Enterprise REXX for Windows

This help corresponds to the embedded scripting language version of Enterprise REXX 1.6.2.

REXX™ Embedded Scripting Language Help

Introduction

- The REXX Language
- Enterprise REXX Overview

REXX Language Reference

- Instructions as defined in *The REXX Language*.
- Operators as defined in *The REXX Language*.
- Built In Functions as defined in *The REXX Language*.
- Language Extensions provided with Enterprise REXX™.
- Windows Extensions to REXX.

The REXX Language

The REXX language is defined by *The REXX Language - A Practical Approach to Programming* by Michael F. Cowlishaw, commonly referred to as *the **TRL***. These help topics consist of a reference to REXX operators, instructions and functions as defined in the TRL.

Glossary

These are help topics available for the version of Enterprise REXX embedded in your Windows application. For information on how to use this Help, press F1.
For more information on Enterprise Rexx see http://www.WinRexx.com or send mailto:Sales@WinRexx.com.

Enterprise REXX™ for Windows

Enterprise REXX for Windows is a full implementation of the REXX programming language, and the surrounding programs and documentation needed to use REXX productively. There is a Win32 version for Windows NT and Windows 95, and Win16 version for Windows 3.1.

Enterprise REXX supports version 4.0 of the REXX language.

You may wish to look at the following topics:

Uses of Enterprise REXX for Windows

Thousands of people use REXX as the best programming language for "trivial programming tasks". Enterprise REXX (WinREXX) lets you solve those same problems, and even run the same programs, in a Windows application. REXX is better for "anything you programmed in BASIC".

Examples of programs perfect for REXX include:

> With WinREXX you **can** write and run conventional programs under Windows **without** learning GUI programming - conventional REXX programs. Examples include repetitious mathematical calculations you use every day. How many people bring up an entire spread-sheet application just to run a simple formula a few times?
>
> REXX has powerful text parsing functions for text format conversion. An example would be formatting text for use in a mail merge program. Another example might be formatting information, e.g. into standard text comma delimited format, for input to spread sheets and databases.
>
> Enterprise REXX **is** a Windows utility language, like BAT files are for DOS, that allow you to manage your Windows environment. for REXX language extensions make it easy to program tasks such as traversal of disk directory structures, determination of disk free space, as well as copy, rename and deletion of disk files. REXX and DOS programs may be started and **REXX programs can wait** for their completion. For example a REXX program could warn that you are running low on disk free space, delete all *.BAK files and schedule file backups.
>
> WinREXX **is** an ideal command language processor of application macros and scripts. This allows in-house and commercial **software developers** to **concentrate on building applications** (*not* upon writing a language) and thereby to complete projects faster and at less cost.

WinREXX is **not** another *high level* language to build Graphical User Interface (GUI) applications *quickly and easily*. WinREXX is **not** an alternative, e.g. to Microsoft's Visual Basic and Sybase's Power Builder with which you can build GUI applications. With Enterprise REXX you can however use REXX as the programming language *behind the pretty face* you create with those products.

API and OEM Uses

REXX is an ideal macro language for any commercial product or in-house company program.

1. An industry standard language, ***not** another proprietary language*.
2. Ease of learning for new users.
3. Language power needed to solve tough problems.
4. Extendibility with External Functions that you can package in Windows EXEs or DLLs.
5. Adaptability as any unrecognized command (statements) are passed to the current "environment", e.g. to your application.
6. A comprehensive API for all of the above.

If you are interested in embedding REXX in your application, please **look at the API help** (in a separate WinHelp file) and **look at the Enterprise REXX for Windows \API sub-directory** files.

OS/2 Users Note: The Enterprise REXX for Windows Application Programming Interface (API) is compatible with release 1.3 of IBM's OS/2 Operating System. A description of this API is provided in Chapter 9 - Application Programming Interface, of IBM Operating System/2 Procedures Language 2/REXX (S01F-0284).

REXX Language Introduction

REXX is widely regarded for its **ease of learning** *and* **use**, and its user **extendibility**. Having been originally conceived as a user command language, REXX is an **ideal "macro and scripting" language** for applications of all types. Enterprise REXX has been implemented with this use strongly in mind. IBM has standardized REXX as the SAA compliant command language for its mainframe, mid-range and personal computer systems. REXX is the programming language most recently accepted by ANSI.

REXX is a procedural language that allows programs and algorithms to be written in a clear and structured way. REXX was designed for ease of use by computer professionals and by "casual" general users. To be easy to use a language must *easily manipulate* the object types you commonly deal with: names, addresses, text messages, words, numbers, etc. REXX features make manipulation of these object types easy. As a command language REXX is designed to be independent of the environment it operates within (e.g. an application or the operating system), and to have **commands** of that environment **embedded** within REXX programs.

For a full description of the REXX language refer to *The REXX Language - A Practical Approach to Programming* by M.F. Colishaw (ISBN 0-13-779067-8). As a general reference we suggest *The REXX Handbook* by Gabriel Goldberg and Philip H. Smith III, ISBN 0-07-023682-8 published by McGraw-Hill.

If you are an experienced programmer you may be interested in browsing through either the REXX Language Instructions or the REXX language Built in Functions.

REXX is a standard feature of IBM's OS/2 Operating Systems since its 1.3 release and of IBM DOS starting with Release 7.

Automatic Arg Prompting

Automatic arg prompting is a feature of Enterprise REXX provided for the easy execution of existing, e.g., DOS or OS/2, REXX programs which expect command line arguments as their input. Since under Windows one does not normally run "from a command line", an easy alternative for argument input is desirable.

If an 'ARG' or 'PARSE ARG' instruction, or an ARG() built in function, is encountered *on the first execution level* of a REXX program Enterprise REXX will pop up a PromptBox requesting arguments. REXX subroutine functions CALLed in the same source file, or in separate REXX programs in separate disk files, are **not** at the first execution level.

If you do *not* want automatic arg prompting, use the **Run** with **No ARG Prompts** menu command. <Alt+R+N>

REXX object code

The REXX programs you write are in a form called source code which is stored as standard ASCII text.

Enterprise REXX translates the source of REXX programs into a binary form called object code. Object code can be efficiently interpreted and executed, and is saved for later re-execution. If you run a large REXX program, or run on a slower computer, you will see "Compiling" and then "Executing" messages in the REXX source window Status Bar.

Object code is created by Enterprise REXX after any successful REXX program compile or syntax check. If current object code exists when Enterprise REXX saves your program source to a disk file, it will append an ASCII EOF character (decimal 26, hex 1A) and then the object code.

Glossary of Windows Terms

clipboard

Clipboard

The clipboard is a memory area within windows that may be shared by all windows applications.
Any application may place data upon the clipboard. Any other application, that can use that type of data, may read that data from the clipboard. While many data types may be placed upon the clipboard, for example bitmap graphics, the most common type, and the type used by WinREXX,, is simple text.

REXX Glossary

- Arithmetic Operators
- Assignment Operators
- Concatenation Operators
- Environment
- Expression
- Operators
- Template
- Terms
- Variable

Environment Commands

If a REXX program statement is not recognized as a REXX language instruction or function or as an assignment, that statement is passed to the current external environment. The REXX "External Environment" is the program to which these "external commands" are passed. This allows application commands to be embedded within a REXX program, and then passed to, *and processed by* the application program.

The Enterprise REXX For Windows default external environment is the operating system command interpreter (Command.COM for DOS and CMD.EXE for NT). This allows Enterprise REXX for Windows to execute operating system commands and program names "embedded" in your REXX progams. Note however that under Windows+DOS a DOS "machine" is started for each DOS command execution which is resource intensive and therefore relatively slow.

Multiple environments may exist. Use the REXX ADDRESS instruction to select by name the current external environment.

API Technical Note: A REXX Environment name and function address is formally registered by a call to RexxRegisterSubcom() which is defined in the Enterprise REXX for Windows API. The default environment is named in the RexxStart()call that invoked the Enterprise REXX language processor.

Expressions

A REXX expression is the general mechanism to combine one or more items of data in a specified way to produce a result, usually different from the original data. An expression is comprised of terms and operators.

Operators

A REXX operator represents an operation, such as addition, to be performed upon one or two terms. REXX has the following groups of operators

Arithmetic Operators

Assignment Operators

Comparative Operators

Concatenation Operators

Logical Operators

Parsing Templates

REXX uses a **template** to parse (split up) a string and assign its parts to specified variables. The ARG, PARSE and PULL instructions use templates in this manner.

Templates allow a string to be split up by:

- **words** (delimited by blanks).
- explicit matching of strings, called **literal patterns**.
- specific numeric positions, called **positional patterns**.

Examples:

To parse the input string '26 February 1947' **by words** you could use:

```
PULL day month year
```

To parse the input string '2/26/47' by **literal pattern** you could use:

```
PARSE PULL month '/' day '/' year
```

To parse the content of a variable **by position** you could use:

```
astring = '6chars 5more and the rest of the string.'
PARSE VAR astring first6 8 next5 14 rest
```

-or-

```
PARSE VAR astring +7 second5 +6 rest
```

Terms

A REXX term is a literal string, symbol, function call, or sub-expression, which represents a character string used within an expression. Terms are combined using operators.

Refer to the Glossary of the TRL (*The REXX Language*, by M. F. Cowlishaw). A few terms are currently defined herein:

Environment
Expression
Operators
Template
Variable

Variable

REXX variables are named objects which may be assigned a value during the execution of a REXX program. The value of a variable is a single character string , of any length, that may contain any characters.

A variable that has not been assigned a value is an unitialized variable. An **unitialized variable has the upper case value of its name**.

Arithmetic Operators

Arithmetic operators can be applied to numeric constants, and to character strings, variables and expressions that evaluate to valid REXX numbers. REXX has the standard arithmetic operators identified by the following operator characters shown in **decreasing order of precedence**:

Prefix **-**	Unary - same as "0 - number".
Prefix **+**	Unary + same as "0 + number".
**	Power.
*	Multiply.
/	Divide.
%	Integer divide - divide and return only the integer part of the result.
//	Remainder - divide and return only the remainder.
+	Add.
-	Subtract.

REXX also has concatenation, comparison and logical operators.

Assignment

The REXX assignment operator is the '=' character. An assignment looks like this:

variable = *expression*

for example:

AreaCode = 408

An assignment is the easiest way to change the value of a variable.

REXX also has arithmetic, comparative, concatenation, and logical operators.

Comparative Operators

REXX comparative operators compare two terms and return the value '1' if the result of the comparison is true, or '0' otherwise. In **normal** comparisons leading and trailing blanks, and leading zeros are ignored. **Normal** comparisons are string comparisons unless **both** operators are numeric. String comparisons **are case sensitive,** and the shorter string is padded with blanks. Comparative operators are identified by the following operator characters:

=	Equal (numerically or when padded, etc.).
\=	Not equal (inverse of =).
>	Greater than.
<	Less than.
<>,><	Greater than or less than (same as "not equal").
>=, \>	Greater than or equal to.
<=, \<	Less than or equal to.

In **Strict** comparison the two strings must be identical to be equal. A shorter **substring** of a longer string is considered **less than** the longer string:

==	Strictly equal (identical).
\==	Strictly not equal (inverse of ==).
>>	Strictly greater than.
<<	Strictly less than.
>=, \>	Strictly greater than or equal to.
<=, \<	Strictly less than or equal to.

REXX also has arithmetic, concatenation and logical operators.

Concatenation Operators

REXX concatenation operators combine two strings to form one by appending the second string to the right-hand end of the first, with or without an intervening blank. Concatenation operators are identified by the following operator characters:

(Blank) Concatenate terms **with one blank** between.
|| Concatenate terms **without** an intervening blank.
(abuttal) Concatenate terms **without** an intervening blank.

Use the **||** operator to **force** concatenation **without a blank**. Normally when two terms are adjacent and not separated by an operator they are concatenated in the same way for example:

```
rate = 8.5
SAY rate'%'
```

Will produce the output **"8.5%"**

REXX also has arithmetic, comparative and logical operators.

Logical Operators

Logical Operators work with and return an indicator of an either TRUE (**1**) or FALSE (**0**) condition. Logical Operators work *only* with the string values 0 and 1, and the use of any other value is an error. Logical operators are used primarily in If() instructions.

The binary logical operators are:

&	**And**	Returns **1** if **both** terms are 1.
|	**Inclusive or**	Returns **1** if **either** term is 1.
&&	**Exclusive or**	Returns **1** if either but **not both** terms are 1.

The unary logical operator is:

**Prefix **	**Logical not**	Negates, 1 becomes 0 and 0 becomes 1.

An example using a Logical Operator is:

```
Day = Date(W)
If Day = 'Saturday' | Day = 'Sunday' then
Say "Today is a Weekend day."
```

REXX also has arithmetic, comparative and concatenation operators.

Enterprise REXX Windows Specific Functions.

The topics below describe the Enterprise REXX language extension specific to Windows™. Choose from the following list to review the area of interest:

NEW: Windows NT Event Log Management. (1.6.1)

- EventLogBackup() - Backup the Windows NT Event Log.
- EventLogClear() - Clear the Windows NT Event Log.

Windows NT and Windows 95 Registry Access.

- System Registry Access - Access and Create Registry information.

Electronic Mail Access.

- EMAIL Access - Find, read and send EMAIL messages.

INI file Access to private Windows Initialization Files.

- GetProfile() - Read Initialization file (INI file).
- SetProfile() - Write Initialization file (INI file).
- Program Example

Message and Prompt dialog box alternatives to SAY and PULL.

- MessageBox () - Present a message box.
- QuestionBox() - Present a question with a 'Yes' or 'No' answer.
- CancelBox() - Present a cancel box.
- PromptBox() - Prompt for input.
- ChoiceBox() - Present a list of choices.

Window and Keystroke Sending Functions.

- FindWindow() - Finds a top level window by Title Bar name.
- WinSendKeys() - Send keystrokes to another window.

Rexx Program control of the Source and I/O Console Windows.

With these functions you can now set your own ICONs!

- RexxSourceWindow() - Control the Rexx Source Window.
- RexxConsole() - Control the Rexx I/O Console Window.

Other Functions.

- CopyToClipboard() - Place text on the Windows Clipboard.
- GetFromClipboard() - Get text from the Windows Clipboard.
- GetRexxDirectory() - Get the path name of the Enterprise REXX directory.
- SPAWN() - Asynchronous Program Execution.
- WinMEM() - Windows Available Memory.
- WinSound() - Play specified sound.
- WinVersion() - Windows Version Number.
- WinPostMessage - Post Messages to a window.
- WinSendMessage - Send Messages to a window.

Command Caution:

You can get misleading return code values if the OS command interpreter execution is successful, with a return of ‘0’, while the *requested* command has failed. For example the command "Cmd.exe /c FooBar" would successfully execute the NT Command interpreter and give a return code of ‘0’. Also if your command contains redirection or piping, WinREXX runs the command interpreter implicitly. For example the command "foobar > stuff.txt" will return "0" from even though it fails (with "bad command or file name"), because Cmd.Exe (or Command.com for Win+DOS) succeeds.

Event Log Management (Windows NT only)

The EventLogBackup() and EventLogClear() functions allow for management of the Windows NT Event log. This is particularly useful on Windows NT Servers, especially those that run unattended.

The Event Log Management Functions Currently include:

EventLogBackup()(*BackupFile,* [*EventSource*], [*NTMachine*])
EventLogClear()([*BackupFile*], [*EventSource*], [*NTMachine*])

Where:

BackupFile is the file name of the event log backup copy.
EventSource is the name of an event source category,
e.g., System, Application or Security.
NTMachine is the name of Windows NT machine.

Return Values:

"0" for success or a message that begins with the word “Error” that explains the failure.

EventLogBackup(*BackupFile*, [*EventSource*], [*NTMachine*]) (Windows NT only)

EventLogBackup() creates a file that contains a backup copy of the information stored in the current Windows NT event log on *NTMachine*, or the current machine.

Arguments:

BackupFile is the file name for the event log backup copy. To save the backup file on a remote server, the backup filename may be a UNC path name, e.g., `\\ServerName\FilePathName`. If *BackupFile* specifies a remote server, only the event log on the local computer may be cleared. If this file already exists, the function fails. You cannot back up an event log from a remote server to a file on a remote server (even if the backup file and the original log are on the same server).

EventSource is the name of an Event Source category, i.e., the three standard categories of "System", "Security" or "Application", and **defaults to Application Events**. *EventSource* may also be a custom category, or a specific source under these categories. Keys for these are found in the System Registry under "`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog`".

NTMachine is the name of a Windows NT machine, or may be a NULL argument in which case it **defaults to the machine on which this function is called**, i.e., on which the Rexx program is running.

Returns:

Success: "0" is returned for success.
Failure: A message that explains the failure and begins with "Error …".

See also:

EventLogClear() and you may wish to look at an overview of Event Log Management.

EventLogClear([*BackupFile*], [*EventSource*], [*NTMachine*]) (Windows NT only)

EventLogClear() clears the event log on the specified (or current) machine, and optionally saves the current log messages to a backup file.

Arguments:

BackupFile is a NULL argument (to only clear the Event Log, and not also make a backup copy) or the file name for an Event Log backup copy. If this argument is NULL no backup copy is created. If this file already exists, the function fails.

To save the backup file on a remote server, the backup filename may be a UNC path name, e.g., `\\ServerName\FilePathName`. If *BackupFile* specifies a remote server, only the event log on the local computer may be cleared. You cannot back up an event log from a remote server to a file on a remote server (even if the backup file and the original log are on the same server).

EventSource is the name of an Event Source category, i.e., the three standard categories of "System", "Security" or "Application", and **defaults to Application Events**. *EventSource* may also be a custom category, or a specific source under these categories. Keys for these are found in the System Registry under "`HKEY_LOCAL_MACHINE\SYSTEM\CurrentControlSet\Services\EventLog`".

NTMachineName is the name of Windows NT machine. *NTMachine* may be a NULL argument in which case it **defaults to the machine on which ClearEventLog() is called**, i.e., on which the Rexx program is running.

Returns:

Success: "0" is returned for success.
Failure: A message that explains the failure and begins with "Error …".

See also:

EventLogBackup() and you may wish to look at an overview of Event Log Management.

SPAWN(*program_name*, *parameter_string, ShowStyle, [“synchronous”]*)
The Spawn() function runs other programs.

Program_name is a text string containing the name of program to be started, which may include the path. *Parameter_string* is a text string containing the input parameters to the program to be started. If *Program_name* contains redirection or piping symbols (‘**<**‘, ’**>**‘, or ‘**|**’), then WinREXX will ask the OS Command interpreter to run *Program_name* by adding the prefix “cmd.exe /c “ to your command (or “command.com /c “ for Win3.1_DOS).

The *ShowStyle* argument specifies how *Program_name*‘s window should be shown and may be one of the following values:

Normal	The default window behaviour.
Minimized	The window will run minized, as an Icon.
Maximized	The window will run maxized, “full screen”.
Hide	The window will not be shown.

Any other value will cause **Norm**al behaviour.
The *ShowStyle* argument may be omitted by using an ‘,’ with no following value.

Compatibility: For backward compatibility *ShowStyle* argument may also be an nCmdShow number defined in the Windows SDK. Allowed values are:
0 - SW_HIDE
1 - SW_SHOWNORMAL
2 - SW_SHOWMINIMIZED
3 - SW_SHOWMAXIMIZED

Sync**hronous*: Normally SPAWN() asynchronously executes *Program_name*, i.e., the REXX program does not wait for the "spawned" program to complete. If the fourth argument begins with the string "sync***", then SPAWN() synchronously executes *Program_name* and the REXX program **waits** for the "spawned" program to complete.

Return Values:

Synchronous SPAWN() places the spawned program’s “return code” in the special **REXX variable RC**. This is consistent with the synchronous execution of an inline command by the default COMMAND Environment.

SPAWN() returns 0 if successful and non-zero otherwise. Non-zero return codes include:

-2	file not found
-3	path not found
-8	insufficient memory
-11	executable file is invalid
-255	memory shortage or OS failure.

Caution: You *must* carefully understand the source of Return Codes.

In the case failure, e.g. the system is out of memory or in the Win3.1+DOS WinRexx1.dll can not be found, Spawn() will return a string describing the error, starting with the word “Error:”. For maximum integrity, your program should probably check for this “Error:” return value.

Spawn Example:
An example of SPAWN() usage is as follows:

```
/* Demonstration of Enterprise REXX Spawn */

/* Many options for Program will work:
 | - Simply "NOTEPAD" will work!
 | - A PIF (in the Windows directory) will work.
 | - Considerations for running DOS programs are
 |   the same as when run from File Manager or
 |   Program Manager.
 | - DOS path rules apply for finding the program.
```

```
 */

Program = "NOTEPAD.EXE"

Parms = "..\WREXX.INI"

Say "Run" Program "with parms" Parms"."
SpawnRC = Spawn(Program, Parms )
Say "Return code is" SpawnRC

Say "Run" Program "synchronously."
SpawnRC = Spawn(Program, Parms,, "sync" )

/* end of SPAWN.REX */
```

SPAWN.REX may be found in the SAMPLES directory.

GetProfile(*inifile, section, key*)

GetProfile() retrieves initialization file information. It provides access to WREXX.INI or any other Windows initialization file.

Arguments:

inifile	the name of the initialization file to access
section	the initialization file section to access
key	the key of the value to be retrieved

Returns:

The current value of the specified section of the named initialization file is returned. Returns 'not-found' if the initialization file, section or key does not exist.

See also:

SetProfile()

You may also wish to look at an example of GetProfile() usage.

SetProfile(*inifile, section, key, value*)

SETPROFILE() sets a value in an initialization file. It provides access to WREXX.INI or any other Windows initialization file.

Arguments:

inifile	the name of the initialization file to access
section	the initialization file section to access
key	the key of the value to be set
value	the value of the key to be set

Returns:

A non-zero value is returned if the value specified was successfully set. If SetProfile()fails to set the value specified, 0 is returned

Notes

Changes to WREXX.INI will usually take place when Enterprise REXX for Windows is restarted. To be certain that a change has taken effect, close Enterprise REXX for Windows and restart it.

See also:

GetProfile()

Initialization file (INI file) Access

The GetProfile() and SetProfile() functions provide access to WREXX.INI or any other Windows Initialization file.

Syntax of these functions are:

GetProfile(*INIFile*, *Section*, *Key*)
SetProfile(*INIFile*, *Section*, *Key, Value*)

Where:

INIFile	is the name of the initialization file to access.
Section	is the initialization file section.
Key	is the key of the value to be set or retrieved.
Value	is the value to be set (for SetProfile() only).

Return Values:

GetProfile() returns the current value of a specified *key* in a specified *section* of a named *INIfile*. GetProfile() returns "not-found" if the key, section or file does not exist.
SetProfile() returns "'0" if it fails to set the *value* specified for *key* in *section* of a named *INIfile* and non-zero if it is successful.

You may wish to look at an example of GetProfile() usage.

ProstGet.REX - Profile String Program Example.

The ProstGet.Rex program shows usage of the GetProfile() function:

```
/* PROSTGET.REXDemonstrate use of GetProfile() External Function */

/* Get a Path to our INI file.*/
   RexxFilePath = GetREXXDirectory()
/* Set up our INI variables.          */
   INIFile = RexxFilePath'\WREXX.INI'
   Section = "ExecOptions"
   Key = "LoggingOn"
/* Is Getprofile installed?           */
   func = "GetProfile"
   bIsProf = RxFuncQuery(func)
   if bIsProf = 0 then
      Say "RxFuncQuery RC for" func "is OK"
   else DO
      Say "RxFuncQuery RC for" func "is not OK"
      EXIT
      end
/* Read the INI file.                 */
   ProfRC = GetProfile(INIFile, Section, Key)
   If ProfRC = "not-found" then do
      Say "GetProfile RC is" ProfRC
      Say Key "is not in" INIFile Section
      end
   else do
      Say Key "=" ProfRC "in the ["Section"] section of" INIFile
      /* Some analysis if the Key was output logging. */
      if Key = "LoggingOn" then
         if profRC = 0 then
            Say "Output logging is *not* on."
         else
            Say "Output logging *is* on!"
      end

/* End of PROSTGET.REX */
```

GetRexxDirectory()

GetRexxDirectory()provides a way to determine the name of the Enterprise REXX for Windows directory.

Arguments:

none

Returns:

The GetRexxDirectory() function **returns the full path** name to the Enterprise REXX for Windows directory, without the ending backslash ('\'). The returned value is the directory from which WRXTRNL.DLL (or WrxFunc.DLL for Win32) was loaded.

Example:

GetRexxDirectory() can be used as follows:

```
/* Show path of Enterprise REXX for Windows directory.*/
Say "Win REXX is in the" GETREXXDIRECTORY() "directory."
```

GetREXXFileName()

The GetREXXFileName() function **returns the full path** name to the currently executing module. Use PARSEFN() to break the returned value into its constituent parts. Also see GETREXXDIRECTORY().

GetREXXFileName() can be used as follows:

```
/* Get a Path to our INI file.*/
RexxFilePath = GetREXXFileName()
tmp = parsefn(RexxFilePath)
Parse VAR tmp drive RexxPath file ext
/* Set up our INI variables.        */
INIFile = Drive':'RexxPath'WREXX.INI'
```

STACKSTACK86STATUS()

The STACK86STATUS() function reports the use of the 80x86 stack for testing and debugging.

Arguments:

none

Returns:

STACK86STATUS() returns a string consisting of two items:

1. the size of the 80x86 stack, in bytes
2. the maximum number of stack bytes used.

Notes:

The Enterprise REXX Utility function STACKSTATUS() returns information about the status of the console stack. Under Windows STACKSTATUS() always returns E 0 1000 (Enabled, 0th buffer, 1000 chars free)

Example:

Check stack usage by running a REXX program similar to the following:

```
stack86 = STACK86STATUS()
PARSE VAR stack86 size maxused
Say "Stack size is " size "bytes"
Say "Maximum stack used is " maxused "bytes"
Say "Minimum stack free size is " size-maxused "bytes"
```

WinMem()

WINMEM() provides a way to determine the amount of memory available.

Arguments:

none

Returns:

WinMem() returns the amount of memory available, including virtual memory under Windows enhanced mode.

Notes:

The amount of memory available is determined using the Windows GetFreeSpace() API and in Windows' 386 Enhanced mode and Windows NT represents *virtual* memory space which may be larger than the physical memory installed in your machine.

WinSound(*soundevent*)

WinSound()causes a sound file to be played by your currently installed audio driver on a sound device installed in your computer, e.g. a sound card (see **Drivers** from the **Windows Control Panel**). If no sound driver is installed, WinSound() will produce a standard beep sound on your computer's speaker.

Arguments:

soundevent a string which specifies the sound to be played and may be:

1. the name assigned to a .WAV file in the [sounds] section of WIN.INI.
2. the name of a .WAV file.
3. a pre-defined sound event that can be assigned to a .WAV file by selecting Sound from the Control Panel window.

Returns:

WinSound() always returns 0.

Notes:

If soundevent is in the [sounds] section of WIN.INI, the associated sound file is played.
If soundevent is not in the [sounds] section of WIN.INI, but is a .WAV file which can be found in Window's file search path, that .WAV file will be played.
If soundevent is neither of the above it can be one of the following pre-defined names:

Default Beep
Critical **S**top
Question
Exclamation
Asterisk

Only the bold portion of the soundevent name needs be specified.

If something other than one of the supported soundevent names is specified, the default beep will be played.

Example:

WinSound() can be used as follows:

```
/* have REXX Play some sounds. */
CALL WinSound "tada.wav"
CALL WinSound "Exclamation"
CALL WinSound "S"
```

WinVersion()

WinVersion()provides the current version of Windows.

Arguments:

none

Returns:

WinVersion() returns the current Windows version number (for example, 3.10).

Notes:

Use DOSVERSION() to obtain the current version of DOS (for example, 5.00).

CopyToClipboard(*TextString*)

The CopyToClipboard() and GetFromClipboard() functions allow text to be copied to and retrieved from the Windows clipboard. The samples program Clipper.rex demonstrates use of these functions.

CopyToClipboard() places the text contained in *TextString* on the Windows clipboard.

Arguments:

TextString the text to be placed on the clipboard

Returns:

Zero (0) is returned if the operations was successful, otherwise a non-zero value is returned.

Notes

Standard text (clipboard format CF_TEXT) is placed on the clipboard.

See also:

GetFromClipboard

You may also wish to look at an example of CopyToClipboard usage.

GetFromClipboard()

The GetFromClipboard() and CopyToClipboard() functions allow text to be retrieved from and copied to the Windows clipboard. The samples program Clipper.rex demonstrates use of these functions.

Arguments:

none

Returns:

The text currently on the Windows Clipboard if the operation was successful, or a NULL string if the clipboard is empty.

Notes

Only standard text (clipboard format CF_TEXT) is retrieved by this function.

See also:

CopyToClipboard

You may also wish to look at an example of GetFromClipboard() usage.

Clipper.REX - Windows Clipboard Program Example.

The Clipper.Rex program shows usage of GetFromClipboard() and CopyToClipboard() functions:

```
/* Clipper.rex - Demonstrate Clipboard functions - 24Aug93.
 |
 |      Uses CopyToClipboard( ) and GetFromClipboard()
 |      to move text to and from the Windows ClipBoard.
 |
 */
signal on novalue
Parse source Environ . OurName
say ""
TestString = "Program" OurName "running under" Environ"REXX."
say "The string '"TestString"'"
say "will be copied to the clipboard."

ClipRC = CopyToClipboard( TestString )
if ClipRC /= 1 then do
MessageBox( "ClipRC is" ClipRC", No Text on ClipBoard", OurName )
exit
end

MbRC = MessageBox(  "Copy new text onto the Clipboard, then press OK", OurName )
ClipText = GetFromClipboard()
say "The string '"ClipText"'"
say "came from the Clipboard."

exit

/* End of Clipper.Rex  */
```

RexxSourceWindow(*action, [arg1], [arg2]*)

RexxSourceWindow(*action*) controls many aspects of the Source Window.

***Actions* available:**

CloseAtExit to end WinREXX when you REXX program ends.
Title to set the title of the WinREXX Source window.
SetIcon to set the ICON of the WinREXX Source window.
Minimize, Maximize, Restore and Close to take the same actions as available in the Source windows' control menu, or to disable or enable those options.

CloseAtExit will cause Enterprise Rexx to end when your REXX program ends. This is the same action as provided by the **/E** command line argument as described above. The are **no arguments** for CloseAtExit.

Title will set the window title to the value of *Arg1*.

SetIcon will set the window's ICON to be an ICON extracted from the file identified by *Arg1*. *Arg1* may be the name of an executable file, a DLL or Icon file and may include a full path name. If the file specified by *Arg1* contains multiple ICONs, *Arg2* may specify the index of the ICON to be extracted and the default is to extract ICON 0, the fist ICON.

Minimize, Maximize, Restore and Close with no additional arguments will perform those actions on the window except that the REXX Source Window may not be closed. ***If Arg1 is provided*** it may be **Disable** or **Enable.** **Disable** will make the specified action, e.g. Minimize, Maximize or Restore, no longer available for the window. **Enable** will cause disabled options to be available.

Returns:

RexxSourceWindow()returns zero (0) if it is successful or an error description starting with "Error:".

See also:

RexxConsole

RexxConsole(*action, [arg1], [arg2]*)

RexxConsole(*action*) controls many aspects of the WinREXX I/O Console Window.

***Actions* available:**

Title to set the title of the WinREXX Console window.
SetIcon to set the ICON of the WinREXX Console window.
Minimize, Maximize, Restore and Close to take the same actions as available in the Console windows' control menu, or to disable or enable those options.

Title will set the window title to the value of *Arg1*.

SetIcon will set the window's ICON to be an ICON extracted from the file identified by *Arg1*. *Arg1* may be the name of an executable file, a DLL or Icon file and may include a full path name. If the file specified by *Arg1* contains multiple ICONs, *Arg2* may specify the index of the ICON to be extracted and the default is to extract ICON 0, the fist ICON.

Minimize, Maximize, Restore and Close with no additional arguments will perform those actions on the window. ***If Arg1 is provided*** it may be **Disable** or **Enable.** **Disable** will make the specified action, e.g. Minimize, Maximize or Restore, no longer available for the window. **Enable** will cause disabled options to be available.

Returns:

RexxConsole()returns zero (0) if it is successful or an error description starting with "Error:".

See also:

RexxSourceWindow

WinSendMessage (*window*, *message*, *wParam*, *lParam*)
WinPostMessage (*window*, *message*, *wParam*, *lParam*)

These functions are intended for use by persons familiar with programming MS-Windows. The **WinSendMessage()** and **WinPostMessage()** functions send or post a message to any window. (If you send a message it is processed synchronously with your call. If you post a message, it is queued for later processing.)

See SendMess.rex in the samples directory as an example. Also the program WinMsgs.Rex will store the IDs of some of the more useful messages.

Arguments:

These functions take the same four arguments as the comparable Windows SDK functions.

Returns:

WinPostMessage() returns non-zero if it is successful and zero (0) if it fails (same as the Windows PostMessage() API call). For WinSendMessage() the return value depends upon the message sent. Refer to your Windows programming documentation.

Notes:

Use the FindWindow() function to get the handle of the Window to which you want to send or post a message. Windows Message IDs can be stored and retrieved using GLOBALV.

Registry Access Functions.

These functions allow you to access the Windows NT and Windows 95 System Registry. The System Registry stores all information about your system's Hardware and Software configuration (including what's installed) and it's user or users. The System Registry is also where all installed OLE software components are registrered.

RegistryConnect - connect to the Registry of another System.
RegistryOpenRoot - open a Root Key in the Registry.
RegistryOpenKey - open a Registry Key.
RegistryCreateKey - create a Registry Key.
RegistryQueryValue - find the value of a Registry Key.
RegistrySetValue - set the value of a Registry Key.
RegistryEnumSubKeys - list keys under a currently open Key.
RegistryEnumValues - list the values under a currently open Key.
RegistryCurrentKey - returns the currently open Registry Key.
RegistryCurrentKeyPath - returns the full path of the current Key..
RegistryFlush - forces registry changes to be witten **now**.
RegistryClose - closes the open Registry Key.

RegistryConnect(SystemID, *RootKeyName*)

The Registry functions can alter a systems configuration, even making it inoperable, so only use them with discretion.

The **RegistryConnect()** function opens a Registry root key on another system on your network. Only the HKEY_LOCAL_MACHINE and HKEY_USERS keys may be opened on remote systems.

Arguments:

SystemID - the ID of the remote system for which you with to open the Registry, for example "Dev63", or "\\RainBird". The "\\" is optional.

RootKeyName must be provided and must be one of the following:
HKEY_LOCAL_MACHINE - information about the machine configuration.
HKEY_USERS - user information.

Returns:

RegistryConnect()returns zero (0) if it is successful and non-zero if it fails.

If a computer with the requested *SystemID* can not be found, the returned value will be a string starting with the word "Error:" and containing the string "computer name *SystemID* unavailable".

If any other failure occurs, the returned value will be a string starting with the word "Error:" and containing the Operating System Error return value.

Notes:

If you have connected to a remote system's registry, i.e. you have used RegistryConnect(), you **must** later call RegistryClose().

RegistryConnect() on your own SystemID, or with System omitted, is equivalent to the use of RegistryOpenRoot() on the same RootKeyName.

The "Registry Enumerate Remote.Rex" (RegRemot.rex) program in the ..\Samples uses RegistryConnect().

RegistryOpenRoot(*RootKeyName*)

The Registry functions can alter your system configuration, even making it inoperable, so they should be used only with discretion, especially on root keys other than the default HKEY_CURRENT_USER.

The **RegistryOpenRoot()** function opens a Registry root key, other than HKEY_CURRENT_USER which all other Enterprise REXX Registry functions use by default.

Arguments:

RootKeyName may be one of the following:
HKEY_CURRENT_USER - information pertaining to the current user.
HKEY_LOCAL_MACHINE - information about the machine configuration.
HKEY_USER - user information.
HKEY_CLASSES_ROOT - OLE and System Shell registration information.

Returns:

RegistryOpenRoot()always returns zero (0).

Notes:

To access the current user's information, the use of RegistryOpenRoot() is not required because the default root key used by RegistryOpenKey() is HKEY_CURRENT_USER. RegistryOpenRoot()must be used to access Registry information under any root key other than HKEY_CURRENT_USER.

RegistryOpenKey(*KeyName*)

The **RegistryOpenKey()** function opens an existing key, or key path, under the currently open key, or under the root key of HKEY_CURRENT_USER if no key is currently open.

Arguments:

KeyName - the Key name, which may include a path, to be opened, for example "Software\ Enterprise Alternatives\REXX".

Returns:

RegistryOpenKey() returns zero (0) if it is successful and non-zero if it fails.

If the Value does not exist, the return value will be a string starting with the word "Error:" and containing the string "does not exist".

If a failure occurs, the returned value will be a string starting with the word "Error:" and containing a description of the failure.

Notes:

If you have opened a registry key you **must** later call RegistryClose().

If no key is currently open, an attempt will be made top open *KeyName* under the HKEY_CURRENT_USER root key.

"Registry Key.Rex" (Regist~2.rex) in the ..\Samples directory shows an example use of RegistrySetValue().

RegistryCreateKey(*KeyName*)

The **RegistryCreateKey()** function creates a new key, or key path, under the currently open key. If *KeyName* already exists, it will be opened.

Arguments:

KeyName - the Key name to be created, which may include a key path, for example "Software\Enterprise Alternatives\REXX".

Returns:

RegistryCreateKey() returns zero (0) if it is successful and non-zero if it fails.

If a failure occurs, the returned value will be a string starting with the word "Error:" and containing a description of the failure.

Notes:

The Registry functions can alter your system configuration, so they should be used only with discretion, especially on root keys other than the default HKEY_CURRENT_USER.

If no key is currently open, *KeyName* will be created under the HKEY_CURRENT_USER root key.

Because RegistryCreateKey() will open an existing key, it can be used in place of RegistryOpenKey() when not testing for key existence.

"Registry Key.Rex" (RegOpen.rex) in the ..\Samples directory shows an example use of RegistrySetValue().

RegistryQueryValue(*ValueName*)

The **RegistryQueryValue()** function returns the Data associated with a value under the current key in the Registry.

Arguments:

ValueName - the Value for which the current associated Data is to be returned.

Returns:

RegistryQueryValue() returns the Data associated with a Value under the current key in the Registry. If no Data exists RegistryQueryValue() returns an empty string.

If the Value does not exist, the return value will be a string starting with the word "Error:" and contained the string "does not exist".

If a failure occurs, the returned value will be a string starting with the word "Error:" and containing a description of the failure.

Notes:

None.

RegistrySetValue(*Value,* [*Data*] *,* [*DataType*])

The **RegistrySetValue()** function sets a value, and optionally it's data, under the currently open Registry key.

Arguments:

Value - the value to be set.
Data - optionally, the data to be associated with the *Value*.
DataType - optionally, the type of the *Data*, which defaults to string data.
DataType may be one of the following:

REG_SZ or "string" for character string data.
REG_DWORD or "dword" for a 32 bit number.

Returns:

RegistrySetValue() returns zero (0) if it is successful and non-zero if it fails.

If a failure occurs, the returned value will be a string starting with the word "Error:" and containing a description of the failure.

Notes:

The Registry functions can alter your system configuration, so they should be used only with discretion, especially on root keys other then HKEY_CURRENT_USER (the default).

"Registry Values.Rex" (RegValue.rex) in the ..\Samples directory shows an example use of RegistrySetValue().

RegistryEnumSubKeys( )
RegistryEnumValues ( )

The **RegistryEnumSubKeys()** and **RegistryEnumValues()** functions are used to enumerate they Subkeys and Values, respectively, under the currently open Registry key.

Arguments:

none.

Returns:

RegistryEnumSubKey() and RegistryEnumValues() return the value of the "next" SubKey or Value under the currently open key, and return zero (0) if there are no more entries which can occur on the first call if no entries exist. Call these functions successively, until zero (0) is returned, to enumerate all SubKeys and/or Values.

If a failure occurs, the returned value will be a string starting with the word "Error:" and containing a description of the failure.

Notes:

"Registry Enumerate.Rex" (RegEnum.rex) in the ..\Samples directory shows an example use of RegistryEnum*() functions.

RegistryCurrentKey ( )
RegistryCurrentKeyPath ( )

The **RegistryCurrentKey ()** and **RegistryCurrentKeyPath()** functions return the currently open Registry key, or complete key path. These are information functions useful mostly for testing new REXX procedures that use the System Registry.

Arguments:

none.

Returns:

RegistryCurrentKey() and RegistryCurrentKeyPath() return the current key or key path name, which is an **empty string** if no key is currently open.

Notes:

None.

RegistryFlush( )

The **RegistryFlush()** function causes all registry changes to be written to the registry database files, i.e., flushed. Changes will be flushed before RegistryFlush() returns for the call. Normally changes are written later, e.g. some time after RegistryCreateKey() or RegistrySetValue() returns, as a system background activity.

Arguments:

none.

Returns:

RegistryFlush()always returns zero (0).

Notes:

Use RegistryFlush() only to guarantee when new registry information has been written, e.g. before new information is to be used by another process. Frequent use of RegistryFlush() could negatively affect system performance.

RegistryClose( )

The **RegistryClose()** function closes the currently open Registry key.

Arguments:

none.

Returns:

RegistryClose() returns zero (0) if it is successful and non-zero if it fails.

Notes:

See also the RegistryOpenRoot() and RegistryOpenKey() functions.

MESSAGEBOX(msg,[title])

MESSAGEBOX() provides an alternative to SAY. A message is presented in a dialog box which contains an OK button.

Arguments:

msg	the message to be displayed which may be up to 250 characters long.
title	the title for the message box,which may be up to 50 characters long

Returns:

Pressing the OK button returns 'OK' . Pressing the <Esc> key or the <Enter> key also returns 'OK'.

See also:

CANCELBOX(), CHOICEBOX(), PROMPTBOX(), and QUESTIONBOX()

QUESTIONBOX(msg,[title])

QUESTIONBOX() provides an easy way to present a question to a user which has a 'yes' or 'no' answer.

Arguments:

msg	the question to be asked which may be up to 250 characters long
title	the title of the question box which may be up to 50 characters long.

Returns:

The text of the key that the user pressed, i.e., 'Yes' or 'No'. The <Enter> key presses the currently selected button. No action is taken when the <Esc> key is pressed.

See also:

CANCELBOX(), CHOICEBOX(), MESSAGEBOX(), and PROMPTBOX()

CANCELBOX(msg,[title])

CANCELBOX() displays a dialog box allowing the user to cancel. A message is presented in a dialog box which contains a Yes, a No, and a Cancel button

Arguments:

msg	the message to be displayed which may be up to 250 characters long.
title	the title for the cancel dialog box which may be up to 50 characters long.

Returns:

The text of the button which the user pressed, i.e., 'Yes', 'No' or 'Cancel'. The <Esc> key returns 'Cancel' and the <Enter> key presses the currently selected button.

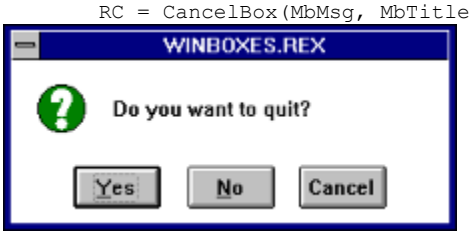
See also:

CHOICEBOX(), MESSAGEBOX(), PROMPTBOX(), and QUESTIONBOX()

Example:

The following statements from the WinBoxes.REX program will produce the cancel box shown below:

```
PARSE SOURCE Environment ExecType ProgName
MbTitle = ProgName
MbMsg = "Do you want to quit?"
RC = CancelBox(MbMsg, MbTitle)
```



PROMPTBOX(msg,[title],[defresponse])

PROMPTBOX() displays a dialog box showing your message or question, and an edit text box in which to enter the response or answer to your question. The prompt box contains an OK and a CANCEL button.

Arguments:

msg	The message to be displayed, up to 80 characters long.
title	The title for the message box, up to 50 characters long.
defresponse	The default response up to 250 characters long.

Returns:

The **text string in the edit text box** when the OK button is pressed, or a NULL string if the CANCEL button or the <Esc> key is pressed.

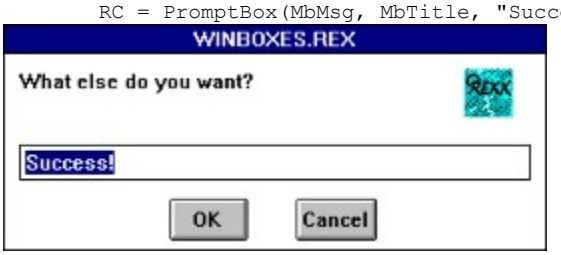
See also:

CANCELBOX(), CHOICEBOX(), MESSAGEBOX(), and QUESTIONBOX()

Example:

The following statements from the WinBoxes.REX program will produce the prompt box shown below.

```
PARSE SOURCE Environment ExecType ProgName
MbTitle = ProgName
MbMsg = "What else do you want?"
RC = PromptBox(MbMsg, MbTitle, "Success!")
```



Your REXX program can **test for** the return of **a NULL string**, as in the WinBoxes example, to determine **if** a response has been entered. WinBoxes.REX may be found in the ..\SAMPLES sub-directory.

CHOICEBOX([msg],[title],stemvar)

CHOICEBOX() displays a dialog box showing your message or question, and a list of the values of all first level variables under a REXX stem variable. One item from the list may be selected as the response or answer to your question. The choice box contains an OK and a CANCEL button.

Arguments:

msg	the message to be displayed, up to 80 characters long.
title	the title for the choice box, up to 50 characters long.
stemvar	the name of a REXX stem variable which contains the values to be displayed.

Returns:

The **text value of the selected name** when the OK button is pressed, or a NULL string if the Cancel button is pressed.

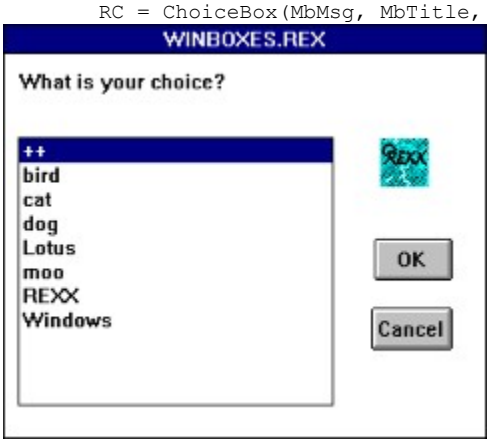
See also:

CANCELBOX(), MESSAGEBOX(), PROMPTBOX(), and QUESTIONBOX()

Example:

The following statements from the WinBoxes.REX program will produce the choice box shown below.

```
PARSE SOURCE Environment ExecType ProgName
MbTitle = ProgName
Choice.peep = "bird" ; Choice.cow = "moo"
Choice.Meow = "cat" ; Choice.bark = "dog"
Choice.GUI = "Windows" ; Choice.car = "Lotus"
Choice.C = "++" ; Choice.language = "REXX"
MbMsg = "What is your choice?"
RC = ChoiceBox(MbMsg, MbTitle, Choice.)
```



Your REXX program can **test for** the return of **a NULL string**, as in the WinBoxes example, to determine if a response has been entered. WinBoxes.REX may be found in the ..\SAMPLES sub-directory.

Enterprise REXX Language Extensions

Enterprise REXX extends the REXX language with a rich set of utility functions. This includes extensions specific to Microsoft Windows and also utility functions in common with other REXX implementations, e.g., on IBM mainframe systems.

Choose from the following list to review the area of interest:

Windows Specific functions exclusive to Windows.
Utility Functions for NT and DOS “system” activities.
LISTFILE, EXECIO and RXWINDOW are **not** supported.
GLOBALV supports permanent global REXX variables.

Enterprise REXX Utility Functions.

Enterprise REXX supports *many* system Utility Functions. If you are new to REXX, you may want to look at some general information on using functions, especially OS CMD functions.

Click on a group name for a list of functions available.

DOS & NT CMD Function Group.
Hardware Information Group.
Hardware Access Group.
Miscellaneous Functions Group.

Error 99 - *Function not supported under Windows* will be returned for Hardware Information and Access (and a few DOS specific) functions that are not implemented by Enterprise REXX for Windows. Some other functions always return a fixed value.

Enterprise REXX DOS and NT CMD Functions

Many DOS and NT CMD utility functions **are supported**. Click on the function name for usage information.

_CD() or DOSCD()
_ChDir() or DOSCHDIR()
_ChMod() or DOSCHMOD()
_Copy() or DOSCOPY()
_Creat() or DOSCREAT()
_Del() or DOSDEL()
_Dir() or DOSDIR()
_DirPos() or DOSDIRPOS()
_Disk() or DOSDISK()
_Drive() or DOSDRIVE()
GetEnv() or DOSENV()
GetEnvSize() or DOSENVSIZE() is **not** supported - returns error 99.
_FDate() or DOSFDATE()
_FName() or DOSFNAME()
_IsDev() or DOSISDEV()
_IsDir() or DOSISDIR()
_SYSMEM() or DOSMEM() is **not** supported - returns error 99. Use WINMEM().
_MkDir() or DOSMKDIR()
_Pid()
_PathFind() or DOSPATHFIND()
_Rename() or DOSRENAME()
_RmDir() or DOSRMDIR()
_OSVersion() or DOSVERSION()Use WINVERSION() for Windows version.
_Volume() or DOSVOLUME()

Enterprise REXX Miscellaneous Functions

Many miscellaneous Functions **are supported**. Click on the function name for usage information.

DATECONV()
EMSMEM() does **not** apply to Windows - always returns 0.
ENDLOCAL() saves only current drive/directory variables.
FCNPKG() always returns 0.
LOWER()
PARSEFN()
PRXSWAP() does **not** apply to Windows - always returns 0.
PRXVERSION()
RXFUNCADD()
RXFUNCDROP
RXFUNCQUERY
SETLOCAL() saves only current drive/directory variables.
STACKSTATUS() returns 'E 0 1000' (Enabled)
UPPER()
VALIDNAME()

EXECIO (from mainframe VM REXX) is **not** supported.
LISTFILE (from mainframe VM REXX) is **not** supported.

Enterprise REXX Hardware Information Functions

For compatibility reference all of the Quercus Systems' Personal REXX functions are listed. Enterprise REXX supports those supported by Personal REXX for Windows. Click on the function name for usage information.

PCDISK() - with limits.
PCDISPLAY() always returns 3 2 7.
PCEQUIP()
PCFLOPPY()
PCGAME()
PCKEYBOARD() always returns 3.
PCPARALLEL()
PCRAM() is **not** supported - returns error 99. Use WINMEM().
PCROMDATE() is **not** supported - returns error 99.
PCSERIAL()
PCTYPE()
PCVIDEO() always returns 4.

Enterprise REXX Hardware Access Functions

In general Enterprise REXX (and Personal REXX for Windows) do not support the hardware access. For compatibility reference all of the Quercus Systems' Personal REXX Hardware Access functions are listed here. Click on the function name for usage information.

CHARSIZE() does **not** apply to Windows - returns error 99.
CURSOR() does **not** apply to Windows - returns error 99.
CURSORTYPE() does **not** apply to Windows - returns error 99.
DELAY() **is** supported.
INKEY() does **not** apply to Windows - returns error 99.
INP() **is** supported.
OUTP() **is** supported.
PEEK() is **not** supported - returns error 99.
POKE() is **not** supported - returns error 99.
SCRBLINK() does **not** apply to Windows - returns error 99.
SCRCLEAR() does **not** apply to Windows - returns error 99.
SCRMETHOD() returns 'W'.
SCRPUT() does **not** apply to Windows - returns error 99.
SCRREAD() does **not** apply to Windows - returns error 99.
SCRSIZE() does **not** apply to Windows - returns error 99.
SCRWRITE() does **not** apply to Windows - returns error 99.
SHIFTSTATE() does **not** apply to Windows - returns error 99.
SOUND() always returns the same sound.

RXWINDOW is **not** supported.

General Differences from Quercus Systems' Personal REXX for DOS

LISTFILE and EXECIO, GLOBALV and RXWINDOW

(LISTFILE and EXECIO and GLOBALV are REXX extensions familiar to IBM mainframe VM Operating System REXX users.)

LISTFILE and EXECIO **are not** supported The REXX I/O functions **LINEIN()**, **LINEOUT()**, **CHARIN()** and **CHAROUT()** often are more efficient methods of doing file I/O, and **should be used**.

GLOBALV supports permanent global REXX variables.

The RXWINDOW set of character mode screen utility functions supplied with Quercus Systems' Personal REXX **for DOS** are **not** supported by Enterprise REXX for Windows or Personal REXX for Windows.

While LISTFILE and EXECIO **are** supported, the REXX I/O functions **LINEIN()**, **LINEOUT()**, **CHARIN()** and **CHAROUT()** often are more efficient methods of doing file I/O, and **should be used** whenever possible.

LISTFILE, EXECIO and GLOBALV are built into PrxCmd32.dll (or PrxCmd.dll for the Win16 version) and no installation is necessary except that PrxCmd32.dll (or PrxCmd.dll) must be in the same directory as RxRexx.dll (or Wrexx.dll). Note that this implementation of GlobalV replaces the previous partial implementation.

The EXECIO PRINT option is **not** supported. If the PRINT option is specified, the lines read will be displayed in the Rexx I/O Console window.

Note that these external commands do not yield execution control to Windows and therefore under windows 3.1 other Windows applications will be “locked out” during their execution. This is not an issue with the Win32 version of these commands under Windows NT or Windows 95.

GLOBALV - persistent global variables

Global Variables

When a REXX Program finishes executing, the values of all its variables are lost. The GLOBALV command allows you to create global variables whose values remain in effect as long as necessary, allowing values determined in one REXX program to be accessed by REXX programs that you run later.

Enterprise REXX for Windows supports only permanent global variables and this GLOBALV data is stored in GLOBALV.INI.

GLOBALV Subcommands

The following subcommands are supported:

INIT	Initializes the currently selected variable group, i.e. deletes all variables in the group.
SELECT	Selects a global variable group.
PUT	Saves a variable and its current value into the currently selected global variable group.
GET	Gets the current value of a global variable from the currently selected variable group, and creates the variable if it does not exist.
PUTS	Behaves just like PUT
PUTP	Behaves just like PUT

Under DOS and OS/2 the GLOBALV command supports a number of additional subcommands (SET, SETL, LIST, STACK, PURGE, GRPLIST AND GRPSTACK) which are not available under Windows.

You may wish to have more information on the persistence of variable group selection.

No Installation of GLOBALV is required. For examples of GLOBALV, see GLOBVGET.REX and GLOBVSET.REX in the SAMPLES directory.

Persistence of variable group selection

It may be useful to be able to access, display, and reset the global variables associated with a particular task as a group. If these groups are kept separate, you can avoid the problem of two unrelated tasks trying to use variables of the same name for different purposes. Every global variable is a member of a specific global variable group. Initially, all GLOBALV operations take place on a global variable group called the 'UNAMED' group. 'GLOBALV SELECT groupname' tells GLOBALV that future operations should take place on variables in the named group.

RXWINDOW

RXWINDOW is **not** supported in Enterprise REXX for Windows.

RXFUNCADD(functionname,DLLname,entrypoint)

RXFUNCADD() registers the function name, making it available for use by the same or other REXX programs. The named DLL must be in the current directory, the Windows or Windows ..\SYSTEM directory, the execution directory of the current Windows task, or a directory in the PATH.

Arguments:

functionname	The external function name to be used in the REXX program.
DLLname	The DLL file name which contains the function. If the file extension is **.DLL** it need not be specified.
entrypoint	The DLL export name of the 'C' function to call in DLLname.

Returns:

RXFUNCADD() returns 0 if successful

Notes:

RXFUNCADD() may only **register by name** external functions **exported by function name** in a DLL. If DLLname or entrypoint do not exist, RXFUNCADD() may still be successful, but an error will occur when the registered functionname is used.

The use of this function is shown in the PROSTGET.REX program distributed with Enterprise REXX for Windows.

RXFUNCADD() is the same function as that of IBM's OS/2 1.3 REXX.

See also:

RXFUNCDROP() and RXFUNCQUERY()

RXFUNCDROP(functionname)

RXFUNCDROP() eliminates or un-registers functionname from the list of available external functions, making it **unavailable** for use by REXX programs.

Arguments:

functionname The name of the external function to be dropped.

Returns:

RXFUNCDROP() returns 0 if it is successful.

Notes:

Any REXX program running in any Window of any Windows Task may drop a function, regardless of what program registered it. Therefore care must be taken when dropping a function as **the execution of other REXX programs can be affected**. It is not necessary to drop a function unless the same functionname is to be registered in a different DLL.

RXFUNCDROP() is the same function as that of IBM's OS/2 1.3 REXX.

See also:

RXFUNCADD() and RXFUNCQUERY

RXFUNCQUERY(functionname)

RXFUNCQUERY() determines if functionname exists in the list of available external functions.

Arguments:

functionname The name of the external function to be checked for availability.

Returns:

RXFUNCQUERY() returns 0 if functionname exists as a registered function name.

See also

RXFUNCADD() and RXFUNCDROP()

REXX Instructions

Listed below are the Instructions of version 4.0 of the REXX language. All of these are provided by Enterprise REXX. Note that the REXX language Built-in Functions are listed separately.

You may also wish to look at these other topics:

Notational Conventions used herein.
Enterprise REXX language extensions.
Windows Extensions to the REXX Language.

Alphabetical listing of REXX instructions:

ADDRESS [environment [command]]
ADDRESS VALUE [environment]
ARG [template]
CALL name [expression] [,expression] ...
CALL ON condition [NAME handler]
CALL OFF condition
DO [repetitor] [conditional]; [statement-list] END [symbol]
DROP name [name] ...
EXIT [expression]
IF expression THEN statement1; [ELSE statement2]
INTERPRET expression
ITERATE [symbol]
LEAVE [symbol]
NOP
NUMERIC DIGITS [expression]
NUMERIC FORM [form]
NUMERIC FUZZ [expression]
PARSE [UPPER] source [template]
PROCEDURE [EXPOSE name [name] ...]
PULL [template]
PUSH [expression]
QUEUE [expression]
RETURN [expression]
SAY [expression]
SELECT; when-list [OTHERWISE [statement-list]] END
SIGNAL name
SIGNAL VALUE expression
SIGNAL ON condition [NAME handler]
SIGNAL OFF condition
TRACE [VALUE] expression

Notational Conventions

Several conventions are used in the following instruction syntax summaries. REXX keywords are in upper case. These must be spelled as shown, though any mixture of lower and upper case may actually be used. Elements in lower case represent user-supplied information. Anything enclosed in brackets ([]) is optional. Alternative forms of the instruction are listed on separate lines. Ellipses (...) indicate that the preceding element may be repeated. Semicolons may be included at the end of any clause. They are included only when required in a context that is not the end of a line.

ADDRESS [environment [command]]
ADDRESS VALUE [environment]

Summary: changes the current default external command environment or issues a command to a specified external environment.

Arguments:

environment: name of a command environment

command: command to issue to a command environment

Notes: When ADDRESS is used by itself, it makes the previous command environment the current environment. If VALUE is not used, the specified environment is taken literally as a name without evaluation. If no command is specified, the environment named becomes the current command environment. If a command is included, it is issued to the specified environment. Use the ADDRESS built in function to determine the name of the current default environment.

ARG [template]

Summary: converts program or procedure arguments to upper case and parses them according to a supplied parse template. See also the ARG built in function.

Arguments:

template: a parse template

Notes: The template may contain one or more sub-templates, separated by commas. Each sub-template is used to parse the corresponding argument. If there are more sub-templates than arguments, variables named in the sub-template are set to a null string. ARG is equivalent to PARSE UPPER ARG.

CALL name [expression] [,expression] ...
CALL ON condition [NAME handler]
CALL OFF condition

Summary: either calls a subroutine with specified expression values as arguments or enables or disables a handler for a specified condition.

Arguments:

name: the name of a subroutine, which may be a label in the program, the name of a built-in function, or the name of an external function.

expression: argument to the subroutine

condition: one of the following condition names: ERROR, FAILURE, HALT, or NOTREADY

handler: the name of a handler for the specified condition

Notes: The first form of CALL is a normal subroutine call. The second form enables a handler for a particular condition. If no handler name is specified, it is the same as the condition name. The third form disables any existing handler for the specified condition.

DO [repetitor] [conditional]; [statement-list] END [symbol]

Summary: delimits a group of statements which may be treated as a single statement and optionally controls repetitive execution.

Arguments:

repetitor: either an expression, the keyword FOREVER, or a phrase of the form assignment [TO expt] [BY expb] [FOR expf], where expt, expb, and expf are expressions.

conditional: either WHILE expression or UNTIL expression

statement-list: zero or more statements separated (if on the same line) by semicolons

symbol: the symbol which is the target of an assignment when the assignment form of repetitor is used.

Notes: TO, BY, and FOR may be used in any order in an assignment repetitor. The expressions following TO, BY, or FOR may not contain the keywords WHILE or UNTIL.

Examples:

Here is a most simple DO loop:

```
DO 4
   SAY "hello"
   END
```

Here is a more capable DO loop:

```
DO ctrl = 1 TO 10
   SAY ctrl
   END
```

And even more:

```
DO x = 1 TO limit          /* find multiples up to limit */
   DO y = 1 TO limit
      SAY x "x" y "=" x * y
      END y                /* control variable */
   END x                   /* control variable */
```

DROP name [name] ...

Summary: resets simple and compound variables to an uninitialized state.

Arguments:

name: a symbol that names a variable or a stem, or a symbol enclosed in parentheses

Notes: When a stem is dropped, all variables having that stem become uninitialized. When a name is enclosed in parentheses, it is assumed to be a string consisting of names of other variables. All variables named in the list (but not the list variable itself) become uninitialized.

EXIT [expression]

Summary: terminates execution of a REXX program and passes a return value to the caller.

Arguments:

expression: value to be returned to caller

Notes: Only the current REXX program is terminated. A calling REXX program (if any) will resume execution at the point the current program was invoked.

IF expression THEN statement1; [ELSE statement2]

Summary: conditionally executes a statement based on the value of an expression.

Arguments:

expression: a REXX expression that evaluates to 0 or 1

statement1: statement that is executed if the expression value is 1

statement2: statement that is executed if the expression value is 0

Notes: THEN is a reserved word and may not be used in the expression. Either statement may be a DO group, consisting of a list of statements contained between DO and END.

INTERPRET expression

Summary: executes one or more REXX statements that are generated as the value of an expression.

Arguments:

expression: an arbitrary REXX expression

Notes: The value of the expression should be a list of REXX statements separated by semicolons. DO, IF, and SELECT statements (if any) must be complete. The statements are executed as if they were a part of the program at that point.

ITERATE [symbol]

Summary: causes control to pass to the top of an iterative DO group.

Arguments:

symbol: the name of the control variable of an active DO group

Notes: The control variable (if any) will be incremented appropriately and the terminating conditions will be tested as if the END statement closing the DO group had been encountered. A symbol may be specified to identify the DO group.

LEAVE [symbol]

Summary: causes control to pass to the statement following the END of an iterative DO group.

Arguments:

symbol: the name of the control variable of an active DO group

Notes: A symbol may be specified to identify the DO group.

NOP

Summary: instruction that does nothing.

Notes: NOP can be used as the instruction required after THEN in an IF or SELECT instruction.

NUMERIC DIGITS [expression]
NUMERIC FORM [form]
NUMERIC FUZZ [expression]

Summary: defines certain parameters of REXX numeric representation and arithmetic.

Arguments:

expression: REXX expression that evaluates to a positive integer

form: either a literal SCIENTIFIC or ENGINEERING, or an expression that evaluates to SCIENTIFIC or ENGINEERING

Notes: NUMERIC DIGITS is (roughly) the number of significant digits retained in a numeric value. NUMERIC FUZZ is the number of least significant digits ignored when doing numeric comparisons. NUMERIC FORM specifies whether the exponent of a number in exponential form should be a multiple of three. See also the FORM built in function and the FUZZ built in function which return current value of NUMERIC FORM and FUZZ respectively.

PARSE [UPPER] source [template]

Summary: parses an input string into REXX variables according to rules specified in a template.

Arguments:

source: defines the source of the input string, which can be:

ARG	program or subroutine arguments
LINEIN	line read from standard input stream
PULL	line read from external data queue
SOURCE	information about the program
VALUE	the value of an expression
VAR	the value of a variable
VERSION	information about the REXX language processor

template: a parse template

Notes: When the source is VALUE it must be followed by an expression and then the reserved word WITH (which cannot occur in the expression). When the source is VAR it must be followed by the name of a variable.

PROCEDURE [EXPOSE name [name] ...]

Summary: creates a new generation of variables for a subroutine

Arguments:

name: a symbol that names a variable or a stem, or a symbol enclosed in parentheses

Notes: When a stem is exposed, all variables having that stem are exposed. When a name is enclosed in parentheses, it is assumed to be a string consisting of names of other variables. All variables named in the list and the list variable are exposed.

PULL [template]

Summary: converts to upper case and parses a line of input read from the external data queue or the standard input stream.

Arguments:

template: a parse template

Notes: PULL is equivalent to PARSE UPPER PULL.

PUSH [expression]

Summary: places a line of data in the external data queue.

Arguments:

expression: data to be placed in the queue

Notes: The data is placed in the queue LIFO ("last-in-first-out"). A null string is placed in the queue if the expression is omitted.

QUEUE [expression]

Summary: places a line of data in the external data queue.

Arguments:

expression: data to be placed in the queue

Notes: The data is placed in the queue FIFO ("first-in-first-out"). A null string is placed in the queue if the expression is omitted.

RETURN [expression]

Summary: terminates execution of a subroutine and passes a return value to the caller.

Arguments:

expression: value to be returned to caller

Notes: RETURN does not terminate a REXX program unless it occurs in the topmost procedure of the program.

SAY [expression]

Summary: writes data to the standard output stream (usually the terminal).

Arguments:

expression: the data to be written

Notes: A null string is written if the expression is omitted. SAY is generally equivalent to a call to LINEOUT with the first argument omitted.

SELECT; when-list [OTHERWISE [statement-list]] END

Summary: execute a statement depending on a set of conditional expressions.

Arguments:

when-list: a list of clauses of the form WHEN expression THEN statement

statement-list: one or more REXX statements, separated by semicolons (if on the same line)

Notes: Each expression following a WHEN is evaluated in sequence. The expression must evaluate to 0 or 1. The statement following THEN is executed for the first expression that has the value 1. THEN is a reserved word which cannot be used in any of the expressions. If none of the expressions have the value 1, the statements following OTHERWISE (if present) are executed.

SIGNAL name
SIGNAL VALUE expression
SIGNAL ON condition [NAME handler]
SIGNAL OFF condition

Summary: either transfers control to a specified label in the program or enables or disables a handler for a specified condition.

Arguments:

name: a label in the program

expression: a REXX expression whose value is a label in the program

condition: one of the following condition names: ERROR, FAILURE, HALT, NOTREADY, NOVALUE, or SYNTAX

handler: the name of a handler for the specified condition

Notes: The first two forms of SIGNAL are used to transfer control to the specified label. All active DO loops are terminated, but control remains within the currently active procedure. The third form enables a handler for a particular condition. If no handler name is specified, it is the same as the condition name. The fourth form disables any existing handler for the specified condition.

TRACE [VALUE] expression

Summary: controls REXX program tracing. See also the TRACE Built-in function.

Arguments:

expression: selects type of tracing as follows:

A - trace all clauses

C - trace commands to external environments

E - trace external commands that end with an error

F - trace external commands that end with a failure

I - trace all clauses and intermediate results of expressions

L - trace all labels

N - same as F (the default)

O - disable tracing

R - trace all clauses and final results of expressions

Notes: If the expression is not a symbol or literal but does begin with a symbol or literal, it must be preceded by the keyword VALUE. The value of the expression may be prefixed with a "?" to indicate that interactive tracing is to be toggled on or off.

REXX Built-in Functions

Listed below are the Built In Functions of version 4.0 of the REXX language. All of these functions are provided by Enterprise REXX.

You may wish to look at these other topics

- Notational Conventions used herein.
- Instructions of the REXX language.
- REXX language extensions.
- Windows specific REXX Language Extensions.

SOURCELINE([number])
SPACE(string, [count], [pad])
STREAM(stream, [option], [command])
STRIP(string, [option], [character])
SUBSTR(string, start, [length], [pad])
SUBWORD(string, start, [length])
SYMBOL(name)
TIME([option])
TRACE([type])
TRANSLATE(string, [output], [input], [pad])
TRUNC(number, [digits])
VALUE(name, [value], [type])
VERIFY(string, search, [option], [start])
WORD(string, number)
WORDINDEX(string, number)
WORDLENGTH(string, number)
WORDPOS(phrase, string, [start])
WORDS(string)
XRANGE([first], [last])
X2B(hex-string)
X2C(hex-string)
X2D(hex-string, [length])

ABBREV(string1, string2, [length])

Summary: indicates whether one string is a beginning segment of another.

Arguments:

string1: the "long form" being checked for abbreviation.

string2: the string which is a potential abbreviation.

length: the minimum length of string2 that will qualify as an abbreviation.

Notes: The function returns 1 if string2 is a substring of string1, starting at the first position and if it is at least length characters long, otherwise it returns 0. The default for length is the length of string2.

ABS(number)

Summary: returns the absolute value of its argument.

Arguments:

number: a valid REXX number

Notes: The result is formatted according to the current setting of NUMERIC DIGITS.

ADDRESS()

Summary: returns the name of the current default environment.

Notes: The default environment name is set with the ADDRESS instruction.

ARG([argument-number], [option])

Summary: returns either the number of arguments, the value of a specific argument, or whether a specific argument has been included or omitted. See also the ARG Instruction.

Arguments:

argument-number: the number of the argument in question.

option: one of the following:

E - test whether argument exists.

O - test whether argument was omitted.

Notes: If no argument is specified, ARG returns the number of arguments passed to the current internal or external procedure. If only the argument number is specified, ARG returns the value of the designated argument. If option is also specified, ARG returns 0 or 1 to indicate whether the argument was present.

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

Summary: returns the logical AND of its arguments.

Arguments:

string1: first operand of AND.
string2: second operand of AND. Default is null string.
pad: character appended before the operation to the shorter of the two operands to make them equal in length if the operands are of different lengths.

Notes: BITAND produces the logical bitwise AND of its operands. If no pad character is specified, the default is 'ff'x.

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

Summary: returns the logical OR of its arguments.

Arguments:

string1: first operand of OR.
string2: second operand of OR. Default is null string.
pad: character appended before the operation to the shorter of the two operands to make them equal in length if the operands are of different lengths.

Notes: BITOR produces the logical bitwise OR of its operands. If no pad character is specified, the default is '00'x.

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

Summary: returns the logical XOR of its arguments.

Arguments:

string1: first operand of XOR.
string2: second operand of XOR. Default is null string.
pad: character appended before the operation to the shorter of the two operands to make them equal in length if the operands are of different lengths.

Notes: BITXOR produces the logical bitwise XOR of its operands. If no pad character is specified, the default is '00'x.

B2X(binary-string)

Summary: returns the hexadecimal representation of the given binary string.

Arguments:

binary-string: character string consisting of 0's and 1's to be converted.

Notes: Both the argument and the result of B2X are character strings. B2X coverts the input data from base 2 representation to base 16.

CENTER(string, length, [pad])

Summary: centers a string in a field of a specified width.

Arguments:

- string: the string to be centered.
- length: width of the field.
- pad: character to be added if the field width exceeds the length of the string.

Notes: If the string is longer than the width of the field, CENTER returns the central length characters of the string. If an odd number of characters are to be either added or removed, one more character is added to or removed from the right end than the left end.

CharIn([*stream*], [*StartPosition*], [*count*])

The **CharIn()** function returns characters read from the specified input stream, typically a file.

Returns:

CharIn() returns the characters read. Less than ***count*** characters will be returned if end-of-file or an error condition is encountered.

Arguments:

stream*:** name of the input stream. If ***stream is **omitted** or is a null string then **CharIn()** reads from the standard input stream, i.e. STDIN.

StartPosition*:** location within the input stream at which to begin reading. The first character of a stream is position 1. If ***StartPosition is **omitted** the default is the current read position, which initially is the beginning of the stream and thereafter is the character following the last one read by **CharIn()** or **LineIn()**. If ***StartPosition*** is **zero (0)** the read pointer is positioned to the beginning of the stream and no characters are read. ***StartPosition*** can **not** be specified for STDIN or a hardware device.

count*:** number of characters to read. If ***count is 0, **CharIn()** moves the current read position to the specified ***StartPosition*** and returns a null string. If ***count*** is **omitted**, one character is read, i.e. the default count is 1.

CharOut([*stream*], [*string*], [*position*])

The **CharOut()** function writes a string of characters to the specified output stream and if all characters are successfully written zero (0) is returned.

Returns:

CharOut() returns the number of characters (if any) which could **not** be written. In some conditions the NOTREADY condition is raised.

Arguments:

stream*:** name of the output stream. If ***stream is **omitted** then **CharOut()** writes to the standard output stream, i.e. STDOUT. This is equivalent to LineOut() to standard out, or the SAY instruction, except that a carriage return-linefeed is not appended to the written string. This is useful for standard out to a (console) window because the cursor is not moved to the next line.

string*:** data to be written. If ***string is **omitted**, the current write position is updated to the value specified by position.

position*:** location within the output stream at which to begin writing. If ***position is **omitted** the default is the current write position, which initially is past the current end of the stream and thereafter is following the last one written by **CharOut()** or **LineOut()**.

Notes: If both string and position are omitted, i.e., **if only *name* is specified, the output stream is closed**. No end-of-file character is written at the end of the file.

CHARS([stream])

Summary: returns the number of characters remaining to be read in the specified input stream.

Arguments:

stream: name of the input stream. Default is the standard input stream.

Notes: The number of characters remaining to be read is defined to be the number of characters from the current read position to the end of the file.

COMPARE(string1, string2, [pad])

Summary: returns the position of the first mismatch between the two input strings.

Arguments:

string1: first string to be compared.
string2: second string to be compared.
pad: character appended before the operation to the shorter of the two operands to make them equal in length if the operands are of different lengths.

Notes: If the two strings are identical (after padding) COMPARE returns 0.

CONDITION([option])

Summary: returns information associated with the current trapped condition.

Arguments:

option: select type of information, which can be one of the following:

C - name of the trapped condition.

D - further descriptive information about the trapped condition.

I - the instruction that invoked the condition handler.

S - state of handling for the condition.

Notes: CONDITION returns a null string if no condition has been raised.

COPIES(string, count)

Summary: returns a concatenation of the specified number of copies of the input string.

Arguments:

string: the string to be copied.
count: number of copies.

Notes: Count must be a non-negative whole number. COPIES returns a null string if count is 0.

C2D(data, [length])

Summary: returns the value of the input data interpreted as a decimal number.

Arguments:

data: the data to be converted.

length: rightmost number of bytes of input data to be converted.

Notes: The data is assumed to be a binary representation of a number with the most significant byte at the left and the least significant byte at the right of the string. If a length is not specified, the number is assumed to be unsigned. If a length is specified, the number is assumed to be signed, and only the rightmost length bytes are converted, padded on the left with 0 if necessary. The result must be a valid whole number according to the current setting of NUMERIC DIGITS.

C2X(data)

Summary: returns the hexadecimal representation of the input data.

Arguments:

data: the data to be converted.

Notes: C2X returns a string consisting of hexadecimal digits (0-9, A-F) which give the internal representation of the input data.

DATATYPE(string, [type])

Summary: returns the type of the input string, or an indication of the class to which the data belongs.

Arguments:

string: the string whose type is needed.

type: a letter which selects a string type to test for, one of the following:

A - alphanumeric (a-z, A-Z, or 0-9).
B - binary (0 or 1).
L - lower case (a-z).
M - mixed case (a-z or A-Z).
N - number (a valid REXX number).
S - symbol (only characters valid in REXX symbols).
U - upper case (A-Z).
W - whole number (valid whole number with current DIGITS).
X - hexadecimal (a-f, A-F, 0-9).

Notes: If type is not specified, DATATYPE returns "NUM" or "CHAR" depending on whether or not the string is a valid REXX number. If the type is specified, it returns 1 or 0 depending on whether or not the string belongs to the designated class.

DATE([option])

Summary: returns the current date.

Arguments:

option: a character that indicates the required date format, one of the following:
- B - base date, number of complete days since January 1, 0001.
- D - number of the current day of the year, starting with 1.
- E - date in European format (dd/mm/yy).
- M - full English name of the current month.
- N - date in default format (dd Mmm yyyy).
- O - date in orderable format (yy/mm/dd).
- S - date in standard format (yyyymmdd).
- U - date in US format (mm/dd/yy).
- W - full English name of the current day.

Notes: The default format, which is provided if option is not specified, consists of 2 digits for the day, the first 3 characters of the English name of the month, and the year.

DELSTR(string, start, [length])

Summary: returns the input string from which a substring starting at a specified position is deleted.

Arguments:

- string: the input string.
- start: the starting position of the substring to be deleted.
- length: the length of the substring to be deleted.

Notes: If the length is not specified, all characters from the start position are deleted. If the start position is beyond the right end of the string, the string is returned unchanged.

DELWORD(string, start, [length])

Summary: returns the input string from which a substring starting at a specified word is deleted.

Arguments:

- string: the input string.
- start: the number of the word which starts the substring to be deleted.
- length: the length in words of the substring to be deleted.

Notes: If the length is not specified, the remainder of the string beginning with the specified word is deleted. If the number of the first word to be deleted is greater than the number of words in the string, the string is returned unchanged.

DIGITS()

Summary: returns the current value of the NUMERIC DIGITS setting.

D2C(number, [length])

Summary: returns the internal representation of a decimal number

Arguments:

number: whole number to be converted.
length: length of the result.

Notes: D2C returns the internal representation of a number as a string of characters with the most significant byte at the left. If length is not specified, the number to be converted must be non-negative. If length is specified and it is less than required for the entire internal representation, the leftmost (most significant) bytes are truncated. If length is longer than required, the result is sign-extended on the left.

D2X(number, [length])

Summary: returns the internal representation of a decimal number in hexadecimal form

Arguments:

number: whole number to be converted.
length: length of the result in characters.

Notes: D2X returns the internal representation of a number as a string of hexadecimal digits with the most significant digit at the left. If length is not specified, the number to be converted must be non-negative. If length is specified and it is less than required for the entire internal representation, the leftmost (most significant) digits are truncated. If length is longer than required, the result is sign-extended on the left.

ERRORTEXT(number)

Summary: returns the text of the message associated with the specified error.

Arguments:

number: the number of an error in the range 0 - 99.

Notes: If no error message is associated with the specified number, a null string is returned.

FORM()

Summary: returns the current value of NUMERIC FORM as set by the NUMERIC instruction.

Notes: Possible values of NUMERIC FORM are SCIENTIFIC or ENGINEERING.

FORMAT(number, [m], [n], [exp1], [exp2])

Summary: formats a number with a given number of digits before and after the decimal point, and with a given number of digits in the exponent.

Arguments:

number:	the number to be formatted.
m:	number of digits before the decimal point in the result.
n:	number of digits after the decimal point in the result.
exp1:	number of digits in the exponent of the result.
exp2:	number of digits required to trigger exponential notation.

Notes: The number is first rounded as in the result of the expression "number + 0". The default values of m and n are the number of digits required for the integral and fractional parts, respectively. Exp2 is the trigger for exponential notation, whose default is NUMERIC DIGITS. That is, exponential notation will be used if the number has more than exp2 digits before the decimal point or more than 2*exp2 digits after the decimal point.

FUZZ()

Summary: returns the current value of NUMERIC FUZZ as set by the NUMERIC instruction.

Notes: NUMERIC FUZZ is the number of least significant digits that will be ignored in numeric comparisons of equality or inequality.

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

Summary: inserts one string at a certain position in a second string.

Arguments:

string1: the string to be inserted.
string2: the string inserted into.
pos: the position in the second string at which the first is inserted.
length: length to which inserted string is extended or truncated.
pad: character appended to the first string when its length is to be extended.

Notes: The first string is inserted in the second string after the character identified by pos. The default position is 0, in which case the first string is inserted before the first character of the second. The default value of length is the length of the first string.

LASTPOS(target, string, [start])

Summary: returns the last position of one string in another, searching from right to left.

Arguments:

target: the string being searched for.
string: the string which is searched.
start: the starting position in the second string at which the search begins.

Notes: LASTPOS returns 0 if the target string is null or if it is not found. The default start position is LENGTH(string).

LEFT(string, length, [pad])

Summary: returns the leftmost part of the input string

Arguments:

- string: the string to be truncated or extended.
- length: desired length of the result.
- pad: character added to the right of the input string when its length is to be extended.

Notes: The input string is left-justified in a field of specified length or truncated if necessary.

LENGTH(string)

Summary: returns the length in characters of the input string

Arguments:

string: the input string.

LINEIN([stream], [position], [count])

Summary: returns a line read from the specified input stream.

Arguments:

stream:	name of the input stream. Default is the standard input stream.
position:	location within the input stream at which to begin reading, specified as a relative line number. This value must be 1 if used.
count:	number of whole lines to read (0 or 1 only).

Notes: The default position is the current read position, which is either the first character of the stream or the character following the last one read by CHARIN or LINEIN. The default count is 1. With a count of 0, LINEIN returns a null string but moves the current read position as specified by the second argument.

LINEOUT([stream], [string], [position])

Summary: writes a string of characters to the specified output stream and returns the number of (0 or 1) which could not be written.

Arguments:

stream:	name of the output stream. Default is the standard output stream.
string:	data to be written.
position:	location within the output stream at which to begin writing, specified as a relative line number.

Notes: The default position is the current write position, which is the position following either the last character of the stream or the last one written by CHAROUT or LINEOUT. If the string is omitted, the current write position is updated to the value specified by position. If both string and position are omitted, the output stream is closed.

LINES([stream])

Summary: Returns 1 if there are more lines to be read in the specified input stream beyond the current read pointer. Returns 0 if there are no more lines to be read in the specified input stream beyond the current read pointer.

Arguments:

stream: name of the input stream. Default is the standard input stream.

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

Summary: returns the largest of a list of numbers.

Arguments:

number: a valid REXX number.

Notes: The result is rounded according to the current setting of NUMERIC DIGITS. That is, like the value of the expression "number + 0".

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

Summary: returns the smallest of a list of numbers.

Arguments:

number: a valid REXX number.

Notes: The result is rounded according to the current setting of NUMERIC DIGITS. That is, like the value of the expression "number + 0".

OVERLAY(string1, string2, [pos], length], [pad])

Summary: returns the result of replacing the characters of one string by the characters of another, starting at a certain position.

Arguments:

string1: the string of replacement characters.
string2: the string being overlayed.
pos: the position in the second string at which the first overlays it.
length: length to which overlaying string is extended or truncated.
pad: character appended to the first string when its length is to be extended.

Notes: The default position is 1, in which case the first string overlays the second starting at the beginning. The default value of length is the length of the first string.

POS(target, string, [start])

Summary: returns the position of one string in another, searching from left to right.

Arguments:

target: the string being searched for.
string: the string which is searched.
start: the starting position in the second string at which the search begins.

Notes: POS returns 0 if the target string is null or if it is not found. The default start position is 1.

QUEUED()

Summary: returns the number of lines contained in the external data queue.

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

Summary: returns a quasi-random whole number.

Arguments:

- max: maximum value that can be returned.
- min: minimum (non-negative) value that can be returned.
- seed: a whole number which is used to generate the first of a repeatable sequence of quasi-random numbers.

Notes: If only one argument is specified, it is assumed to be the maximum value, in which case the minimum will be 0. Otherwise the defaults for min and max are 0 and 999.

REVERSE(string)

Summary: returns the input string with the characters reversed end-for-end.

Arguments:

string: the string to be reversed.

RIGHT(string, length, [pad])

Summary: returns the rightmost part of the input string.

Arguments:

- string: the string to be truncated or extended.
- length: desired length of the result.
- pad: character added to the left of the input string when its length is to be extended.

Notes: The input string is right-justified in a field of specified length or truncated if necessary.

SIGN(number)

Sign() returns the arithmetic sign of the input number.

Arguments:

number: a valid REXX number.

Notes: SIGN returns -1 for a negative number, 1 for a positive number, and 0 dfor 0.

SOURCELINE([*number*])

SourceLine() returns number of lines in the program or the specified line of the program's source code.

Arguments:

number: the number of the line to return. If this argument is omitted then SourceLine() returns the number of lines in the program.

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

Space() returns the input string reformatted with a specified number of pad characters between each blank-delimited word.

Arguments:

string: the input string.
count: number of pad characters inserted between each blank-delimited word.
pad: the character to be inserted between words of the input string.

Notes: The default count is 1 and the default pad character is a blank. If count is 0, all blanks are removed. SPACE always removes leading and trailing blanks.

Stream (*stream*, [*option*], [*command*])

The **Stream()** function performs an operation, such as file seek, on a specified I/O stream.

Arguments:

***stream*:** name of the I/O stream.

Option*:** type of information to be returned or operation to be performed. ***option may be one of the following:

'C' - perform a command specified by third argument.
'D' - return extended information about the state of the stream.
'S' - return indication about the state of the stream.

For an *Open Stream*, the 'S' and 'D' options (or no option) will return either READY, NOTREADY, ERROR. For a stream that is not open they will return UNKNOWN.

***command*:** Is the command to be performed if option is "C". The following commands are supported:

open read will open the stream in READ mode.

open or **open write** will open the stream in READ WRITE mode.

close will close the stream.

seek { *n* | +*n* | -*n* | <*n* | =*n* } where ***n*** is a number less than 9 digits in length.

'**=**' specifies the seek should be from the beginning.

'**<**' specifies the seek should be from the end of the stream.

'**+**' or '-'specify the direction of the seek, the default is '+'.

The **default** is seek from the beginning.

query { exists | size | datetime }

'**exists**' returns the fully qualified path name of the file, if it exits, or a NULL string if it does not.

'**size**' returns the size of the stream, the last time it was closed, or a NULL string if the stream does not exist. **Note** that if you are writing to a stream, e.g. with lineout(), the stream must be closed before **query size** will return it's current size.

'**datetime**' returns the file date and time in the form "mm-dd-yy hh:mm:ss" if the stream exits, or a NULL string if it does not.

Notes: Only the "S" option behavior is specified of the REXX language specification, and other behavior of STREAM is unstandardized and dependent on the implementation and therefore particular to Enterprise REXX.

Strip (string, [option], [character])

Strip() returns the input string with specified leading or trailing characters removed.

Arguments:

string: the input string.

option: specifies whether leading or trailing characters are to be removed. It can be one of the following:

B - both leading and trailing characters (the default).

L - only leading characters.

T - only trailing characters.

character: the character to be stripped from the input string. Default is a blank.

SubStr (*string*, *start*, [*length*], [*pad*])

SubStr() returns a substring of the input string.

Arguments:

string: the input string.
start: the beginning position in the input string of the desired substring.
length: length of the desired substring.
pad: character added to the left of the substring when its length is to be extended. Default is a blank.

Notes: The substring may extend beyond the right end of the input string, in which case it is extended with pad characters. The start position may be greater than LENGTH(string), in which case the result will consist entirely of pad characters.

SubWord(*string*, *start*, [*length*])

Suubword() returns a substring of the input string.

Arguments:

string: the input string.
start: the beginning position in the input string of the desired substring, expressed in terms of blank-delimited words.
length: length in words of the desired substring.

Notes: Leading and trailing blanks will be removed from the result. If the length is greater than the number of words remaining in the string, only the remainder is returned. The default for length is the number of words left in the string.

SYMBOL(name)

Summary: returns an indication of whether a given string is a valid symbol, and if so whether it has an assigned value.

Arguments:

name: a string that represents a possible symbol name.

Notes: SYMBOL returns "BAD" if name is not a valid symbol name (for instance, the string contains characters not allowed in a symbol). SYMBOL returns "VAR" if the string is the name of a symbol which has been assigned a value. Otherwise it returns "LIT".

TIME([option])

Summary: returns the current time.

Arguments:

option: a character that indicates the required time format, one of the following:

C - civil format: hh:mm, followed by "am" or "pm".
E - elapsed time in seconds since timer was reset (R option).
H - complete hours since midnight.
L - hh:mm:ss.uuuuuu format (fractional seconds).
M - complete minutes since midnight.
N - normal time format (hh:mm:ss), 24-hour clock (default).
R - reset time and return time elapsed since last reset.
S - complete seconds since midnight.

Notes: Time values are never affected by NUMERIC DIGITS setting. The E and R options can be used for computing elapsed time without concern for crossing midnight.

TRACE([type])

Summary: returns current trace settings and optionally changes them. See also the TRACE instruction.

Arguments:

type: new trace setting in the same form as used in the TRACE instruction.

Notes: When the TRACE function is used to change trace settings it works generally like the TRACE instruction, except that counts cannot be specified, and the settings will be changed even during interactive tracing.

TRANSLATE(string, [output], [input], [pad])

Summary: replaces specified characters in an input string.

Arguments:

- string: the string to be modified.
- output: the table of output characters.
- input: the table of input characters.
- pad: pad character used to extend the output table when it is shorter than the input table.

Notes: Every occurrence in the input string of a character from the input table is replaced by the corresponding character from the output table. The pad character, which defaults to a blank, is used as a replacement when there is no corresponding character in the output table, because it is shorter than the input table. Characters not present in the input table are not changed. The default input table is XRANGE('00'x, 'ff'x). If neither input nor output table is specified, TRANSLATE converts all lower case characters to upper case.

TRUNC(number, [digits])

Summary: formats a number with a given number of digits after the decimal point.

Arguments:

number: the number to be formatted.
digits: number of digits after the decimal point in the result.

Notes: The number is first rounded as in the result of the expression "number + 0". The default for digits is 0, in which case the result will be an integer without a decimal point.

Value(*name*, [*value*], [*type*])

Value() returns the value of a specified REXX or OS ENVIRONMENT variable and optionally changes it. See also PARSE VALUE.

Arguments:

name: a string that is the name of a variable.

value: new value for the specified variable.

type: system-dependent type or class of variable to be accessed. The default is REXX variables of the current generation. For Enterprise REXX specify "ENVIRONMENT" for ***type*** in order to access the OS ENVIRONMENT instead of the REXX variable pool. Any other value for ***type*** will produce REXX Error 40.. Note that with the Win16 version of Enterprise REXX (for Windows 3.1x), the VALUE() function can only be used to retrieve the value of an environment variable, not to set it.

Notes: The VALUE function can be used instead of an INTERPRET statement to fetch or set a variable whose name isn't known until run-time. If a new value is not specified, the variable is not changed.

When VALUE references REXX variables of the current generation (the default), the specified name is upper cased and subject to substitution if it is a compound name.

VERIFY(string, search, [option], [start])

Summary: indicates whether or not characters from a given set occur in a specified string.

Arguments:

- string: the string to be searched.
- search: a string composed of the characters to be searched for.
- option: either 'N' (nomatch) to find the location of the first character of string that is not in the search string, or 'M' (match) to find the first character of the string that is in the search string. Default is 'N'.
- start: the starting position in the string for the search. Default is 1.

Notes: VERIFY returns 0 if string is entirely composed of characters in search (option 'N'), if string is entirely comprised of characters not in search (option 'M'), if the string to be searched is null, or if the start position is beyond the end of the string.

WORD(string, number)

Summary: returns a specific blank-delimited word from a string.

Arguments:

string:	the input string.
number:	number of the word to select from the string.

Notes: WORD returns a null string if there are fewer than number words in the string.

WORDINDEX(string, number)

Summary: returns the character position of a specific blank-delimited word in a string.

Arguments:

string: the input string.
number: number of the word whose index is required.

Notes: WORDINDEX returns 0 if there are fewer than number words in the string.

WORDLENGTH(string, number)

Summary: returns the length of a specific blank-delimited word in a string.

Arguments:

string: the input string.
number: number of the word whose length is required.

Notes: WORDLENGTH returns 0 if there are fewer than number words in the string.

WORDPOS(phrase, string, [start])

Summary: returns the word position of one string of words in another, searching from left to right.

Arguments:

phrase: the string of words being searched for.
string: the string which is searched.
start: the starting word position in the string at which the search begins. Default is 1.

Notes: WORDPOS returns 0 if the phrase being searched for is a null string or is not found. Excess blanks between words in the target and search strings are ignored.

WORDS(string)

Summary: returns the number of blank-delimited words in a string.

Arguments:

string: the string whose length is required.

XRANGE([first], [last])

Summary: returns a string of all characters whose encodings lie in a given range.

Arguments:

first: the first character in the range. Default is '00'x.
last: the last character in the range. Default is 'ff'x.

Notes: The result consists of characters in ascending order if first is less than last. If first is greater than last, the result consists of characters in ascending order, but wrapping at 'ff'x. The result depends on the specific character collating sequence used by the implementation.

X2B(hex-string)

Summary: returns the binary-string representation of a given hexadecimal string.

Arguments:

hex-string: the hex string whose binary-string representation is required.

Notes: Both the argument and the result of X2B are character strings. X2B converts the input data from base 16 representation to base 2.

X2C(hex-string)

Summary: returns the binary (internal) representation of a given hexadecimal string.

Arguments:

hex-string: the hex string whose binary representation is required.

Notes: The result of X2C is the internal representation of the given hex string. It will normally contain nonprintable characters.

X2D(hex-string, [length])

Summary: returns the decimal representation of a given hexadecimal string.

Arguments:

hex-string: the hex string whose decimal representation is required.
length: rightmost number of hex digits to be converted.

Notes: The input string must consist of valid hex digits. The digits may be separated by spaces as long as each blank-delimited group except the first contains an even number of hex digits. The spaces are ignored in selecting the rightmost digits when length is specified. If length is specified, the result is a signed number. Otherwise the result will be unsigned. If length exceeds LENGTH(hex-string), the string is first padded on the left with 0.

Utility Functions Overview

Enterprise REXX includes a set of built in utility functions that allow your programs to take advantage of the full power of the PC. Among those supplied are functions that give you information about your PC's hardware configuration, allow you to interface with DOS's file system and directory structure, and give you more direct access to your screen, cursor position, and keyboard.

These functions are part of Enterprise REXX and are **not** part of the REXX language. These functions are **not** likely to be found in other implementations of the REXX language. While some functions are specific to DOS Windows or Windows NT, most functions are available in all these environments.

Some of these functions will be of interest only to advanced PC users who are acquainted with some of the more technical details of DOS and the PC.

Hardware Information Group

\index{Functions, hardware information}\index{Extensions;hardware functions}%

\index{Functions, hardware information;PCDISK}\index{PCDISK function}%

PCDISK(subfunction,[drive])

Returns information about the installed fixed and floppy disks. Under Windows only the **N**umber subfunction is available. All other options return error 99 - function not supported under Windows.

Possible subfunctions are:

> **N**umber - Total *number* of fixed and floppy disks. (With DOS 3.0 or later, it returns the maximum number of drives, as controlled by the CONFIG.SYS LASTDRIVE parameter.)
>
> **H**eads - Number of disk read-write *heads* (surfaces) on specified drive (that is, the number of tracks per cylinder). (n/a under Windows)
>
> **C**ylinders - Number of disk *cylinders*. (n/a under Windows)
>
> **S**ectors - Number of *sectors* per track. (n/a under Windows)

Only the first letter of each subfunction needs to be specified. *Drive* can be specified only for the **H**, **C**, and **S** subfunctions. If *drive* is omitted for these subfunctions, the current DOS drive is assumed. The **H**, **C**, and **S** subfunctions apply only to hard disks, and the information returned is not always valid on non-IBM machines or with non-IBM hard disks.

PCDISPLAY() DOS Only

Returns a string containing three numbers, separated by blanks, giving information about your display and display adapter. The values returned indicate the display mode, the display type, and the adapter type.

Windows

Under Windows PCDISPLAY() always returns 3 2 7 (see codes below).

DOS

The **display mode** is controlled by function 0 of the BIOS Video Interrupt, and is documented in IBM's Technical Reference manuals. Possible display modes include:

2	80 column alphanumeric mode with color burst disabled
3	80 column alphanumeric mode with color burst enabled
7	80 column monochrome

The **display type** is 1 for a monochrome display and 2 for a color display.

The **adapter type** is the value returned by function 1AH of the BIOS Video Interrupt, as documented in IBM's Technical Reference manuals. (Other methods are used to determine the adapter type on systems that don't support this BIOS call.) Values that can be returned include:

1	IBM Monochrome Display Adapter
2	IBM CGA (Color/Graphics Adapter)
4	IBM EGA with color display
5	IBM EGA with monochrome display
6	IBM PGA
7	IBM VGA with monochrome display
8	IBM VGA with color display
11	IBM MCGA with monochrome display
12	IBM MCGA with color display

For example, on a PS/2 in 80 by 25 text mode with a VGA and an 8513 color display, PCDISPLAY() is 3 2 8.

Because of the variety of possible displays, adapters, and BIOS versions, the values returned by PCDISPLAY() represent Enterprise REXX's best guess as to your video configuration, but may not be accurate for all configurations.

PCEQUIP()

Returns the BIOS equipment flags, as a string of 0's and 1's. (Several of the other functions in this group obtain their information by looking at these flags.)

PCFLOPPY()

Returns the number of floppy disks installed, as indicated in the BIOS equipment flags.

PCGAME()

Returns 1 if a game port is installed, else 0, as determined from the BIOS equipment flags.

PCKEYBOARD() DOS Only

Windows

Under Windows PCKEYBOARD() always returns 3.

DOS

Returns 0 if the keyboard is the old PC or XT non-``enhanced" keyboard that does not have function keys 11 and 12, separate cursor keys, etc. Returns 1 for enhanced keyboards. This is useful for deciding whether to assign program functions to the extended keys. It may also be useful in conjunction with the INKEY function and input functions in the RXWINDOW function package.

PCPARALLEL()

Returns the number of parallel ports, as determined from the BIOS equipment flags.

PCRAM() DOS Only

Windows

Use the WINMEM() function to obtain the amount of memory available, inlcuding virtual memory under Windows Enhanced mode. Under Windows PCRAM() will generate error 99 - function not supported under Windows.

DOS

Returns the amount of RAM storage, as determined from the BIOS, in units of 1024 bytes. Usually does not include memory above 640K.

PCROMDATE() DOS Only

Windows

Under Windows PCROMDATE() will generate error 99 - function not supported under Windows.

DOS

Returns the date (*mm/dd/yy*) of the installed ROM.

PCSERIAL()

Returns the number of serial ports, as determined from the BIOS equipment flags.

PCTYPE()

On IBM machines, returns a number that indicates the type of PC. Possible values are:

-1 --- not known to Enterprise REXX
 0 --- IBM PC
 1 --- IBM PC/XT or Portable
 2 --- IBM PCjr
 3 --- IBM PC/AT
 4 --- IBM XT 286
 5 --- IBM PC Convertible
25 --- IBM PS/2 Model 25
30 --- IBM PS/2 Model 30
50 --- IBM PS/2 Model 50
55 --- IBM PS/2 Model 55
60 --- IBM PS/2 Model 60
70 --- IBM PS/2 Model 70
80 --- IBM PS/2 Model 80

Note: PC compatibles may return other values.

PCVIDEO() DOS Only

Windows

Under Windows PCVIDEO() always returns 4.

DOS

Returns a number that identifies the initial video mode, as determined from the BIOS. Possible values are:

1 --- 40 x 25 black & white, graphics adapter
2 --- 80 x 25 black & white, graphics adapter
3 --- 80 x 25 black & white, monochrome adapter

The PCDISPLAY() function gives more complete information about installed displays and adapters.

OS CMD Functions

For several of the DOS functions group, there are OS commands that perform similar tasks. For example,

```
'cd \barbara'
```

would invoke the DOS CD command to make \BARBARA become the current directory, while

```
call doschdir '\barbara'
```

would invoke a Enterprise REXX function to do the same thing. Either method can be used, however:

Running the OS command under Windows requires either swapping out Windows (in standard mode) or starting a virtual DOS machine (in enhanced mode or under NT).

Even under DOS, the Enterprise REXX functions are in some cases more efficient than invoking the DOS command.

The Enterprise REXX functions return an indicator of success or failure to your REXX program.

In the above examples, if the subdirectory \BARBARA did not exist, the OS CD command would give REXX no indication of an error, while the call to DOSCHDIR() would set the REXX variable *RESULT* to 0 to indicate the error.

Note:

```
call doschdir \barbara
```

is invalid. Quotes around \barbara are required:

```
call doschdir '\barbara'
```

Note also that DOSCHDIR() is a function, so it returns a result. You can either invoke it with a CALL instruction, in which case the result is assigned to the REXX variable *RESULT*, or you can invoke it as a function and assign the result directly to a variable, as in

```
directory_changed = doschdir('\barbara')
```

What you should not do is invoke it as a function without assigning the result to some variable. For example,

```
doschdir('\barbara')
```

would return either **0** or **1**, and REXX would then try to execute an OS command called **0** or **1**, which is probably not what you intended.

DOSCD([drive]) or _CD([*drive*])

Returns the current directory on the current DOS drive if *drive* is omitted; otherwise returns the current directory on the specified drive.

Returns a null string if an invalid drive is specified.

DOSCHDIR(pathname) or _ChDir(*pathname*)

DOSCHDIR() makes *pathname*, which may include drive and path information, become the current directory for the current (or specified) drive.

DOSCHDIR() returns 0 if it encountered any error conditions, or 1 if it was successful.

DOSCHMOD(fileid,[turnon],[turnoff]) or _ChMod(*fileid*, [*turnon*], [*turnoff*])

Is used to change a file's attribute bits. *Fileid*, which must be given and which may contain drive and path information, specifies the file involved. It can contain no wildcard characters.

> *Turnon*, which can be omitted, consists of one or more of **H**, **S**, **R**, or **A**, allowing you to turn on a file's **H**idden, **S**ystem, **R**ead-only, or **A**rchive bits.
>
> *Turnoff*, which can also be omitted, is specified in the same way, and lets you specify which bits are to be turned off.

DOSCHMOD() returns 1 if the attributes were successfully changed, and 0 if not (for example, file not found, drive not ready, access denied).

For example, to turn on a file's Archive bit and turn off its Hidden and Read-only bits:

```
CALL DOSCHMOD fileid, 'A', 'HR'
```

DOSCOPY(*from*, *to*)or
_Copy(*from*, *to*)

DosCopy() will copy one file to another, like the command line COPY command. If the ***to*** file exists, it will be replaced.

Return Values

- 0 for success.
- non-zero for failure.
  - 2 if the ***from*** file is not found.
  - 123 if the ***from*** or ***to*** specification contains wildcards or is a directory.

Note: DosCopy() is more efficient than using the COPY command, which requires a process be started to run the OS command processor (CMD.EXE or COMMAND.COM).

DOSCREAT(fileid) or _Creat(*fileid*)

If *fileid*, which may contain drive and path information, does not exist, it is ``created'' as an empty file (0 bytes in length). If *fileid* already exists, it is truncated to 0 bytes in length, and its existing contents are lost.

DOSCREAT returns 0 if it encountered any error conditions, or 1 if it was successful.

DOSDEL(fileid) or
_Del(*fileid*)

Fileid which may contain drive and path information but should not contain wildcard characters, is deleted (that is, erased).

DOSDEL() returns 0 if it encountered any error conditions, or 1 if it was successful.

DOSDIR([fileid],[output],[search],[mask],[position]) or _Dir([*fileid*], [*output*], [*search*], [*mask*], [*position*])

Returns directory information for *fileid*. If *fileid* contains drive or path information, the specified drive or directory is searched. Otherwise, the current directory of the current drive is searched. *Fileid* may contain wildcard characters in the name or extension, in which case the first matching file is located. If *fileid* is omitted, it is assumed that a previous call was made to DOSDIR() specifying a *fileid* with wildcard characters, and the next matching file is located.

You may want to look at some DOSDIR() examples.

A null string is returned by DOSDIR() if *fileid* was not located. Otherwise, a string containing the following is returned:

the name and extension of the file, in the form *name.ext*

the size of the file in bytes

the date of the last update of the file, in the form *mm/dd/yy*

the time of the last update of the file, in the form *hh:mm:ss*

a group of characters indicating the file's attributes:
R for a *read-only* file,
H for a *hidden* file,
S for a *system* file,
D if the file is a *directory*,
A if the file's *archive* bit is set,
and a dash (**-**) if none of the above attributes applies.

If *output* is omitted or is null, all of the above information is returned by DOSDIR(), in the above order. If specified, *output* is a string of characters that can be used to limit or reorder the returned information.
N causes the file's *name* and extension to be returned,
S returns the *size* of the file,
D the *date*,
T the *time*, and
A the file's *attributes*.

The order in which the characters appear determines the order in which the information is returned.

If *search* is omitted or is null, DOSDIR() searches only for ``normal'' files: hidden files, system files, and directory files are ignored. If specified, *search* is a set of characters that can be used to extend the search to special files:
H extends the search to *hidden* files,
S to *system* files, and
D to *directory* files.

If *fileid* is not specified and DOSDIR() is searching for the next match of a previously-specified pattern,*search* is ignored.

If specified, *mask* is a set of characters that can be used to restrict the search to files with particular attribute bits set:
R restricts the search to *read-only* files,
H to *hidden* files,
S to *system* files,
D to *directory* files, and
A to files with the *archive* bit set.

With *fileid* not specified, DOSDIR() normally searches for the next file matching the pattern that was last used. The *position* operand can be used to cause a search for the next file matching a previously used pattern, allowing intervening calls to DOSDIR() using other patterns. The DOSDIRPOS() function can be used to save the current position in a directory search.

DOSDIR (_Dir()) use examples:

(In these examples, assume that the name of a DOS file has been assigned to the variable ***f***.)

```
if dosdir(f) = '' then say f 'does not exist'}
```

DOSDIR() is frequently used to see if a file exists. (Note that in this example DOSDIR() does not search for hidden, system, and directory files.) The statement

```
say dosdir(f, 's')}
```

will output the size of the file in bytes (or a null string if the file does not exist).

```
/*   this example lists the name.ext and size for   */
/*   all files in the current directory, including */
/*   hidden, system, and directory files              */
s = dosdir('*.*', 'nsa', 'hsd')
do while s \= ""
say s
s = _dir(, 'nsa')
end
```

DOSDIRPOS() or _DirPos()

DOSDIRPOS() returns a string representing the current DOSDIR() directory position status. A later call to DOSDIR() can specify the status via the *position* argument to resume processing a directory search at the current position.

DOSDISK(option,[drive])or
_Disk(*option*, [*drive*])

Returns information about the specified DOS disk (or the current default drive, if the *drive* argument is omitted).

Available - Returns the number of *available* clusters on the disk.
Bytes - Returns the number of *bytes* per disk sector.
Clusters - Returns the number of *clusters* on the disk.
Free - Returns the number of *free* (unused) bytes on the disk.
Sectors - Returns the number of *sectors* per disk cluster.
Total - Returns the *total* number of bytes on the disk.
Used - Returns the total number of bytes in *use* on the disk.
Returns **-1** for an invalid drive, drive not ready, etc.

DOSDRIVE([newdrive]) or _Drive([*newdrive*])

The drive letter given as the first character of *newdrive* becomes the current DOS drive, and the drive letter (**A**, **B**, etc.) that was in effect before the change is returned. If *newdrive* is not specified, DOSDRIVE() simply returns the drive letter of the current DOS drive.

DOSENV(string) or
GetEnv(*string*)

Returns the value of the specified DOS environment *string*, or a null string if *string* is not defined.

DOS only:

If RXNEWCOM=YES or OPTIONS~NEWCOM is in effect, the value will come from the active copy of the environment. If RXNEWCOM=NO or OPTIONS~NONEWCOM is in effect, it will come from the master copy of the environment.

The VALUE function may also be used to examine and change the values of environment variables.

DOSENVSIZE() DOS Only

Windows

Under Windows DOSENVSIZE() will generate error 99 - function not supported under Windows.

DOS

Returns the total size of the current DOS environment area and the amount of free space remaining. Both values are in bytes. The values are returned as two numbers, separated by blanks. This is useful for determining whether there is room to add new data to the environment.

DOSFDATE(fileid, [newdate], [newtime]) or _FDate(*fileid*, [*newdate*], [*newtime*])

Changes the date and time recorded as the date/time of the last modification to the file. *Newdate* is specified in the ``standard'' date format as returned by the DATE() function: *yyyymmdd*. The date must be on or after January 1, 1980. *Newtime* is specified as *hhmmss* (24-hour format, but no punctuation). The appropriate number of digits must be specified for each field. For instance, 1:01AM is 010100. DOSFDATE() returns 1 if successful, otherwise 0.

DOSFNAME(fileid)or _FName(*fileid*)

Returns the fully qualified name of the *fileid* that you specify, including a drive specifier, path specification, and name and extension. This function is useful because it fills in any missing drive or path information in *fileid*, and it converts relative paths to absolute paths that start at the root directory.

For example, if the current drive is the C: drive and the current directory is \PROGRAMS,

DOSFNAME('abc.def')

returns 'c:\programs\abc.def',

DOSFNAME('c:object\xyz.obj')

returns 'c:\programs\object\xyz.obj', and

DOSFNAME('..\autoexec.bat')

returns 'c:\autoexec.bat'.

The fileid that you specify need not actually exist, but a null string is returned if the fileid is invalid.

DOSISDEV(fileid)or
_IsDev(*fileid*)

Returns 1 if the specified *fileid* is a device, such as CON, PRN, LPT1, NUL, CLOCK, STK, etc., otherwise 0. This may be useful for validating file names supplied to programs. DOSISDEV() returns 1 for names such as ``PRN.XYZ'', which DOS treats as the PRN device rather than a file.

DOSISDIR(fileid)or
_IsDir(*fileid*)

Returns 1 if the specified *fileid* is a directory. Returns 0 if *fileid* is a file or an invalid name. *Fileid* should not include a trailing \.

DOSMEM() or _SYSMEM()

DOS Only

Windows

Use the WINMEM() function to obtain the amount of memory available, including virtual memory under Windows Enhanced mode. Under Windows DOSMEM() will generate error 99 - function not supported under Windows.

DOS

Returns the size in bytes of the largest block of unallocated DOS memory. This indicates the size of the largest DOS program that can be called from your REXX program.

DOSMKDIR(pathname) or _MkDir(*pathname*)

Creates a new directory with the specified *pathname*, which may include drive and path information.

Returns 0 if it encountered any error conditions, or 1 if it was successful.

_Pid( ) or DOSPID()

Returns the current Process ID.

DOSPATHFIND(fileid, [envvar]) or _PathFind(*fileid*, [*envvar*])

Searches your DOS PATH for the specified *fileid*. If the file is found, its fully qualified name is returned. If the file is not found, a null string is returned.

The search process is similar to the process DOS carries out when searching for an executable file. If the fileid contains a drive or path specification, DOSPATHFIND() looks only there. Otherwise, DOSPATHFIND() looks for the file in the current directory and then in each of the directories specified in your PATH environment variable.

The second argument, *envvar*, is optional. It lets you specify the name of another environment variable, other than the default of PATH, to use to determine the list of directories to search.

DOSRENAME(fileid1,fileid2)or _Rename(*fileid1*, *fileid2*)

Renames *fileid1* to *fileid2*.

Fileid1 and *fileid2* should contain any necessary drive and path information, but should not contain wildcard characters. The DOS rename function is invoked directly, so DOSRENAME() can move files from one subdirectory to another on the same disk; with DOS 3.0 or later, DOSRENAME() can be used to rename directories.

Returns 0 if it encountered any error conditions, or 1 if it was successful.

DOSRMDIR(pathname) or _RmDir(*pathname*)

Removes the subdirectory with the specified *pathname*, which may include drive and path information.

Returns 0 if it encountered any error conditions (for example, the subdirectory was not empty), or 1 if it was successful.

DOSVERSION()or _OSVersion()

Returns the current Operating System version number (for example, 5.00).

Use WINVERSION() to obtain the current version of Windows (for example, 3.10).

DOSVOLUME([drive])or
_Volume([*drive*])

Returns the volume label of the specified *drive*, or of the current DOS drive if *drive* is omitted. Returns null if the label cannot be located (for example, the disk has no label, drive not ready).

Hardware Access Group}

\index{Extensions;hardware functions}\index{Keyboard functions}%

\index{Display functions}\index{Functions, hardware access}%

CHARSIZE() DOS Only

Windows

Under Windows CHARSIZE() will generate error 99 - function not supported under Windows.

DOS

Returns the height of the character box in scan lines for the current screen mode. This can be used to provide appropriate values for the CURSORTYPE function.

CURSOR([row],[col]) DOS Only

Windows

Under Windows CURSOR() will generate error 99 - function not supported under Windows.

DOS

Sets the cursor position to the specified new *row* and *column*.

If only *row* or only *column* are given, only the row or column position of the cursor is changed; if both are omitted, the cursor position is not changed. The initial row and column position of the cursor, before any change, is returned by the CURSOR function.

If the cursor is currently in row 6, column 15,

c = cursor()

sets *c* to '6 15' and does not affect the cursor location, while

c = cursor(10, 22)

sets *c* to '6 15', and moves the cursor location to row 10, column 22.

\index{SCREEN.REX}

The file SCREEN.REX ... is a sample program illustrating the use of the SCRWRITE(), INKEY(), and CURSOR() functions.

\end{describe}

CURSORTYPE([start],[end]) DOS Only

Windows

Under Windows CURSORTYPE() will generate error 99 - function not supported under Windows.

DOS

Returns the current cursor type, and possibly sets new values for the cursor type. The cursor type is a pair of numbers indicating the *start*ing and *end*ing rows within a character box used for display of the cursor. For the monochrome adapter, each character box is 13 pixels high, the rows are numbered 0 through 12, and the cursor is normally displayed in rows 11 and 12. For the color adapter, each character box is 8 pixels high, the rows are numbered 0 through 7, and the cursor is normally displayed in rows 6 and 7.

For example, if you are using a color display with the cursor currently in rows 6 and 7,

```
OLD\_TYPE = CURSORTYPE(1, 7)
```

would give you a large block cursor occupying rows 1 to 7, with OLD_TYPE set to `6 7'.

The CHARSIZE() function can be used to get the height of the character box for the current video mode.

DELAY(duration)

Causes your REXX program to delay for a specified *duration*, which you specify in seconds. Under Windows the DELAY() routine is accurate to within less than a second.. Under DOS the DELAY() routine is accurate to within about a tenth of a second. For example,

```
CALL DELAY 5
```

will delay for approximately 5 seconds.

INKEY([wait_option], [keyboard_option]) DOS Only

Windows

Under Windows INKEY() will generate error 99 - function not supported under Windows.

DOS

Reads in a character directly from the keyboard (without echoing it to the display). In most cases, the result is returned as a string of length 1. If a special key is hit (function key, PgUp, etc.), a 2-character string is returned: The first character is normally '00'x, and the second character is the scan code for the key that was hit, as documented in the IBM BIOS Interface Technical Reference.

Note: The first character may be 'E0'x for some Enhanced Keyboard key combinations.

Wait_option , which defaults to 'Wait', can be:

Wait

INKEY() waits until a character is ready.

Nowait

If no character is ready, INKEY() will immediately return a null string.

Keyboard_option, which defaults to 'Fold', can be:

Enhanced

Reads from the keyboard using Enhanced Keyboard BIOS calls, if available on your machine, so that Enhanced Keyboard keys such as F11 and F12 can be processed.

Fold

Like Enhanced, except that key codes are ``folded'' so that analogous keys on the numeric and dedicated keypads have the same codes as they did on the original IBM PC.

Old

Uses old style BIOS calls, as supported on the original IBM PC, to read from the keyboard even if Enhanced Keyboard BIOS support is available.

The PCKEYBOARD function can be used to determine whether an enhanced keyboard is present.

INP(port)

Reads 1 byte from the specified hardware *port* (which must be in the range 0 to 65535).

OUTP(port,value)

Writes 1 byte to the specified hardware *port* (which must be in the range 0 to 65535).

Value must be in the range 0 to 255. OUTP() always returns a value of 0.

PEEK(segment,offset) DOS Only

Windows

Under Windows PEEK() will generate error 99 - function not supported under Windows.

DOS

Segment and *offset* must be in the range 0 to 65535. Returns the value (as a number from 0 to 255) contained in the specified location of your PC's memory.

POKE(segment,offset,value) DOS Only

Windows

Under Windows POKE() will generate error 99 - function not supported under Windows.

DOS

Segment and *offset* must be in the range 0 to 65535. Stores *value* (which must be in the range 0 to 255) in the specified location of your PC's memory. POKE() always returns a value of 0.

SCRBLINK([state]) DOS Only

Windows

Under Windows SCRBLINK() will generate error 99 - function not supported under Windows.

DOS

Returns the current state of video attribute handling. If the state is 1, then video attributes (as described in connection with the SCRWRITE() function) above 128 produce blinking text. If the state is 0, such attributes produce a bright background color. This state may be changed by supplying a value of 0 or 1 for *state*.

SCRCLEAR([attr],[char],[row],[col],[height],[width]) DOS Only

Windows

Under Windows SCRCLEAR() will generate error 99 - function not supported under Windows.

DOS

Clears the screen, or any rectangular region of the screen. The cleared area can be filled with any desired character and display attribute. Always returns 0.

Calling SCRCLEAR() with no arguments will clear the screen by filling it with blanks at the default attribute of 7.

> *Attr* is the display attribute to use in the cleared area. The default is 7. (See the description of the SCRWRITE() function for a discussion of valid attribute values.)
>
> *Char* is the character to use to fill the area. The default is a blank.
>
> *Row* and *col* define the row and column on the screen of the upper left corner of the rectangular area. The default for each is 1.
>
> *Height* and *width* define the height and width of the area to be cleared.

The defaults are the number of rows and columns remaining on the screen.

SCRMETHOD([method]) DOS Only

Windows

Under Windows SCRMETHOD() will *always* return **W** (Windows) and may not be changed. *Method* if specified will be ignored.

DOS

Returns the existing screen method and optionally allows you to change the screen method.

Method can be either **D** ('Direct'), **R** ('Retrace'), or **B** ('Bios'). It is the method used to access the screen by the SCRREAD(), SCRWRITE(), and SCRPUT() functions.

> With the **D**irect method, the fastest method, the display buffer is accessed directly.
>
> With the **R**etrace method, REXX accesses the display buffer directly, but is somewhat slower because it accesses the buffer only during the retrace interval, as is required for the IBM Color Graphics Adapter.
>
> The **B**ios method uses the BIOS to access the display buffer. This is much slower than the other two methods, but may be required in some specialized situations.

The default screen method is **D**irect if you have a Monochrome Display Adapter, EGA, or VGA. Otherwise, the default screen method is **R**etrace.

SCRPUT(row,col,string,[option]) DOS Only

Windows

Under Windows SCRPUT() will generate error 99 - function not supported under Windows.

DOS

Writes contents of *string* to screen, starting at *row* and *col*umn.

Option can be:

T (**T**ext), the default, *string* has text to be written, attributes don't change;
A (**A**ttribute) *string* has attributes to be written, text doesn't change;
B (**B**oth) *string* has character-attribute pairs. The length of *string* must be even; LENGTH(*string*)/2 screen positions are written.

Always returns a null string as its value.

SCRREAD(row,col,length,[option]) DOS Only

Windows

Under Windows SCRREAD() will generate error 99 - function not supported under Windows.

DOS

Returns *length* screen positions, starting at given *row* and *col*umn.

Option can be:

T (**T**ext), the default, to read text only from the screen;
A (**A**ttribute) to read attributes only;
B (**B**oth) to read both (returns 2 x *length* bytes).

SCRSIZE() DOS Only

Windows

Under Windows SCRSIZE() will generate error 99 - function not supported under Windows.

DOS

Returns the number of rows and columns on your PC's screen. The result is normally '25 80', but may be different if you have an EGA or VGA and have a utility that can change the number of rows and/or columns.

SCRWRITE([row],[col],[string],[length],[pad],[attr]) DOS Only

Windows

Under Windows SCRWRITE() will generate error 99 - function not supported under Windows.

DOS

Writes *string* to the screen at the specified *row* and *col*umn. *String* is truncated (or padded with *pad*) to the given *length*. Characters are written to the screen using the attribute *attr*. SCRWRITE() does not affect the current cursor position, and always returns 0 as its value.

If *row* and/or *col* are omitted, current row and/or column positions of the cursor are used.

The default for *string* is the null string.
The default for *length* is the length of *string*.
The default for *pad* is a blank.
The default for *attr* is 7.

Some examples of SCRWRITE:

\begin{describe}{{\tt call scrwrite , , 'hello'}}

writes \literal{hello} at current cursor position.

\end{describe}

\begin{describe}{{\tt call scrwrite 1, 1, 'hello'}}

writes \literal{hello} starting in row 1, column 1.

\end{describe}

\begin{describe}{{\tt call scrwrite 1, 1, 'hello', 80}}

writes \literal{hello} starting in row 1, column 1, and clears the rest of the line.

\end{describe}

\begin{describe}{{\tt call scrwrite 1, 1, , 2000}}

clears the screen.

\end{describe}

\begin{describe}{{\tt call scrwrite 1, 1, , 2000, '+', 99}}

fills the screen with plus signs, using attribute 99.

\end{describe}

With the IBM monochrome display, the attribute values that work best for {\em attr\/} are 7 (normal display), 15 (highlighted), 65 (underlined), and 112 (reverse video).

With the IBM color display, find the foreground color and background colors that you want in the table that follows, and add them together. For example, to get yellow text on a red background, add together 14---the value for yellow as a foreground color---and 64---the value for red as a background color---to get 78.

If blinking attributes are enabled, as they are by default, adding 128 will produce blinking text. If blinking attributes are disabled with the SCRBLINK function, adding 128 will produce text with bright background colors.

\index{Attributes for color display with SCRWRITE function}%

\index{Color display attributes for SCRWRITE function}

\begin{samepage}

\begin{tabbing}

xxxxxxxxxx\=xxx\=xxxxxxxxxxxxxxxxxxxxxxxxx\=xxx\= \kill

\> {\bf Foreground Colors} \> \> {\bf Background Colors} \\

\> \> \ 0 black \> \> \ \ 0 black \\

\> \> \ 1 blue \> \> \ 16 blue \\

\> \> \ 2 green \> \> \ 32 green \\

\> \> \ 3 cyan \> \> \ 48 cyan \\

\> \> \ 4 red \> \> \ 64 red \\

\> \> \ 5 magenta \> \> \ 80 magenta \\

\> \> \ 6 brown \> \> \ 96 brown \\

```
    \> \>   \ 7 white                    \>   \>         112 white                \\
    \> \>   \ 8 gray                                                             \\
    \> \>   \ 9 light blue                                                      \\
    \> \>     10 light green                                                     \\
    \> \>     11 light cyan                                                      \\
    \> \>     12 light red                                                      \\
    \> \>     13 light magenta                                                    \\
    \> \>     14 yellow                                                          \\
    \> \>     15 high intensity white
\end{tabbing}
\end{samepage}
```

SHIFTSTATE(key, [state]) DOS Only

Windows

Under Windows SHIFTSTATE() will generate error 99 - function not supported under Windows.

DOS

Returns the current shift state of the NumLock, CapsLock, and ScrollLock keys. The shift state can be changed by supplying a value of 0 or 1 for *state*. *Key* may be specified as:

C - **C**apsLock key
N - **N**umLock key
S - **S**crollLock key

SOUND([freq],[duration])

Windows

Under Windows SOUND() will always play the sound associated with the **Default Beep** event. Use WINSOUND() to play sound files associated with Windows sound events.

DOS

Sounds the PC's speaker. *Freq* is in cycles per second, gets rounded to the nearest integer, and defaults to 880. *Duration* is in seconds and is accurate to around a tenth of a second, with .2 seconds as the default. For example,

```
CALL SOUND 1000, .5
```

will sound the PC's speaker at 1000 cycles per second for approximately .5 seconds.

Miscellaneous Group

DATECONV(date,input,[output])

Converts the string *date*, which is in *input* format (**B**, **C**, **D**, **E**, **J**, **N**, **O**, **S**, or **U**), to *output* format (**B**, **C**, **D**, **E**, **J**, **M O**, **S**, **U**, or **W**). The formats are the same as those used for the DATE() built-in function.

Default *output* format is **N**. Dates which are not in proper input format convert to a null string.

EMSMEM() DOS Only

Windows

Under Windows EMSMEM() always returns 0.

DOS

Returns, as two words, the number of bytes of Lotus-Intel-Microsoft EMS memory currently unallocated and the total number of bytes of EMS memory on your system.

ENDLOCAL()

Windows

Restores only the drive/directory variables saved by the last SETLOCAL() call. Returns 1 if successful, otherwise 0.

DOS

Under DOS ENDLOCAL() also restores the current DOS environment variables saved by SETLOCAL.

FCNPKG(package-name) DOS Only

Windows

Under Windows FCNPKG() always returns 0.

DOS

Result is 0 if *package-name* is not loaded, 1 if loaded. Packages are sets of optionally loadable REXX functions.

Result is 2 if *package-name* is loaded and a resident environment.

LOWER(string)

Lowercases the specified *string*.

PARSEFN(fileid)

Breaks the given *fileid* into four components, returned as a string consisting of four uppercase words:

1. drive specifier (without a colon),
2. path specifier,
3. file name,
4. file extension.

Omitted components are returned as `**-**'. For example,

```
x = parsefn('abc\def.ghi')
```

sets *x* to '- \ABC\ DEF GHI'.

A null string is returned if *fileid* contains invalid characters or invalid components (for example, an extension longer than three characters). Note that the checking is purely syntactic, and *fileid* may refer to a file, drive, or directory that does not exist.

PRXSWAP() DOS Only

Windows

Under Windows PRXSWAP() always returns 0.

DOS

Under DOS the PRXSWAP() function controls when and how swapping is be done.

PRXVERSION()

PRXVERSION() returns the current Enterprise REXX version number, for instance 3.00. (This is distinct from the REXX language version number provided by PARSE VERSION.)

SETLOCAL()

Windows

Saves only the current drive/directory variables. When used with ENDLOCAL(), this function permits arbitrary changes to be made to the current drive/directory variables and allows the original state to be restored easily. Returns 1 if successful, otherwise 0.

DOS

Under DOS SETLOCAL() also saves the current DOS environment variables.

STACKSTATUS()

Returns information about the status of the console stack.

Windows

Under Windows STACKSTATUS() always returns 'E 0 1000' (Enabled, 0th buffer, 1000 chars free).

DOS

Under DOS STACKSTATUS() returns a string consisting of three items:

1. the character **E** (stack **E**nabled), **D**(stack **D**isabled), or **N** (stack **N**ot installed),
2. the number of the current stack buffer,
3. the remaining capacity of the stack, in characters.

For example, if STACKSTATUS() returns 'E 1 443', the stack is enabled, you are processing stack buffer 1, and the stack has room for up to 443 additional characters.

UPPER(string)

Uppercases the specified *string*.

VALIDNAME(fileid, [wildcard])

Returns 1 if *fileid* is a valid file name. If *fileid* includes a drive letter, it must be a valid drive, and the name syntax is checked according to the rules of the file system on the specified drive. Otherwise the syntax is checked according to the rules of the file system on the current drive. Names are tested to verify that all parts of the name do not exceed allowed lengths and that characters not allowed in names are not present. Directory names included in the *fileid* do not need to exist.

If the *wildcard* flag is 1 then ``**?**'' and ``*****'' are allowed in file names (but not paths). If it is 0 (the default) then ``**?**'' and ``*****'' cannot be used anywhere.

WinSetKeyWindow(*WindowTitle*)

NOTE: This function is obsolete and provided only for existing REXX program compatibility. This functionality is built into WinSendKeys(), or use FindWindow ().

The WinSetKeyWindow() function names the window to receive keystrokes to be sent by the WinSendKeys() function.

Arguments:

WindowTitle specifies the title of the window to receive the keystrokes.

Returns:

A positive number if the function is successful, or 0 if the window named by *WindowTitle* cannot be found. The number returned if the function is successful is the handle of the found window.

Notes:

The *WindowTitle* must be the name **exactly** as shown in the window's title bar.

FindWindow(*WindowTitle*)

The FindWindow() function find a top level window with the specifid WindowTile in it's title bar.

Arguments:

WindowTitle The title of the window to be found.

Returns:

A positive number if the function is successful, or 0 if the window named by *windowtitle* cannot be found. The number returned is the handle of the found window. This handle can be used in other functions such as WinPostMessage() ,WinSendMessage() and WinSendKeys().

Notes:

The *WindowTitle* must be the name **exactly** as shown in the window's title bar. See the example below to find an empty Notepad window.

Example:

To find an emply Notepad window use:

```
FindResult = FindWindow ( "Notepad - (Unititled)" )
if FindResult = 0
signal NoWindow
```

WinSendKeys(*keystring*, [*windowtitle*])

WinSendKeys () sends the keys specified in keystring to the receiving window. If windowtitle is not provided, then the keys are sent to the window named in a previous call to WinSendKeys().

Arguments:

keystring	a string of keystrokes to be sent to a window, including send key codes to specify non-character control keys.
windowtitle	specifies the title of the window to receive the keystrokes. This must be the name **exactly** as shown in the window's title bar.

Returns:

Returns 0 if the function is successful.

Notes:

A returned value not equal to zero may be a message indicating the cause of the failure. The receiving window name must be provided in the first call to WINSENDKEYS(). After the receiving window is specified, keystrings are sent to that window until the windowtitle value is changed.

Example:

To display the File Manager About dialog box send the sequence <ALT+H+A> as follows:

```
SendResult = WINSENDKEYS("%ha", "File Manager" )
```

To have Program Manager run Notepad on a file you could use the following:

```
RC = WINSENDKEYS("%FRNOTEPAD c:\config.sys~","Program Manager")
```

Send Key Codes

WinSendKeys() uses the following conventions to send keys to a window. Each key is represented by one or more characters. There are several classes of keys:

Normal Keyboard characters.
Keys combined with Shift, Ctrl, and Alt.
The keys plus (+), caret (^), and percent (%).
Characters that are not displayed such as Home or Delete.
Repeating keys.

Normal Keyboard Characters

To Specify a single keyboard character, **use the character itself**. For example, to represent the letter a, use 'a'. To represent more than one character, append each additional character to the one before. To represent the letters a, b, and c, use 'abc'.

Keys Combined with Shift, Ctrl, and Alt

To specify keys combined with any combination of Shift, Ctrl, and Alt, precede the regular key code with one or more of these codes:

Shift	**+**
Ctrl	**^**
Alt	**%**

To specify that Ctrl, Alt and/or Shift should be held down while several keys are pressed, enclose the keys in parentheses. For example, to hold down the Shift key while pressing E then C, use +(EC). To hold down Shift while pressing E, followed by C without the Shift key, use +EC.

Because the plus (+) sign, caret (^), and percent sign (%) have special meanings, **to specify one of these special characters**, enter the character inside braces. For example, to specify the plus sign, use {+}. To send a { character or a } character, use {{} and {}}, respectively.

Characters that are not displayed

To specify characters that are not displayed when you press a key (such as Enter or Tab) and other keys that represent actions rather than characters, use the codes shown below:

Backspace	{backspace} or {bs} or {bksp}
Break	{break}
Caps Lock	{capslock}
Clear	{clear}
Del	{delete} or {del}
↓	{down}
End	{end}
Enter	{enter} or ~
Esc	{escape} or {esc}
Help	{help}
Home	{home}
Insert	{insert}
←	{left}
Num Lock	{numlock}
Page Down	{pgdn}
Page Up	{pgup}
Print Screen	{prtsc}
→	{right}
Scroll Lock	{scrollock}
Tab	{tab}
	{up}
F1 to F16	{f1} to f16

Repeating Keys

To specify repeating keys, use the form *{key number}* where there is always a space between key and number. For example, {left 42} means press the left arrow key 42 times; {x 10} means press the character x 10 times.

Enterprise REXX Mail Access Functions.

The functions below allow Enterprise REXX programs to be clients (users) of MS Mail on Windows NT and Windows 95. Mail support is not provided for the Win16 version of Enterprise REXX. Choose from the following list to review the area of interest:

MailLogon() - Log on to the Mail System.
MailSend() - Send a mail message.
MailFind() - Find a mail message.
MailRead() - Read a mail message.
MailDelete() - Delete a mail message.
MailSendDocuments() - Send files and prompt for a message.
MailLogoff() - Log off of the Mail System.

Note: To register these MAIL functions automatically at Enterprise REXX startup, place the following entry in a “Register.INI” file located in the same directory where REXX is installed.

```
[Register]
RxMail=1
```

MailRC = MailLogon(*MailID*, *PassWord*)

MailLogon(***MailID*****,** ***PassWord, MailSystem*** **)** logs Rexx onto the mail system as a “user”. Rexx must be logged on before other mail functions can be used.

Arguments:

MailID A valid ID on the Mail system being used.
PassWord The current password for *MailID*.
MailSystem Is currently unused. A default MAPI system is assumed.

Returns:

MailLogon() returns zero (0) if it is successful and non-zero if it fails.

If MailLogon() is not successful, the return value begins with the sub-string "Error:" and contains an explanation of the failure.

See also:

MailLogoff

MailRC = MailLogoff( )

MailLogoff( ) logs Rexx off of the mail system it is currently logged onto as a “user”.

Arguments:

None.

Returns:

MailLogoff() returns zero (0) if it is successful and non-zero if it fails.

If MailLogoff() is not successful, the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailLogon

MailRC = MailSend(*To, Subject, MsgText*)

MailSend( ) sends a mail message.

Arguments:

To The Email ID to which the message is to be sent, e.g., Santa@NorthPole.com or a name on your mail system.

Subject The subject line for the message to be sent.

MsgText The text of the message to be sent.

Returns:

MailSend() returns zero (0) if it is successful and non-zero if it fails.

If MailSend() is not successful, the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailLogon
MailLogoff

MailRC = MailSendDocuments(*FilePaths, FileNames*)

MailSendDocuments( ) sends files as attachments to a mail message. A mail system Send Note dialog box will be displayed prompting the user to enter the message text and it's destination.

Arguments:

FilePaths A list of fully qualified file name paths, separated by semi-colons. This identifies the files to be sent.

FileNames A corresponding list of file names to be sent in a mail message, separated by semi-colons. These names will be shown in the message, and their extension will determine the ICONs shown in the message and the application to be launced when activated.

Returns:

MailSendDocuments () returns zero (0) if it is successful and non-zero if it fails.

If MailSendDocuments() is not successful, the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailSend

MailRC = MailFind( )
MailRC = MailFindNext( )
MailRC = MailFindFirst( )

MailFind( ) locates a mail message in the InBOX of the mail system it is currently logged onto as a “user”.

If this is the first call to any **MailFind*( )** function then **MailFind( )** is equivalent to **MailFindFirst( )**.

If this is **not** the first call to any **MailFind*( )** function then **MailFind( )** is equivalent to **MailFindNext( )**.

MailFindNext( ) locates **the message after** the current mail message previously found with **MailFindFirst( )** or **MailFindNext( )** or **MailFind( )**.

MailFindFirst( ) locates **the first** mail message in the InBOX.

Arguments:

None.

Returns:

The MailFind*() functions return zero (0) if a message is successfully found and non-zero Otherwise.

- If no message is found, the MailFind*() functions return "No More Messages".
- Otherwise the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailRead

MailRC = MailRead(*StemName*)

MailRead( ) reads a mail message already found with one of the **MailFind*( )** functions and places it's contents into the "***StemName*.**" Rexx **stem variable**. If "***StemName*.**" is null or not provided, then the name "***NewMail*.**" will be used.

***StemName*.To** will contain the name from the message TO: field.

***StemName*.From** will contain the name from the message FROM: field.

***StemName*.Date** will contain the Date. The US English form is yy/mm/dd hh:mm which can be parsed with the following REXXX statement:

```
parse var NewMail.Date year"/"month"/"day hour":"minute.
```

***StemName*.Subject** will contain the text from the message SUBJECT: field.

***StemName*.Text.0** will contain the count of the number of message text lines found.

***StemName*.Text.n** will contain lines of message text where **n** ranges from 1 to the value stored in *StemName*.Text.0.

The following REX statements would display the content of a just read message:

```
ReadRC = MailReadMail( )
if ReadRC = AOK then call NEWMAIL_SAY
.
.
NEWMAIL_SAY:
      say NewMail.To
      say NewMail.From
      say NewMail.Date
      say NewMail.Subject
      do i = 1 to NewMail.Text.0
        say NewMail.Text.i
return AOK
```

Arguments:

None.

Returns:

The MailRead() function returns zero (0) if it is successful and non-zero if it fails.

If not successful, the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailFind

MailRC = MailDelete( )

MailDelete( ) deletes from the mail system a mail message already found with one of the **MailFind*( )** functions.

Arguments:

None.

Returns:

The MailRead() function returns zero (0) if it is successful and non-zero if it fails.

If not successful, the return value begins with the sub-string "Error:" or the sub-string "Warning:" and contains an explanation of the failure.

See also:

MailFind