure (control block).....	339
Function calls	11
Functions	13
built-in.....	13, 46, 111
configuration management	222
external.....	13, 33, 46
file and directory management	223
general rules.....	113
internal.....	13, 46
interprocess communication.....	223
invocation.....	111
locating	33
process management	222
regular expression processing	223
system error processing	223
Unix-specific	20, 222

	user-written	291, 323
	Fuzz factor	71
	determining current setting	154
	FUZZ function	154
G	GETCWD function	18, 155
	GETENV function	18, 156
	GLOBALV command	19, 274, 284
H	HALT condition	15
	Hexadecimal string	6
	Hexadecimal to binary conversion	207
	Hexadecimal to character conversion	208
	Hexadecimal to decimal conversion	209
	Host command environments	35, 39, 42
	command	35, 42
	csh	35, 42
	default	35
	defining	291, 315
	determining default	75
	determining the current setting	115
	ksh	42
	sh	35, 42
	UNIX	35
	Host commands	34
	environments	35, 39
	execution	34, 36, 39, 172
	I/O redirection	39-41
	Host configuration	236, 238
I	I/O	
	SEE Input/Output	
	I/O redirection	15
	IF instruction	63
	IF-THEN-ELSE	63
	Implementation name	
	determining	76
	Implementation release date	
	determining	76
	Implicit execution	23

Implicit execution string	23
INDEX function.....	157
Informational messages	353
Input/Output	15-17
commands	17-18, 274, 277
functions	15-16, 124, 126, 163, 183
instructions	17, 74, 89, 95
redirection	39-41
streams	17
INSERT function	158
Inserting a string.....	158
INSTBLK structure (control block).....	341
Instruction.....	7
assignment	7
command.....	7
keyword	7
Instructions	37
ADDRESS	35, 39
ARG	44
CALL	46
DO	53
DROP	59
EXIT	61
IF.....	63
INTERPRET	66
ITERATE.....	67
LEAVE	68
list of	38
NOP	63, 69, 97
NUMERIC	70
OPTIONS	73
PARSE	16, 74
PROCEDURE	85
PULL.....	16, 89
PUSH	16, 91
QUEUE.....	17, 92
RETURN	93
SAY.....	17, 95
SELECT	96
SIGNAL.....	99

	TRACE	103
	UPPER	110
	Interactive tracing	104
	Intermediate code generation.....	27
	Intermediate code version	29
	Internal functions	13
	INTERPRET instruction.....	66
	Interprocess communication.....	34, 223, 348
	Introduction	1
	irx.h	320
	irxproto.h	320
	IRXSUBCT structure (control block).....	343
	ITERATE instruction	67
J	JUSTIFY function	159
K	Korn shell	35, 42
	ksh host command environment	42
L	Label	8
	Language definition	1
	Language extensions	2, 18, 34
	Language features.....	5
	clauses.....	5-12
	expressions	10-12
	functions	13
	input/output.....	15
	operators	11
	parsing	17
	special variables	14
	symbols	8-10
	LASTPOS function	159
	LEAVE instruction.....	68
	Leaving a DO loop.....	68
	Leaving a program	61, 93
	LEFT function.....	160
	LENGTH function.....	162
	Length of a string	162, 204
	librx.a	320
	librxi.a.....	348

	Limitation	375
	LINEIN function	16, 163
	LINEOUT function	16, 165
	LINES function	16, 167
	Literal string	6, 10
	Locating a uni-REXX program	22, 24
	Locating a uni-REXX program	24
	Logical operators	13
	LOSTDIGITS condition	15
	LOWER function	168
M	MAKEBUF command	19, 274, 289
	MAX function	169
	Message summary	353
	Message text	
	retrieving	146
	MIN function	170
N	NOP instruction	63, 69, 97
	Normal comparison	12
	NOTREADY condition	15, 49, 100
	NOVALUE condition	15
	Null clause	8
	Numeric comparisons	71
	NUMERIC instruction	70
O	Opening a directory	248
	Opening a file	184
	Operator	7
	Operators	11
	arithmetic	11
	comparative	11-12
	concatenation	12
	logical	13
	OPTIONS instruction	73
	OR operation	119
	ORXXVersionCheck() C language function	320
	Output of Unix commands	18
	OVERLAY function	170

P

Parent process id.....	241
PARSE instruction.....	16, 74
PARSE LINEIN	16
PARSE PULL.....	16
PARSE SOURCE.....	75
PARSE VERSION.....	76
Parsing	17, 74-83
by pattern	78
by position	79
by words.....	77
placeholder.....	82
template	74, 76-82
PATH environment variable	24
Pausing a program	257
Persistent I/O streams	17
Pipes	17
reading from	124, 163
writing to.....	126, 165
Pitfalls in uni-REXX programs	365
POPEN function	18, 36, 172
POS function	173
Precision	
determining current setting	144
Precision of numbers.....	70
PROCEDURE instruction	85
Process group leader	254
Process id.....	240-241
Process management functions	222
Program execution.....	22
explicit.....	22
file permissions.....	22-23
implicit.....	23
locating the program	22, 24-25
Program invocation	
determining method	75
Program name	
determining	75
Program names.....	21, 24, 29
Program pathname	
determining	75
Program stack	

	SEE uni-REXX program stack	
	Protecting variables	47, 85
	PULL instruction.....	16, 89
	PUSH instruction	16, 91
	PUTENV function.....	18, 174
Q	QUALIFY function	175
	QUEUE instruction.....	17, 92
	QUEUED function.....	17, 176
R	RANDOM function	177
	Random numbers	177
	RC special variable.....	14
	Reading a file	124, 163, 277
	Reading directory entries	249
	Real userid	244
	Regenerating the interpreter	28
	Regular expressions	223, 251
	Removing blanks	185
	RESULT special variable	14, 46, 93, 111
	RETURN instruction.....	93
	Returning a value	61, 93
	REVERSE function.....	178
	REXX language definition	1
	REXX language level	
	determining	76
	REXXCOMMANDNOTQUOTED environment variable	377
	REXXENVPREFIX environment variable	377-378
	REXXEXECIOCASELESS environment variable	378
	REXXEXTERNALUPPERCASEFILENAME environment variable	378
	REXXIOPERSISTENTNAMEDFIFOS environment variable	378
	REXXLIB environment variable	31, 321, 378
	REXXLOADPATH environment variable.....	379
	REXXMODULECC environment variable	30, 379
	REXXMODULECFLAGS environment variable	31, 379
	REXXMODULEEXTRALIBS environment variable.....	379
	REXXMODULEELDFLAGS environment variable....	31, 379
	REXXMODULEELDSYMBOLS environment variable	31, 380
	REXXNUMERICDIGITS environment variable.....	380

REXXPARSESOURCEACTIVEENVIRONMENT environment variable	380
REXXPATH environment variable	22, 25, 33, 380
REXXSTACKSHARED environment variable	33, 381
REXXSTREAMCOMMANDNOCOLON environment variable	381
REXXSUFFIXSOURCE environment variable	24-25, 29-30, 381
REXXTEMP environment variable	30, 381
REXXTEMPDISK environment variable	382
REXXTEMPIPC environment variable	382
REXXTRACE environment variable	382
REXXTRACELOGFILE environment variable	382
RIGHT function	178
rx command	28-29, 32
RXFUNCCADD function	212
RXFUNCDROP function	212
RXFUNCQUERY function	212
RXQUEUE command	290
rx command	22, 24-25
RXXCOMMANDKILL function	213
RXXCOMMANDSPAWN function	213
RXXCOMMANDWAIT function	213
RXXOSENDOFLINESTRING function	214
RXXOSENVIROMENTSEPARATOR function	214
RXXOSPATHSEPARATOR function	214
RXXSLEEP function	214

S

SAY instruction	17, 95
SELECT instruction	96
SENTRIES command	19, 290
Server configuration	241
Setting environment variables	377
sh host command environment	35, 42
Sharing variables	19, 274, 284, 292-293
SHVBLOCK structure (control block)	344
SIGL special variable	14, 99
SIGN function	179
SIGNAL instruction	99
Simple symbol	9
Socket operations	225, 227, 229-230, 247, 249, 253, 257

Source code security	27
SOURCELINE function	180
SPACE function.....	181
Spawning a process	234
Special characters	7
Special variables	14
RC	14
RESULT	14, 46, 93, 111
SIGL.....	14, 99
STDIN	17
STDOUT	17
Stem	10
STMT structure (control block)	347
STREAM function	17, 183
Streams	17
Strict comparison.....	12
STRIP function	185
Subroutines	
execution	99
external	33, 46
internal	46
invoking.....	46, 99
locating	33
SUBSTR function.....	186
SUBWORD function.....	187
Symbol	6, 8-10
compound	9
constant	8
simple	9
status of.....	188
value of	198
SYMBOL function	188
SYNTAX condition	15
System error processing	224, 232, 260
System error processing functions	223
System limitations.....	375
System name	
determining	75

T

Template	
SEE Parsing	
Terminal input	16, 74-75, 89, 124, 163
Terminal output.....	16-17, 95, 126, 165
Terminating a process	233, 246
Time formats	190-191
TIME function	190
TRACE function	193
TRACE instruction.....	103
Trace output	105
Trace setting	103-104, 193
Tracing a program	103-104
Transient I/O streams	17
TRANSLATE function	194
TRUNC function	196
Truncating a file	261

U

uni-REXX archive libraries	31
librx.a.....	320
librxi.a	348
uni-REXX header file (irx.h).....	320
uni-REXX program stack	33
access from a C language program	292, 308
adding data.....	91-92
buffers	19, 274-276, 289
host command output.....	172, 290
number of items	290
number of lines	89, 176
retrieving data.....	75, 89
sharing	33
uni-SPF.....	1
Unix commands	
executing.....	18, 39, 172
output.....	18, 172
Unix processes	42
Unix userid	197
effective.....	235
real.....	244
retrieving	134
Unix-specific functions.....	20, 222

_ACCEPT	225
_BIND	227
_CLOSEDIR	229
_CLOSESOCKET	229
_CONNECT	230
_ERRNO	232
_EXIT	233
_FD_CLR	233
_FD_ISSET	234
_FD_SET	234
_FD_ZERO	234
_FORK	234
_GETEUID	235
_GETHOSTBYADDR	236
_GETHOSTBYNAME	236
_GETHOSTID	238
_GETHOSTNAME	238
_GETPEERNAME	239
_GETPID	240
_GETPPID	241
_GETSERVBYNAME	241
_GETSOCKNAME	242
_GETSOCKOPT	243
_GETUID	244
_IOCTL	245
_KILL	246
_LISTEN	247
_OPENDIR	248
_READDIR	249
_RECV	249
_REGEX	251
_SELECT	252
_SEND	253
_SETSID	254
_SETSOCKOPT	255
_SLEEP	257
_SOCKET	257
_STAT	258
_SYS_ERRLIST	260
_TRUNCATE	261

	_UMASK	261
	_WAIT	262
	_WAITPID	263
	list of	222-223
	uni-XEDIT	1
	UPPER function	197
	UPPER instruction	110
	User-written functions	291, 323
	in embedded applications	328
	in stand-alone uni-REXX programs	330
	USERID function	197
V	VALUE function	198
	Variable reference	59, 85
	Variables	
	assigning a value	7
	compound	9
	control	53
	dropping	59
	exposing to a subroutine	47, 85
	names	10
	parsing	75
	protecting	47, 85
	sharing	19, 274, 284, 292-293
	simple	9
	special	14
	stem	10
	variable reference	85
	VERIFY function	200
	Version number	25
W	Waiting for processes	262-263
	Waiting on a process	262-263
	WORD function	202
	WORDINDEX function	203
	WORDLENGTH function	204
	WORDPOS function	205
	WORDS function	206
	Writing to a file	126, 165, 277

X

X2B function	207
X2C function	208
X2D function.....	209
XRANGE function	206