The Regina Rexx Interpreter – Rexx Utility Functions

Patrick TJ McPhee (ptjm@interlog.com)
DataMirror Corporation

Version 1.1.9, 1 January 2002

Contents

1 Introduction

This paper describes an implementation of IBM's Rexx Utility functions for Unix and Windows/NT. The Rexx Utility functions extend the capabilities of Rexx programs in useful ways, and it's desirable to have compatible interfaces available for all implementations. This implementation is meant for the Regina interpreter, however it should also work with other interpreters. An earlier version of the library was binary compatible with Rexx/IMC on Solaris.

Although I have access to the rexxutil source code through the OS/2 developer's kit, this implementation contains new code, and is based on the documentation for IBM's implementation, rather than its internals. The routines here are generally compatible with IBM's, although some IBM functions or options are not implemented, and in some cases, the regutil version provides minor enhancements. Please see the file status.txt in the distribution for claims of completion.

The manual, and the source code, are divided along the lines of functionality, the major groupings being housekeeping (regutil.c), file system (regfilesys.c), system (regini.c), macro-space manipulation (regmacrospace.c), stem manipulation (regstem.c), console I/O (regscreen.c, regscreenux.c), and semaphore handling (regsem.c, regsemux.c).

1.1 Reporting Bugs

Theorem A: Every program can be reduced by at least one line.

Theorem B: Every program contains at least one bug.

Corollary: Every program can be reduced to one line which doesn't work correctly.

Regutil undergoes very little testing before new releases are shipped. I have not had the time to produce a regression test, for instance, and although it is on my list of things to do, the pressures of work and life keep me from doing it. Since the first 'full' release of Regutil (1.0.4) in February 1999, there have been surprisingly few bugs discovered, given the amount of testing it undergoes at this end. When bugs are reported, I do my best to fix them and to get a new release out within a short time. My time tends to be very tight, though, so I can't make any guarantees.

If you do find a bug, an error in the documentation, or you simply have a suggestion for improving the distribution, please send me details at ptjm@interlog.com. It's useful to know the operating system you're using, the version of Regina (or Rexx/IMC), and the version of regutil, and to have a set of steps for reproducing the bug.

If you are using regutil for a serious purpose and therefore take the time to produce a test suite for your own use, I would appreciate it if you'd contribute it to the cause.

1.2 Using RxAddFunc

All the routines in RegUtil can be loaded either directly using RxFuncAdd, or indirectly using SysLoadFuncs. One difference between Regina and OS/2 Rexx is that Regina requires that the case of the function name passed to RxFuncAdd match the case of the

function name in the shared library (which is always lower-case in RegUtil). The effect of this is that existing code which looks like this

```
call rxfuncadd 'SysFileTree', 'RexxUtil', 'SysFileTree'
```

will fail, and should be re-written like this

```
call rxfuncadd 'sysfiletree', 'rexxutil', 'sysfiletree'
```

Also note that Regina includes two executables, one called 'rexx', and the other called 'regina'. The difference is that 'rexx' includes the Rexx interpreter as part of the executable, while 'regina' loads the interpreter from a shared library. RxFuncAdd works only with the 'regina' version of the interpreter (the 'rexx' version is slightly faster, though).

1.3 Licencing

Regutil is distributed free of charge in the hopes that it will be useful, but without any warranty. Previous versions of the library have been distributed under the terms of the GNU Library General Public License. This version is distributed under the terms of the Mozilla Public License. The precise details of the licence are found in the file MPL-1.0.txt in the distribution.

If you use the library purely as distributed by me, then you can cheerfully ignore the licencing change. If you modify the source code or adopt portions of it in your own programs or libraries, you should be aware of and fulfill your obligations under the licence. I believe that the restrictions placed by the Mozilla licence are less onerous than the ones in the GNU Library licence, and they are more in the spirit in which I would like my work to be distributed.

Although there are no obligations or restrictions related to use of the library, I would prefer that you do not use regutil in applications which cause injury or hardship to others. Also, if you derive a significant monetary benefit from the use of regutil, please share a portion with someone less fortunate. For instance, if you save $10,000 by implementing an application with Regina and regutil, rather than buying a commercial Rexx interpreter, you could give $1,000 to Unicef.

2 Housekeeping Routines

These are routines which help you use the other routines.

2.1 SysLoadFuncs

```
sysloadfuncs() -> 0
```

SysLoadFuncs registers all the other routines in the utility package with the Rexx interpreter. This registration takes less work on your part than registration using rxfuncadd, and it's probably faster to use sysloadfuncs whenever you need more than one utility function, plus it's less typing.

2.2 SysDropFuncs

```
sysdropfuncs() -> 0
```

SysDropFuncs removes the registration of all the utilities in the package from the Rexx interpreter. I don't feel there's a compelling reason for doing this, and it has the potential to be positively harmful in the IBM interpreters, since they don't do proper reference counting for load/drop. It's safe to call SysDropFuncs even if you didn't load all the functions using SysLoadFuncs.

3 File System Routines

The file system routines manipulate files and directories in useful ways.

3.1 List of File System Routines

SysCopyObject	(from,to) → 0 or failure: copies a file
SysCreateShadow	(from,to) → 0 or failure: creates a link to a file
SysFileDelete	(file) → 0 or failure: deletes a file
SysFileSearch	(target,file,stem, [options]) → 0 or failure: searches a file for some text
SysFileSystemType	(file) → string: returns the name of the file-system in use for file
SysFileTree	(filespec,stem, [options], [tattrib], nattrib]) → 0 or failure: search for files matching filespec
SysMkDir	(directory) → 0 or failure: creates a new directory
SysMoveObject	(from,to) → 0 or failure: renames a file
SysRmDir	(directory) → 0 or failure: removes a directory
SysSearchPath	(var,file) → full filename: searches a list of directories from an environment variable for a file
SysTempFileName	(template, [filter]) → name: returns a temporary name based on a template;
SysGetFileDateTime	(name[, which]) → timestamp: returns a timestamp for a file;
SysSetFileDateTime	(name[, date[, time]]) → 0 or failure: sets the modification timestamp for a file;

3.2 Example

Here's a script which creates a directory with a temporary name, finds the file rgb.txt somewhere on the system, searches it for all different kinds of blue, and writes them out to another temporary file in the temporary directory.

```
call rxfuncadd 'sysloadfuncs', 'rexxutil', 'sysloadfuncs'
call sysloadfuncs

/* make a temporary directory */
dir = SysTempFileName('dir?????')
call SysMkDir dir

/* find an rgb.txt out there somewhere */
if SysFileTree('/rgb.txt', 'RGB.', 'SO') = 0 & rgb.0 > 0 then do
  if SysFileSearch('blue', rgb.1, 'BLUE.') = 0 then do
    file = SysTempFileName(dir'/blue?????')
    do i = 1 to blue.0
      call lineout file,blue.i
      end
    end
  end

 exit 0
```

3.3 SysCopyObject

```
SysCopyObject(from,to) -> 0 or failure
```

Copies the file named by *from* to a new name *to*. The access and modification times are preserved, for file systems that maintain that sort of arcane information. Under OS/2, this file will also copy workplace shell objects. Obviously, that doesn't work on other systems. Returns 0 on success. See SysFileDelete, section 3.5, for the meanings of the non-zero failure codes.

3.4 SysCreateShadow

```
SysCreateShadow(from,to) -> 0 or failure
```

Under Unix, SysCreateShadow creates a link to the *from* file, under the name *to*. If possible, this is a hard link, but a symbolic link is made if necessary (*e.g.*, if the *from* and *to* directories are on different devices). Note that the symbolic link will not work correctly unless the *from* file is specified as a full path, so it's best to specify full paths if the potential to cross devices exists.

Under NT, I intend for SysCreateShadow to create short-cuts. Currently, to create a short-cut, you can use my other package, w32funcs.

Returns 0 on success. See SysFileDelete, section 3.5, for the meanings of the non-zero failure codes.

3.5 SysFileDelete

```
SysFileDelete(file) -> 0 or failure
```

Deletes the file specified by *file*. Returns 0 on success. On failure, it returns 1 (unknown cause of failure), 2 (no such file), 3 (path to file does not exist), 5 (insufficient rights), 32 (file in use by another process), 36 (too many symbolic links on the way to the file), 87 (invalid file name), and 206 (file name is too long).

3.6 SysFileSearch

```
SysFileSearch(target,file,stem, [options]) -> 0 or failure
```

Searches the file *file* for the text in *target*, and puts the matching lines in *stem*. *Stem.0* receives a count of the lines, and lines are indexed sequentially starting with *stem.1*. By default, the search is case-insensitive, however it can be made case sensitive by specifying the option 'c'. The other option 'n' causes each output line to have the line number in *file* to be prepended to it.

The function returns 0 on success. On failure, it returns 2 (insufficient memory) or 3 (could not open *file* for reading).

3.7 SysFileSystemType

```
SysFileSystemType(file) -> string
```

Returns the file system type used for the specified file. For NT, this should be just the drive letter and a colon, although it may be a full path to a file (I believe this is an enhancement over IBM's implementation, so be careful). For Unix, it should be the full path to the file of interest.

On success, returns the file system name if it could be determined (or 'UFS' if it couldn't be), or the empty string if the drive or mount point is not accessible.

3.8 SysFileTree

```
SysFileTree(filespec,stem, [options], [tattrib],
            [nattrib]) -> 0 or failure
```

Finds all files whose names match *filespec*, and writes their names into *stem*. *filespec* may contain wild-card characters such as are normally allowed on the platform in question. On NT, I believe this limits you to '*', which matches 0 or more occurances of any character, and ?, which matches exactly one occurance of any character. On Unix, the glob() routine, which usually follows the rules of /bin/sh, is used. This allows character classes ([a-e] matches any letter from a to e, for instance), and probably other features which don't come to mind. If *filespec* ends with a directory separator character (slash on Unix, back-slash or slash on NT), the routine acts as if the pattern had ended with '*'.

options controls how directories are searched, and how the output is delivered. By default, the output is the time-stamp, size, file attributes, and full path to the file, for

every file and directory which matches *filespec*. If *options* includes an 'f', only files are reported. If it includes a 'd', only directories are reported. If it contains 'b', both files and directories are reported. If more than one of these is given, the right-most one wins. If *options* contains 's', SysFileTree searches sub-directories for files matching *filespec*. If it includes 'o', only the full path to the file is reported. If it includes 't' time-stamp is returned in the format yyyy/mm/dd/hh/mi. Note that IBM's implementation uses 2-digit years. If *options* includes 'l', the time-stamp is returned in the format yyyy-mm-dd hh:mi:ss, which can be processed with the Rexx date() and time() functions.

On Unix systems, the attributes are returned in the usual format from ls -l: the first byte is '-' for a normal file or 'd' for a directory, and it is followed by three 'rwx' pairs indicating the read, write, and execute permissions for each of the user, group, and others. On NT, the attributes are ADHRS, matching the archive, directory, hidden, read-only, and system bits.

tattrib allows the user to specify the file attributes which should be matched. For each position in ADHRS, '*' means match regardless of the state of the bit, '+' means to match if it's set, and '-' means to match if it's not set. Thus '*-*+*' would match all non-directories with the read-only bit set. For Unix systems, the DOS attribute positions are preserved, but they're given meanings which are specific to this implementation:

Attribute	*'+' match*	*'-' match*
A	files with more than one hard link	files with exactly one hard link;
D	files with execute permission	files without execute permission;
H	files without read permission	files with read permission;
R	files without write permission	files with write permission;
S	files with owner id less than ('system' files)	files with owner greater than or equal to 10.

In IBM's implementation, *nattrib* uses the same scheme to specify how these attributes should be changed by this function (which otherwise has no effect on its environment). *nattrib* is not supported by this implementation.

3.9 SysMkDir

```
SysMkDir(directory) -> 0 or failure
```

Creates a sub-directory with the specified name. On Unix systems, this is created with the permissions rwxr-xr-x, masked with the value of the process's umask.

On success, returns 0. On failure, it returns 1 (unknown cause of failure), 2 (no such file), 3 (path to file does not exist), 5 (insufficient rights, quota exceeded, or the directory already exists), 36 (too many symbolic links on the way to the file), 87 (invalid file name), 108 (file system is read-only), and 206 (file name is too long).

3.10 SysMoveObject

```
SysMoveObject(from,to) -> 0 or failure
```

SysMoveObject renames the file *from* to *to*. If these files are on different devices, the file is copied and then the original file is deleted, otherwise only the diretory entries are manipulated.

Under OS/2, this function will also move workplace shell objects. Obviously, that doesn't work on other systems. Returns 0 on success. See SysFileDelete, section 3.5, for the meanings of the non-zero failure codes.

3.11 SysRmDir

```
SysRmDir(directory) -> 0 or failure
```

Removes the sub-directory with the specified name.

On success, returns 0. On failure, it returns 1 (unknown cause of failure), 2 (no such file), 3 (path to file does not exist), 5 (insufficient rights, quota exceeded, or the directory already exists), 16 (some other process is using the directory), 36 (too many symbolic links on the way to the file), 87 (invalid file name), 108 (file system is read-only), and 206 (file name is too long).

3.12 SysSearchPath

```
SysSearchPath(path,file) -> full filename
```

Searches the list of directories specified by *path* for the file specified by *file*. Each element of the path is separated by the usual path separator for the platform (*e.g.*, ':' on Unix and ';' on NT). The full file name must be specified (that is, 'regina.exe' rather than just 'regina').

On success, returns the full path to the file. On failure, returns the empty string.

3.13 SysTempFileName

```
SysTempFileName(template, [filter]) -> name
```

Given a prototype filename *template*, with up to five wild-card characters, return a the name of a file which does not already exist, replacing the wild-card characters with numbers. By default, the wild-card characters is the question mark (?), but it can be any character specified by *filter*.

The routine works by first generating a pseudo-random number and using Digits from this number to replace the wild-card characters, and then incrementing the number until there's no file matching the name generated from the template.

3.14 SysGetFileDateTime

```
SysGetFileDateTime(name [, which]) -> timestamp
```

SysGetFileDateTime returns a time-stamp associated with the file identified by *name*. The timestamp is returned in the format 'yyyy-mm-dd hh:mi:ss'.

If *which* is 'modify', the time of the last modification is returned. If it's 'access', the time of last access is returned. If it's 'create', the file creation time is returned. Only the first letter of each of those options is significant. If the file system doesn't support access and creation times, the function returns the last modification time for everything.

3.15 SysSetFileDateTime

```
SysSetFileDateTime(name [, date [, time]]) -> success
```

SysSetFileDateTime sets the last modification time of the file specified by *name* to the date and time specified by *date* and *time*. The format of *date* is 'yyyy-mm-dd', and the format of *time* is 'hh:mi:ss'.

If neither *date* nor *time* are specified, the last modification time is set to the current time. If *date* is specified but *time* is not, only the date is changed. If *time* is specified by *date* is not, only the time is changed.

Returns 0 on success, or –1 on failure.

4 System Routines

The system routines return information about the operating system, library, or active processes, or perform process-control operations. This is the major area of incompleteness in the library.

4.1 List of System Routines

SysIni ([inifile],app,key,val,stem) → value: retrieve values from .ini files;

SysBootDrive () → value: returns the drive from which NT was booted, or the name of the Unix kernel file;

SysUtilVersion () → value: returns the regutil version;

SysVersion () → value: returns the operating system version;

SysWinVer () → value: returns the operating system version;

SysOS2Ver () → value: returns the operating system version;

SysLinVer () → value: returns the operating system version;

SysVersion () → value: returns the operating system version;

SysUtilVersion () → value: returns the version of this library;

SysDriveInfo (drive) → returns the free space on the drive, or the partition containing the argument;

SysDriveMap ([drive],[opt]) → list: lists accessible drives;

SysSetPriority (class,delta) → 0 or success: set the priority of the current process;

SysQueryProcess (thing) → 0 or success: get the process ID, or some canned data;

SysSleep (time) → 0 or success: block for the specified period of time;

SysSwitchSession (name) → 0 or success: brings a named application to the foreground;

SysSystemDirectory () → value: returns the name of the system directory;

SysVolumeLabel (drive) → returns the label on a specified drive;

SysWaitNamedPipe (name,[timeout]) → 0 or failure: waits on a named pipe.

4.2 SysIni

```
SysIni([inifile],app,key[,val]) -> value
SysIni([inifile],app,'all:',stemname) -> value
SysIni([inifile],'all:',stemname) -> value
```

Retrieves a value from a .ini file. This refers to Windows 3.1-style .ini files. I originally planned to use this function to retrieve values from the NT Registry as well, but I can't be bothered at this point. To read the registry, you can use my other package, w32funcs.

inifile is the name of the .ini file. The default is 'win.ini'. If you don't specify a path, the .ini file is expected to be in the system directory.

app is the name of a block of parameters in an .ini file. The block names appear in brackets in the file. *key* is the name of the parameter being retrieved or set. *val* is the value to set the parameter to. *stemname* is the name of a stem variable into which application or key names can be enumerated.

If *app* and *key* are specified, but *val* is not, the current value of the specified key is returned.

If *val* is 'delete:', the specified key is deleted. If *key* is 'delete:' or not specified, the entire block of parameters is deleted. Note that if *key* is not specified, the 'delete:' keyword is optional. The entire block of parameters will be deleted. It's not me, it's IBM.

If *key* is 'all:', the names of the keys in the block are returned in *stemname*, following the numeric index convention. If *app* is 'all:', the names of all the blocks in the file are returned in *stemname*.

4.3 SysBootDrive

```
SysBootDrive() -> value
```

Under NT, SysBootDrive returns the letter of the drive from which the system was booted (*e.g.*, 'C:' if the system was booted from drive c). This is mostly useful under OS/2 when you want to change the correct config.sys file. Under Unix, SysBootDrive() will return the name of the kernel from which the system was booted, or 'C:' if it isn't implemented.

4.4 SysWinVer

```
SysWinVer() -> value
```

Returns the system id and version in the format '*id major.minor*'. For NT, *id* is 'Windows95' or 'WindowsNT'. For Unix, it is the value returned by the command 'uname -s'.

4.5 SysOS2Ver

```
SysOS2Ver() -> value
```

SysOS2Ver is a synonym for SysWinVer. Lesson 1 in writing portable APIs: don't change the function names when you move from one platform to another, the way IBM did.

4.6 SysLinVer

```
SysLinVer() -> value
```

SysLinVer is another synonym for SysWinVer.

4.7 SysVersion

```
SysVersion() -> value
```

SysVersion is yet another synonym for SysWinVer. (Lesson 1 learned, I suppose).

4.8 SysUtilVersion

```
SysUtilVersion() -> value
```

SysUtilVersion returns the version number of the regutil library. Because the library is not strictly compatible with IBM's RexxUtil library, it does not return the same version numbers. The value returned by SysUtilVersion is the major version number followed by the minor version and release numbers, concatenated together. For instance, for version 1.1.5, the return value is 1.15. For 1.1.10, the return value would be 1.110.

4.9 SysDriveInfo

```
SysDriveInfo(drive) -> value
```

Under NT, returns the free space on the specified drive in the format '*drive free total label*'. *label* is the label of the drive, if any. *free* and *total* are the number of bytes free, and the total number of bytes on the drive. *drive* is the argument.

Under Unix, instead of drive, any file or directory name can be specified, and the information for the file's partition will be returned. The *drive* returned is the volume's mount point, and the *label* is the actual device name.

4.10 SysDriveMap

```
SysDriveMap([drive],[opt]) -> list
```

This function is not implemented for Unix.

Under NT, SysDriveMap returns a list of accessible drives. Under Unix, it returns a list of mounted partitions. The optional *drive* argument specifies the first drive to consider under NT, but has no effect on Unix.

opt can be one of the following values:

USED List all accessible drives or mount points;

FREE For NT, lists available drive letters (that is, if you have only a C: drive, it will list all the letters from D: to Z:). Under Unix, it does nothing;

LOCAL Lists only local files systems. Under NT, this means drives which are actually on the local machine and use a standard file system;

REMOTE Under Unix, returns only NFS and Samba-mounted drives. Under NT, returns LAN drives, and drives mounted using an installable file system;

REMOVABLE Lists drives which are not fixed or network drives, such as floppies and ZIP drives, but excluding CD-ROM drives;

CDROM Lists CD-ROM drives;

RAMDISK Lists ram disks.

4.11 SysSetPriority

```
SysSetPriority(class,delta) -> success code
```

Sets the priority of the current process. Possible values for *class* (NT only) are:

0 Don't change priority class;

1 Change class to idle;

2 Change class to normal;

3 Change class to 'real time';

4 Change class to 'server'.

Don't use any value other than 0 unless you know what you're doing.

delta can be any value between -31 and 31. 31 tries to increase the priority as much as it can, and -31 tries to decrease the priority as much as it can.

On success, SysSetPriority returns 0. On failure, it returns a return code which I may document some day.

4.12 SysQueryProcess

```
SysQueryProcess(thing) -> data
```

SysQueryProcess returns information based on its input:

PID	process id;
TID	thread id (currently always returns 0 on Unix);
PPRIO	process priority (currently always returns NORMAL);
TPRIO	thread priority (currently always returns NORMAL);
PTIME	process time used;
TTIME	thread time used (currently always returns process time used);

Currently, only PID and PTIME give anything like useful information. TID works for NT.

4.13 SysSleep

```
SysSleep(time) -> success code.
```

SysSleep blocks the current process for *time* seconds. Time may be a fraction of a second (*e.g.*, .24 or 6.5), but note that this is incompatible with IBM's original SysSleep. It is compatible with the Object Rexx SysSleep, and it's useful.

Calling SysSleep is better than, say, looping on calls to time(). This is called 'busy waiting':

```
endtime = time('s')+2
do while time('s') < endtime
  nop
  end
```

and it's bad because it uses all kinds of CPU cycles simply testing the current time. The equivalent code using SysSleep:

```
call SysSleep 2
```

has the same effect on the program, but doesn't use additional cycles because the blocking is handled by the system scheduler, which is constantly testing the current time anyway.

4.14 SysSwitchSession

```
SysSwitchSession(name) -> success code
```

SysSwitchSession is supposed to bring the session identified by *name* to the foreground. Under NT, where it is implemented, *name* is the name on the title bar. I'll tell you more about the Unix implementation when it's done.

This function is not implemented on Unix.

4.15 SysSystemDirectory

```
SysSystemDirectory() -> value
```

Under NT, returns the name of the system directory, which is generally WinNT on the boot drive. On Unix, it returns '/etc'.

4.16 SysVolumeLabel

```
SysVolumeLabel(drive) -> name
```

On NT, returns the label on a specified drive. On Unix, returns the device file associated with the specified volume.

This function is not implemented on Unix.

4.17 SysWaitNamedPipe

```
SysWaitNamedPipe(name [, timeout]) -> 0 or success
```

Waits for the specified named pipe to become readable. The named pipe name must have the format `\\server\pipe\name` on NT, where *server* is the name of a server machine (or '.' for the local machine), *pipe* is 'pipe', and *name* is a name which doesn't include any slashes or back-slashes. On Unix, a named pipe is just a fifo and can have any name. *timeout* is specified in milliseconds. –1 means there is no timeout. 0 and omitting the timeout value cause the operation to wait some default period of time.

If there is data to read on the pipe, returns 0. If the operation times out, returns 1460. Otherwise, it returns a system-defined error number.

5 Macro-Space Manipulation Routines

The macro-space manipulation routines allow a program to control the macros available in the execution environment. Most usefully, they allow external macros to be loaded in to the local macro address space, and they allow collections of macros to be saved to, and loaded from, a compact, binary format. This allows library functions to be stored externally and loaded quickly, and it may allow proprietary code to be shipped in a format which is not susceptible to casual inspection.

As of version 2.2, the necessary API is provided only as a set of 'stub' routines in Regina, so these routines are not yet functional. They are not included in pre-compiled versions of regutil, since this would force people to upgrade to Regina 2.2. They can be included in a build for use with a later version of Regina or with another interpreter, by adding "-DMACROSPACE" to the CFLAGS line in the appropriate make file.

5.1 List of Macro-Space Manipulation Routines

SysAddRexxMacro (name, file, [order]) → 0 or failure : adds a macro;

SysClearRexxMacroSpace () → 0 or failure : clears all macros;

SysDropRexxMacro (name) → 0 or failure : drops a macro;

SysLoadRexxMacroSpace (file) → 0 or failure : initialises a macro-space from a file;

SysQueryRexxMacro (name) → ‘B’, ‘A’ or ‘’ : determines whether a macro is defined;

SysReorderRexxMacro (name,order) → 0 or failure : moves the search order of a macro;

SysSaveRexxMacroSpace (file) → 0 or failure : saves a macro space to a file.

5.2 SysAddRexxMacro

```
SysAddRexxMacro(name, file, [order]) -> 0 or failure
```

Reads a macro called *name* from a file called *file* and makes it available to the current program. If *order* is specified and starts with ‘A’, the macro name will be added to the end of the macro space. Otherwise, it will be added to the beginning. See SysReorderRexxMacro5.7 for a discussion of what this means.

Macros loaded using SysAddRexxMacro have the useful characteristics of external functions (they are stored in a separate file) but act like locally defined procedures (they can access global stem variables, for instance).

Returns 0 on success. The other possible return codes from the macro-space functions are 1 (not enough storage available), 2 (requested function not found), 3 (file extension required for save), 4 (macro functions exist), 5 (file I/O error in save/load), 6 (incorrect format for load), 7 (requested cannot be found), 8 (invalid search order position), and 9 (API not initialized).

5.3 SysClearRexxMacroSpace

```
SysClearRexxMacroSpace() -> 0 or failure
```

Clears all macros previously loaded using SysAddRexxMacro or SysLoadRexx-MacroSpace from the macro space.

Returns 0 on success. See SysAddRexxMacro, section 5.2, for the other possible return values.

5.4 SysDropRexxMacro

```
SysDropRexxMacro(name) -> 0 or failure
```

Drops the named macro fromt he macro space. The macro must have previously been loaded using SysAddRexxMacro or SysLoadRexxMacroSpace.

Returns 0 on success. See SysAddRexxMacro, section 5.2, for the other possible return values.

5.5 SysLoadRexxMacroSpace

```
SysLoadRexxMacroSpace(file) -> 0 or failure
```

Loads all macros from the file *file*, which must have been saved using SysSaveRexxMacroSpace. It's not guaranteed that macro space files will be compatible between releases of Regina. It *is* guaranteed that they will not be compatible between different Rexx implementations.

Returns 0 on success. See SysAddRexxMacro, section 5.2, for the other possible return values.

5.6 SysQueryRexxMacro

```
SysQueryRexxMacro(name) -> 'A', 'B', or ''
```

Searches the macro space for a function called *name*. If it finds it, it returns 'A' if the macro was loaded using load order 'after' or 'B' if it was loaded using load order 'before'. If it doesn't find it, returns the empty string.

5.7 SysReorderRexxMacro

```
SysReorderRexxMacro(name,order) -> 0 or failure
```

Changes the search order for macro *name*. If *order* begins with 'B' ('before'), the function from the macro space will override any locally-defined function of the same name. If it begins with 'A' ('after'), any locally-defined function will over-ride the version loaded into the macro space.

Returns 0 on success. See SysAddRexxMacro, section 5.2, for the other possible return values.

5.8 SysSaveRexxMacroSpace

```
SysSaveRexxMacroSpace(file) -> 0 or failure
```

Saves all macros loaded using SysAddRexxMacro or SysLoadRexxMacroSpace to a file called *file*. The file name must include an extension.

Returns 0 on success. See SysAddRexxMacro, section 5.2, for the other possible return values.

6 Console I/O Routines

The console I/O routines allow simple terminal-mode updates. The Curses library gives more flexibility, and will be more efficient over slow connections. Rexx/TK is currently the best option for implementing GUI interfaes.

I originally considered implementing these routines using curses, but the value added would be slight (SysCurPos and SysTextScreenRead are the only functions which would be fixed by this), and the availability of the full curses package makes the effort redundant.

6.1 List of Console I/O Routines

SysCls () : clears the screen;

SysCurPos ([row],[column]) → row column: moves the cursor, and returns its current position on the screen;

SysCurState (state): makes the cursor visible or invisible;

SysGetKey ([echo],[timeout]): retrieves a keystroke;

SysTextScreenRead (row,column,len) → text: reads the screen;

SysTextScreenSize () → rows columns: gets the size of the screen;

RxMessageBox (text, [title], [button], [icon]) → button id: displays a message box and returns the button selected.

6.2 Example

Here's a script which prints an X of screen positions on the screen, then waits for a keypress.

```
call rxfuncadd 'sysloadfuncs', 'rexxutil', 'sysloadfuncs'
call sysloadfuncs

call syscurstate 'off'
call syscls

parse value systextscreensize() with rows cols

top=min(rows,cols)-1

do i = 1 to top
  call syscurpos i,i
  call charout 'stdout', '('i',' i')'
  call syscurpos i,top-i+1
  call charout 'stdout', '('i',' top-i+1')'
  end
call SysGetKey 'noecho'
call syscurstate 'on'
call SysDropFuncs
say ''
```

6.3 SysCls

```
SysCls()
```

SysCls clears the screen as quickly as possible.

6.4 SysCurPos

```
SysCurPos([row],[column]) -> row column
```

SysCurPos sets the cursor position to *row* and *column*, and returns the former position. If *row* and *column* are not returned, it doesn't move the cursor. I don't know how to retrieve the current position on Unix, so it always returns 0 0 on that platform. For more advanced screen handling, consider using the RxCurses package.

6.5 SysCurState

```
SysCurState(state)
```

SysCurState makes the cursor visible or invisible. If *state* is 'on', the cursor is made visible. If it is 'off', the cursor is made invisible.

6.6 SysGetKey

```
SysGetKey([echo],[timeout]) -> keystroke
```

SysGetKey returns a keystroke. If *echo* is specified, and it is 'n' or 'no', the keystroke is not displayed on the screen. Otherwise it is.

If *timeout* is specified, it is a number of seconds to wait for input. As with SysSleep, fractions of seconds are allowed. if *timeout* seconds pass without a key being pressed, SysGetKey returns the empty string. By default, or if *timeout* is 0, SysGetKey waits until a key has been pressed before returning. This behaviour is incompatible with IBM's implementation.

For compatibility with IBM's implementation, function keys can be returned in an ugly, system-dependent manner. For NT, this means that if SysGetKey returns a 0, you must call it again, and the second return value tells you what key was pressed. For Unix, it means something different again – generally, alt-keys will return either a high-ascii value, or escape followed by the ascii value of the key, while function keys return different escape sequences depending on the terminal.

The current release tries to return 'f1' for F-1, 'Home' for the home key, and so forth. To figure out what gets returned, you have to press the keys and print it out.

6.7 SysTextScreenRead

```
SysTextScreenRead(row,column,len) -> text
```

SysTextScreenRead returns the characters printed on the screen for *len* characters, starting at position *row,column*. The end of line is indicated by a new-line character (ascii value 10).

SysTextScreenRead is not implemented for Unix systems, because I don't know how to retrieve the information from a tty terminal.

6.8 SysTextScreenSize

```
SysTextScreenSize() -> rows columns
```

SysTextScreenSize returns the number of rows and columns in an NT text window, or a Unix terminal. For NT, the size is the size of the buffer behind the window, so if you have a scroll-back set up, for instance, the value will be larger than the actual screen size. For Unix, the size is the size the kernel believes the terminal to be. This will generally be accurate for xterms and many terminal emulators, but can be wrong if you're using a terminal emulator which uses non-standard sizes, on a system (HP-UX comes to mind) which doesn't account for that. The problem can be fixed by using the stty command to tell the system the actual size of the window.

6.9 RxMessageBox

```
RxMessageBox(text, [title], [button], [icon]) -> button id.
```

RxMessageBox displays a message box on the screen and waits for the user to press a button. It's not a console I/O function, but it doesn't seem to fit in anywhere else, so I document it here.

Text is the text that will be written in the message box. The system will wrap long text at around 120 characters, or 60% of the screen width (this is from observation – I haven't seen any system documentation on this). This tends to look ridiculous, so you can force line breaks by adding a line-feed (character 10) to the string. See the example for. . . an example.

Title is the text that is put on the title bar. By default, the title is 'Error!', meaning that you really should specify something.

Icon is one of 'hand', 'question', 'exclamation', 'asterisk', 'information', or 'stop'; the default is 'hand'. Note that 'hand' and 'stop' present the same icon, and on my machine, it's a sort-of X in a red circle, which is nothing like a hand or a stop-sign. 'Question' presents a question mark in a bubble. "Asterisk' and 'information' present the same icon, which is a lower-case i in a bubble, and exclamation is an exclamation point in a yellow rhombus.

Button specifies which buttons appear on the message box. It is one of 'ok' for an OK button, 'okcancel' for an OK button and a Cancel button, 'abortretryignore', for an abort button, a retry button, and an ignore button, 'yesnocancel' for a yes button, a no button and a cancel button, 'yesno' for a yes button and a no button, and 'retrycancel' for a retry button and a cancel button.

The return code is a number from 1 to 7 indicating which button was selected by the user. The numbers correspond to 'OK', 'Cancel', 'Abort', 'Retry', 'Ignore', 'Yes', or 'No', respectively.

```
rcc = RxMessageBox('Things are going badly, which is ' ||,
                   'my fault, but your problem.' || d2c(10) ||,
                   'Press cancel to give up, or retry to take ' ||,
                   'another stab at it.', 'Oops', 'RetryCancel')

/* cancel */
```

```
if rcc = 2 then do
  say 'good choice'
  exit 1
  end
else do
  say 'your funeral'
  redo()
  end
```

7 Stem Manipulation Routines

The stem manipulation routines are used to manipulate stem variables. Not all of the options in IBM's routines are supported, and regutil has a few extra functions which are not available in IBM's version.

Generally speaking, the stems must follow the numeric index convention. This is a long-standing way of emulating numeric arrays in Rexx, where the .0 stem element holds the number of entries in the array (*count*), and elements 1 through *count* hold the data.

I include sysdumpvariables() here even though it's not specifically anything to do with stems.

7.1 List of Stem Manipulation Routines

SysDumpVariables ([filename]) → 0 or -1: dump the names and values of all variables to a file;

SysStemCopy (from, to[, fromindex, toindex, count, insertoverlay]) → 0 or -1: copy a stem to another stem;

SysStemDelete (stem, index[, count]) → 0 or -1: delete elements from a stem;

SysStemInsert (stem, index, value) → 0 or -1: insert a value into a stem;

SysStemSort (stemname[, order] [,sensitivity] [,startpos,endpos] [,firstcol,lastcol]) → 0 or -1: sort the elements of a stem;

RegStemDoOver (stem, variable[, outstem]) → 0 or 1: enumerates the indexes of a stem;

RegStemRead (filename, stemname) → 0 or 1: read file into stem;

RegStemWrite (filename, stemname) → 0 or 1: write file from stem;

RegStemSearch (needle, haystack[, start] [, flags]) → 0 or index: search a stem for a value.

7.2 Example

Here's a script which reads a file into a stem, sorts the stem, and then writes it out to the file again:

```
call rxfuncadd 'sysloadfuncs', 'rexxutil', 'sysloadfuncs'
call sysloadfuncs

call regstemread 'bob', 'bob'
call sysstemsort 'bob'
call regstemwrite 'bob', 'bob'
```

7.3 SysDumpVariables

```
SysDumpVariables([filename])
```

SysDumpVariables is a debugging aid which dumps all variables to the file *filename* in the format

```
Name=GREETING, Value="Have a nice day."
```

Nothing special is done with variables that include new-lines or quotes. If no file is specified, the dump is written to standard output.

7.4 SysStemCopy

```
SysStemCopy(from, to[, fromindex, toindex, count, insertoverlay])
```

SysStemCopy copies stem *from* to *to*. The stems must currently follow the numeric index convention (this might change in the future).

fromindex is the index number of the first element in *from* which should be copied. *toindex* is the index number of the first target element in *to*. The default for both indices is 1. *count* is the number of elements to copy. The default is all of them. If *insertoverlay* is 'I', the elements from *from* are inserted into *to*, and existing elements in *to* are shifted. Otherwise, existing elements in *to* are overwritten.

If *toindex* is beyond the end of *to*, the array is extended and filled with zero-length strings. If *fromindex+count* is greater than the number of elements in *from*, only the number of elements between *fromindex* and the end of *from* are copied.

If the default options are given for *fromindex*, *toindex*, *count*, and *insertoverlay*, *from* is copied exactly on top of *to*. So, if *to* has 20 elements, and *from* has 10 elements, the last 10 elements of *to* will be deleted. Probably, this is a bug, but you can work around it by passing *count* as *from*.0.

7.5 SysStemDelete

```
SysStemDelete(stem, index[, count]) -> 0 or -1
```

Deletes *count* entries from a stem, starting at index *index*. The default count is 1. The stem must follow the numeric index convention.

7.6 SysStemInsert

```
SysStemInsert(stem, index, value)
```

Inserts *value* at index position *index*. If there are elements with larger indices than *index*, they are shifted up one. The stem must follow the numeric index convention.

7.7 SysStemSort

```
SysStemSort(stem[, order] [, sensitivity] [,startpos, endpos]
[,firstcol,lastcol])
```

SysStemSort sorts a stem. This is a pure ASCII sort, which doesn't take into account any language-based collation sequence. *order* can be 'ascending' or 'descending', the default is 'ascending'. *sensitivity* can be 'sensitive' or 'insensitive', which determines whether to fold upper-case letters into lower-case letters. Again, this doesn't take into account accented characters, although it might if Regina were to call setlocale(3). Only the first letter of each of these options is significant.

If *startpos* and *endpos* are given, only elements from *startpos* to *endpos* (inclusive) will be sorted.

If *firstcol* and *lastcol* are given, elements will be sorted based on the *firstcol*th to *lastcol*th characters, inclusive. We start counting characters at 1.

The stem must follow the numeric index convention.

7.8 RegStemDoOver

```
RegStemDoOver(stem, variablename[, outstem]) -> 0 or 1
```

RegStemDoOver simulates the object rexx construct 'do x over y.'. It allows the indices of a stem to be treated as data by retrieving each index in turn. It returns 1 while there are additional elements, and 0 after the last element has been returned.

stem is the name of the stem. *variablename* is the name of a variable to set to the next stem index. If given, *outstem* is the name of a stem to set to the complete set of indices of *stem*, using the numeric index convention.

```
/* read values into a stem */
do until name = 'end'
  parse linein name otherstuff
  data.name = otherstuff
  end

/* process the values */
do while regstemdoover('data.', 'i')
   /* skip the 'end' element. This is something
    * to do with this dumb example, not a feature
    * of regstemdoover */
   if i \= 'end' then
     call somefunction i, data.i
   end
```

The stem does *not* have to follow the numeric index convention (otherwise the function would be a bit pointless, but I thought I'd mention it). You can't nest calls to RegStemDoOver. For instance, this code will not work as desired:

```
/* loop over indices of data */
do while regstemdoover('data.', 'i')
```

```
/* and now loop over mana */
do while regstemdoover('mana.', 'j'
  call somefunction i, j
  end
end
```

Instead, you must first store the indices of 'data' in another stem.

Also, RegStemDoOver does not pick up changes which have occurred to the stem since the first call to the function.

7.9 RegStemRead

```
RegStemRead(filename, stem)
```

RegStemRead reads the contents of file *filename* into stem stem using the numeric index convention (number of lines in the 0 element, data in numbered elements starting at 1). When possible, it uses memory-mapped I/O to read the values, which should be the most efficient method possible. As a result, RegStemRead is expected to be measurably faster than, eg, using linein, as well as being more convenient.

7.10 RegStemWrite

```
RegStemWrite(filename, stem)
```

RegStemWrite reads the contents of stem *stem* to file *filename*. The stem must follow the numeric index convention. This might be faster than using lineout, and it's convenient, but it's mostly included as a companion to RegStemRead.

7.11 RegStemSearch

```
RegStemSearch(needle, haystack [, start] [,flags]) -> 0 or index
```

RegStemSearch searches the stem *haystack* for *needle*. It returns the index position of a stem element which matches *needle*, or 0 if there are no such elements. The stem must follow the numeric index convention.

Start is the starting index position (the default is 1). *Flags* can be any combination of 'C', 'E', and 'S'. 'C' indicates that the search should be case-sensitive (the default is case-insensitive). 'E' indicates that an exact match is required (the default is to perform a substring match). 'S' indicates that the stem is sorted. When the stem is sorted, RegStemSearch uses a binary search, otherwise it uses a linear search.

RegStemSearch is primarily a convenience function. In its fastest mode of operation (performing exact matches on a case-sensitively sorted stem), the overhead of looking up Rexx variable values makes it slightly slower than the equivalent code written in Rexx, and much slower than a more sensible use of stems. What I mean by this is that this code:

```
colours.0 = 3
colours.1 = 'blue'
colours.2 = 'green'
colours.3 = 'red'

if regstemsearch(colour, 'colours',,'ces') \= 0 then
   say colour 'is a colour'
```

would be more sensible and also much faster as

```
colours. = 0
x = 'blue'
colours.x = 1
x = 'green'
colours.x = 1
x = 'red'
colours.x = 1

if colours.colour then
   say colour 'is a colour'
```

In its default mode of operation, it can be used along with RegStemRead to replace SysFileSearch, but again the overhead of setting and retrieving Rexx variables from the library makes it slower.

8 Semaphore Routines

A semaphore is an inter-process communications mechanism which allows information to be signalled between processes. Generally speaking, semaphores are counters which can be shared between processes, and which allow processes to block while waiting for the semaphore to reach special values (normally 0 and not-0).

RexxUtil provides two specialised kinds of semaphores: mutual exclusion, or mutex semaphores, and event semaphores.

You use mutex semaphores to cooperatively control access to shared resources. For instance, if two programs use the same log file to record their progress, they might use a mutex to ensure that log messages don't overlap. The routine that writes to the log would first 'lock' the mutex by calling SysRequestMutexSem, then write the log message, flush the log file, and finally release the semaphore using SysReleaseMutexSem.

An event semaphore allows a process to wait efficiently until another process lets it know that there's something to do. For instance, a server process which accepts requests in the form of text files could call SysWaitEventSem when it has nothing to do. A client process which has created input for the server would call SysPulseEventSem to notify the server that an new file has been written.

There are two kinds of event semaphores. Manual-reset semaphores have their states changed to 'posted' or 'reset' and the state stays that way no matter how many other processes wait for the semaphore. Auto-reset semaphores (the default) automatically change from 'posted' to 'reset' as soon as a waiting process has been released.

A manual-reset semaphore is a bit like a door which is always either open or shut, while an auto-reset is like a turnstile that lets one person through, then locks. See SysPulseEventSem and SysPostEventSem for more details.

8.1 List of Semaphore Routines

SysCloseEventSem (semid) → 0 or failure: close an event semaphore;

SysCloseMutexSem (semid) → 0 or failure: close an mutex semaphore;

SysCreateEventSem ([name],[manualreset]) → handle: create an event semaphore;

SysCreateMutexSem ([name]) → handle: create a mutex semaphore;

SysOpenEventSem (name) → handle: open an event semaphore;

SysOpenMutexSem (name) → handle: open a mutex semaphore;

SysPostEventSem (semid) → 0 or failure: set the semaphore status to 'on';

SysPulseEventSem (semid) → 0 or failure: set the semaphore status to 'on' then 'off' again;

SysReleaseMutexSem (semid) → 0 or failure: unlock a mutex semaphore;

SysRequestMutexSem (semid, [timeout]) → 0 or failure: lock a mutex semaphore;

SysResetEventSem (semid) → 0 or failure: set the semaphore status to 'off';

SysWaitEventSem (semid, [timeout]) → 0 or failure: wait for an event semaphore to be turned 'on'.

8.2 SysCloseEventSem

```
SysCloseEventSem(semid) -> 0 or failure
```

Closes the event semaphore associated with *semid*. *semid* must have been returned by SysOpenEventSem or SysCreateEventSem. A return code of 0 means success. Any other return code means 'not success'.

8.3 SysCloseMutexSem

```
SysCloseMutexSem(semid) -> 0 or failure
```

Closes the mutex semaphore associated with *semid*. *semid* must have been returned by SysOpenMutexSem or SysCreateMutexSem. A return code of 0 means success. Any other return code means 'not success'.

8.4 SysCreateEventSem

```
SysCreateEventSem([name],[manualreset]) -> handle
```

Creates a new event semaphore keyed on *name*. If *name* is not specified, the semaphore is private to the process, and so completely useless until Regina supports multi-threading. If *manualreset* is set to a non-zero value, it changes the behaviour of SysPulseEventSem and SysPostEventSem as described later.

On success, SysCreateEventSem returns a handle to the semaphore, which should be used in subsequent semaphore calls. On failure, it returns the empty string.

8.5 SysCreateMutexSem

```
SysCreateMutexSem([name]) -> handle
```

Creates a new mutex semaphore keyed on *name*. If *name* is not specified, the semaphore is private to the process, and so completely useless until Regina supports multi-threading. I hope you're not getting the impression this documentation has been cut-and-pasted a lot.

On success, SysCreateMutexSem returns a handle to the semaphore, which should be used in subsequent semaphore calls. On failure, it returns the empty string.

8.6 SysOpenEventSem

```
SysOpenEventSem(name) -> 0 or handle
```

Opens an event semaphore keyed on *name*. The semaphore must have been previously created (usually by another process) using SysCreateEventSem.

On success, SysOpenEventSem returns a handle to the semaphore, which should be used in subsequent semaphore calls. On failure, it returns 0. I don't know why this is different from SysCreateEventSem.

8.7 SysOpenMutexSem

```
SysOpenMutexSem(name) -> 0 or handle
```

Opens a mutex semaphore keyed on *name*. The semaphore must have been previously created (usually by another process) using SysCreateMutexSem.

On success, SysOpenMutexSem returns a handle to the semaphore, which should be used in subsequent semaphore calls. On failure, it returns 0.

8.8 SysPostEventSem

```
SysPostEventSem(semid) -> 0 or failure
```

Sets an event semaphore to 'posted' state.

If the semaphore is auto-reset (meaning the *manualreset* flag was *not* set in SysCreateEventSem), the behaviour is different when there are processes waiting than it is

when there are no processes waiting. If there are waiting processes, they are all released, and the state of the semaphore is set to 'reset'. If there are no waiting processes, the state of the semaphore is set to 'posted'.

If the semaphore is manual-reset, it stays in 'posted' state until it is explicitly reset using SysPulseEventSem or SysResetEventSem.

8.9 SysPulseEventSem

```
SysPulseEventSem(semid) -> 0 or failure
```

Sets an event semaphore to 'posted' and then automatically resets it.

If there are no processes waiting on the semaphore, the semaphore is reset without releasing anything.

Otherwise, if the semaphore is auto-reset, exactly one process will be released before the semaphore is reset. If the semaphore is manual-reset, all waiting processes will be released before the semaphore is reset. Hopefully, this table will make the behaviour clear:

Type	*Event*	*Waiters*	*Final State*	*Empty State*
Auto-reset	Pulse	Release 1	Reset	Reset
	Post	Release all	Reset	Posted
	Reset	None	Reset	Reset
Manual-reset	Pulse	Release all	Reset	Reset
	Post	Release all	Posted	Posted
	Reset	None	Reset	Reset

8.10 SysReleaseMutexSem

```
SysReleaseMutexSem(semid) -> 0 or failure
```

Unlocks a mutex semaphore which was previously locked using SysRequestMutexSem.

8.11 SysRequestMutexSem

```
SysRequestMutexSem(semid[, timeout]) -> 0 or failure
```

Locks a mutex semaphore. If the semaphore is already locked by another process, waits *timeout* milliseconds. If *timeout* is not specified, it will wait until the end of time (which is currently projected to be in September 2038).

If it returns 0, then the lock was attained. If it returns a non-zero value, then the lock was not attained, and the shared resource must not be manipulated. The return value on time-out is supposed to be 121, but this sounds quite improbably to me.

You should always release the mutex by calling SysReleaseMutexSem as soon as possible after aquiring it.

8.12 SysResetEventSem

```
SysResetEventSem(semid) -> 0 or failure
```

Sets an event semaphore to ‘reset’. See SysPulseEventSem and SysPostEventSem for more discussion.

8.13 SysWaitEventSem

```
SysWaitEventSem(semid[,timeout]) -> 0 or failure
```

Waits for an event semaphore to be set to ‘posted’. If *timeout* is specified, SysWaitEventSem waits up to *timeout* milliseconds. Otherwise, it waits forever.

If the semaphore is auto-reset and its state is ‘posted’ before the call to SysWaitEventSem, the state will be changed to ‘reset’.

SysWaitEventSem returns 0 when it is returning due to the semaphore being posted. It’s supposed to return 121, if you can believe it, if the function timed out. Other nonzero return codes can’t be good.

Index