# Associative Arrays for Rexx

Patrick TJ McPhee (ptjm@interlog.com)

Version 1.0.0, 26 May 2003

# Contents

# 1 Introduction

Associative arrays are indexed data structures which use arbitrary data as keys. Where a 'normal' array idenifies a value by its location in the array structure, associative arrays associate values with their keys, which are often useful application-specific data in themselves. This concept is widely used in scripting applications; for instance, Rexx's stem variables are a form of associative array.

While they are a form of associative array, stem variables in ANSI Rexx don't provide two useful features: an operation to enumerate the keys in use in the array, and operations which act on the array as a whole, especially copying and passing to a function. The RxHash package is a set of routines for manipulating associative arrays which can be copied and passed as function arguments. It includes functions to convert between RxHash arrays and stem variables, and C language functions which can be used by other Rexx loadable libraries to generate and modify RxHash arrays.

This manual describes both the C and Rexx routines. People who don't wish to program in C can ignore the C routines section.

## 1.1 Installation

RxHash includes two pre-compiled binaries for Win32 platforms, one pre-compiled binary for OS/2, and source code which should compile on any system which includes an ANSI C compiler. There is no installation program.

On Win32 platforms with Regina, copy win32/rxhash.dll to a directory on your path, or to the directory containing regina.exe or your rexx-enabled application. If you use a Rexx interpreter other than Regina, you either need to recompile the library using your interpreter's development kit or download and install rexx/trans and use rexxtrans/rxhash.dll instead.

On OS/2, copy os2/rexxre.dll to a directory in your LIBPATH.

On Unix, you need to compile the library. The distribution does not include a configuration script, but it includes make files which have been known to work using the stock vendor compiler on several Unix systems. If you have one of those systems, link the appropriate make file to the name 'Makefile' and build the 'dist' target. For instance, on Solaris:

```
ln Makefile.sun Makefile
make dist
```

On most platforms, this builds a shared library called librxhash.so. On HP-UX, the file is called librxhash.sl, and on AIX, it's called librxhash.a. The path to this library can be set in three ways:

Most Unix systems allow a shared library search path to be embedded into program files. If you build regina (or your rexx-enabled application) such that this path is set to include a directory such as /opt/regina/lib or /usr/local/lib, you can install RxHash by copying the shared library to this directory. Otherwise, you need to either set an environment variable or change the way the system searches for shared libraries.

Unix systems typically use a different path for shared libraries than they do for program files. The name of the environment variable used for the shared library path is not standardised, however most systems use LD_LIBRARY_PATH. Notable exceptions are AIX (LIBPATH) and HP-UX (SHLIB_PATH for 32-bit executables, LD_LIBRARY_PATH for 64-bit executables). To install RxHash, add an appropriate directory to the shared library path for your machine and copy the shared library to that directory.

Finally, some systems provide a utility (often called ldconfig) which can be used either to set the standard search path for shared libraries, or to provide a database of shared libraries. On such a system, RxHash can be installed by copying the shared library to an appropriate directory and using this utility to add it to the search database. You'll need to consult your system documentation for more information.

## 1.2 Reporting Bugs

I don't anticipate making a lot of changes to this library in the future, but I would like it to be bug-free.

If you 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 Rexx interpreter and its version, and the version of rxhash, and to have a set of steps for reproducing the bug. The example below shows how to retrieve the interpreter and library version information:

```
/* report useful version information */
parse version ver
say 'interpreter:' ver

call rxfuncadd 'arrversion', 'rxhash', 'arrversion'
say 'rxhash:' arrversion()
```

## 1.3 Using RxFuncAdd

All the routines in rxhash can be loaded either directly using RxFuncAdd, or indirectly using ArrLoadFuncs. RxFuncAdd takes three arguments – the name of the function as it will be used in the rexx program, the name of the library from which to load the function, and the name of the function as it appears in the library.

```
if rxFuncAdd('arrloadfuncs', 'rxhash', 'arrloadfuncs') then
  call ArrLoadFuncs
else do
  /* rxFuncErrMsg() is a Regina extension */
  say 'RxHash load failed:' rxFuncErrMsg()
  exit 1
  end
```

RxFuncAdd returns 0 on success, or 1 on failure. Regina has a function called RxFuncErrMsg which can give useful information about the reason for a load failure. A few common reasons for failure are:

Path issues: the library is called rxhash.dll on Win32 and OS/2 platforms, librxhash.a on AIX, librxhash.sl on HP-UX, and librxhash.so on other Unix platforms. On

Win32, this file needs to be in the path, or in the directory containing your rexx interpreter. On OS/2, it needs to be in a directory specified in LIBPATH. On Unix systems, it needs to be in a directory listed in LIBPATH on AIX, SHLIB_PATH on HP-UX 32-bit, or LD_LIBRARY_PATH on most other Unix systems. Some systems have an ldconfig utility which allows you to forego setting this environment variable.

Windows 95: early releases of windows 95 did not include msvcrt.dll, the C runtime library used by RxHash. This library is sometimes installed with applications software. It can also be obtained through service packs, or from the Microsoft web site.

Rexx.exe: 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). This will not be a problem with most other interpreters.

Already loaded: IBM’s interpeters leave functions registered between invocations, unless they are explicitly deregistered. rxFuncAdd( ) will return 0 if the function is already registered. You can test for it using rxFuncQuery( ).

### 1.4 Licencing

RxHash is distributed free of charge in the hopes that it will be useful, but without any warranty. It is distributed under the terms of the Mozilla Public License. The precise details of the licence are found in the file MPL-1.1.txt in the distribution.

My reading of the licence is that you may use the library for any purpose, however if you make changes to the library, you must make them available for others to use. If you use the library purely as shipped by me, even if you link in the C routines to your C callable libraries, you have no obligation to publish your code. If you alter, say, the hash routines, you do need to publish those alterations. If you, say, replace the hash routines completely, you can organise things in such a way that your replacements don’t need to be published, but you’ll have to publish at least the interface to your routines.

## 2 Rexx Functions

The functions described in this section are meant to be called from rexx programs. In general, you call ArrNew( ) to initialise an array, use the other functions to manipulate it, and finally use ArrDrop( ) to release the memory used to hold those values.

### 2.1 ArrLoadFuncs

```
ArrLoadFuncs() -> 0
```

ArrLoadFuncs( ) registers all the functions in the library with the Rexx interpreter. This is the fastest and most convenient way of initialising the library. See section 1.3 for an example.

## 2.2 ArrDropFuncs

```
ArrDropFuncs() -> 0
```

ArrDropFuncs( ) deregisters all the functions in the library. This needs to be done if you are using an IBM interpreter and want to upgrade the library. On the other hand, functions are not reference counted, so adding twice and dropping once results in the functions being dropped. If you use an IBM interpreter, you need to be careful about dropping functions if there are running programs which use them.

## 2.3 ArrVersion

```
ArrVersion() -> 1.0.0
```

ArrVersion( ) returns the version number of the library in the format *version.release. modification*. This is useful for reporting bugs (see section 1.2), and for ensuring that an upgrade has been successful.

## 2.4 ArrNew

```
ArrNew([default]) -> arr
```

ArrNew( ) allocates memory for the array and returns a pointer to it. *Arr* can be printed using c2x( ), but is not intended to be used apart from passing it to the other functions in the package. All the array manipulation routines require the return code of ArrNew( ) to be passed as the first argument.

If *default* is specified, it is the default value to return from ArrGet( ) when the key is not found. Otherwise, the default value is the null string.

## 2.5 ArrSet

```
ArrSet(arr, key, value) -> 0
```

ArrSet( ) associates *value* with *key* in array *arr*. *value* can be any value, but *key* must not be the null string. If you set the same *key* twice, the previous value is replaced. Look-ups in RxHash are case-sensitive.

## 2.6 ArrDefault

```
ArrDefault(arr, value) -> 0
```

ArrDefault( ) sets the default value which is returned by ArrGet( ) if the *key* argument is not in *arr*. The default default value is the null string.

## 2.7 ArrGet

```
ArrGet(arr, key) -> value
```

ArrGet( ) looks up *key* in array *arr* and returns its value. If *key* is not found in the array, the null string is returned, unless a user-specified default is in effect for *arr*. If there's a user-specified default, that value is returned. In contrast to many C associative array interfaces, *key* is not added to the array if it isn't already there. Look-ups in RxHash are case-sensitive.

Use ArrIn( ) to distinguish cases where *key* is associated with the null string or user-specified default value from cases where *key* is not in the array.

## 2.8 ArrIn

```
ArrIn(arr, key[, key2, ...]) -> 0 or 1
```

ArrIn( ) returns 1 if all the specified keys are stored in the array *arr*, or 0 if any of them aren't. For instance:

```
respondto: procedure
 parse arg guy

 do select
   when ArrIn(guy, 'tall', 'dark', 'handsome') then
     call go_on_date guy
   when ArrIn(guy, 'medium', 'dirty blonde', 'scruffy') then
     call send_bug_report guy
   otherwise
     call wash_hair
   end
```

In case you missed it, look-ups in RxHash are case-sensitive.

## 2.9 ArrDoOver

```
ArrDoOver(arr[, reset]) -> key
```

ArrDoOver( ) is named after the object rexx 'do x over y.' instruction. It returns the 'next' key value from array *arr*. When the end of the array is reached, the null string is returned. This contrasts with RegUtil's RegStemDoOver( ), which returns 0 or 1 and takes the name of a variable into which to stuff the key value.

The order in which keys are returned depends on the hashing algorithm used and the order in which values were added to the array. It may change between releases if the hash algorithm is improved.

If *reset* is passed as an argument, with any value, the enumeration starts over from the start of the hash table. If you always iterate fully through the hash table, stopping only when the null string is returned, you don't need to use *reset*. If you exit the loop after some interesting value is discovered, you should start your next iteration with a *reset* argument.

You can intersperse ArrDoOver( ) calls on different arrays, however there is state information associated with each array, so you can't nest different ArrDoOver( ) loops using the same *arr* argument. You can get the same effect by making a copy using ArrCopy( ). If you add elements to *arr* while looping using ArrDoOver( ), you will get inconsistent results – some new values will be returned, but others won't. If you delete the most recently returned element from *arr* then call ArrDoOver( ), your application will crash. If you delete any other element, you will get inconsistent results.

## 2.10 ArrCopy

```
ArrCopy(arr) -> arr2
```

ArrCopy( ) makes a copy of *arr*. The copy is complete, except the ArrDoOver( ) state is not copied. After the copy is made, the two arrays are independent of each other.

## 2.11 ArrDrop

```
ArrDrop(arr[, key, ...]) -> 0
```

ArrDrop( ) either removes one or more key values from *arr* or removes all values from *arr* and frees the array memory. It's a good idea to clean up memory when you're finished with it (this is not only true for RxHash's data – it applies to stem variables and even ordinary rexx variables). If *key* is not found, nothing happens, but it's not treated as an error. If *arr* is not a value originally returned by ArrNew( ), ArrCopy( ), or some other library which uses the C interface to generate arrays, the process is likely to crash.

## 2.12 ArrToStem

```
ArrToStem(arr, stemname) -> 0
```

ArrToStem( ) drops the stem *stemname*, then, for each element of array *arr*, sets the corresponding tail of *stemname*. If a default value has been specified for *arr*, it is assigned to *stemname*.

The effect of this is to copy the array values into the stem.

## 2.13 ArrFromStem

```
ArrFromStem(stemname) -> arr
```

ArrFromStem( ) returns a new array which is populated with all the data from stem *stemname*. If a value has been assigned to the stem itself, this is made the default value of *arr*.

The effect is to copy the stem values into the array.

## 2.14 Passing arrays to subroutines

The principal motivation behind RxHash was to have something that acts like an array and can be passed to functions. The normal situation in ANSI Rexx is that you must expose a stem, and either have the stem's name be 'well known', or pass the name as an argument.

The return code of ArrNew( ) can be assigned and passed to subroutines like any other variable, with one proviso: it's actually a pointer, so this code:

```
height = ArrNew()
call ArrSet height, 'giraffe', 'tall'
colour = height
call ArrSet colour, 'giraffe', 'spotty brown'

animal = 'giraffe'
say 'name' animal 'height' ArrGet(height, animal) 'colour' ArrGet(colour, animal)
```

prints 'name giraffe height spotty brown colour spotty brown', which is not what was desired. In this case, a sensible person would probably use ArrNew( ) on each array, but in other cases it would make sense to copy the array using ArrCopy( ).

## 2.15 Arrays returned by other packages

At the time of writing, there are no packages which use this library, however there is a C interface and I expect some users will take advantage of it in their own extension libraries. The expectation is that some routines will return arrays populated with certain data. These arrays will be created using exactly the same routines as the RxHash routines, and will be fully inter-operable.

# 3 C Functions

The C interface is intended for application developers who want to return values in RxHash arrays, rather than setting a stem. They could also be used as a general hash library, but they're not especially tuned for performance, and I'm sure there are better hash libraries available.

The Rexx array variables are simply the pointer value returned by rxhash_new( ). The macros RXHASH_TO_RXSTRING( ) and RXSTRING_TO_RXHASH( ) assist in converting between the rxhash_t and RXSTRING types.

You need to include rxhash.h in your source file. On Unix, you usually don't have to link against rxhash, but if you don't the user must load rxhash in the Rexx program before calling routines from your library. On NT and OS/2, you need to link against rxhash.lib. If your compiler doesn't understand the structure of rxhash.lib, you can probably generate a compatible library from rxhash.def using tools supplied with your C compiler.

The hash function was taken from CWeb, by Knuth and Levy, and the routines are adapted from an awk interpreter I wrote a few years ago.

## 3.1 rxhash_new

```
rxhash_t tbl;
tbl = rxhash_new();
```

rxhash_new( ) initialises a new hash table. It returns NULL if memory allocation failed.

## 3.2 RXHASH_TO_RXSTRING

```
rxhash_t tbl;
PRXSTRING result;

tbl = rxhash_new();
RXHASH_TO_RXSTRING(result, tbl);
```

This macro copies the contents of *tbl* to the Rexx string *result*. You need to ensure *result* has enough space to hold the pointer (sizeof(*tbl*)). Note that the resulting Rexx string is a pointer value which is meant to be passed to array-processing C routines. It is not a printable string.

## 3.3 RXSTRING_TO_RXHASH

```
rxhash_t tbl;
PRXSTRING result;

RXSTRING_TO_RXHASH(tbl, result);
```

This macro copies the contents of the Rexx string *result* to *tbl*. You need to ensure *result* is the right size.

## 3.4 rxhash_set

```
int rxhash_set(rxhash_t tbl, PRXSTRING key, PRXSTRING val);
```

rxhash_set( ) associates *val* with *key* in array *tbl*. It returns 0 if this operation failed (usually due to memory allocation failure), or 1 if it's successful. There's no way to determine whether the value was previously set (you can call rxhash_get( ) first if this is important).

*key* must be non-null and the string must not be zero-length. *val* must be non-null, but the string may be zero-length.

## 3.5 rxhash_get

```
PRXSTRING rxhash_get(rxhash_t tbl, PRXSTRING key);
```

rxhash_get( ) retrieves the value associated with *key* in array *tbl*. If *key* is not found, it returns NULL. In contrast to ArrGet( ), you can distinguish between this value and a successful return value, hence rxhash_get( ) takes the place of both ArrGet( ) and ArrIn( ). In contrast to many C associative array interfaces, *key* is not added to the array by rxhash_get( ) if it's not already there.

## 3.6 rxhash_drop

```
void rxhash_drop(rxhash_t tbl, PRXSTRING key);
```

rxhash_drop( ) drops a single element associated with *key* from array *tbl*. It's not possible to determine whether *key* was set before being dropped.

## 3.7 rxhash_delete

```
void rxhash_delete(rxhash_t tbl);
```

rxhash_delete( ) drops all elements from array *tbl* then frees the memory used to hold the array.

## 3.8 rxhash_iterate

```
int rxhash_iterate(rxhash_t tbl, PRXSTRING prevkey,
                   PRXSTRING * pkey, PRXSTRING * pval);
```

rxhash_iterate( ) returns the 'next' element after *prevkey* from array *tbl*. It returns 1 if there is a 'next' element, or 0 if there isn't. *prevkey* should be NULL, in which case the interation starts from the beginning of the table, or a value previously returned in *pkey*.

*pkey* and *pval* point to buffers which will be used to store pointers to the key and value entries. These pointers are valid as long as the associated key value is not deleted from *tbl*. It's probably not a good idea to change the array while you iterate through it, though.

The state information stored against each table for ArrDoOver( ) is not used by rxhash_iterate( ). In fact, the function is re-entrant. There are no restrictions on how many times you iterate through a given array at the same time, although more than once seems excessive, and you need to ensure you use different variables for *pkey* and *pval*.

Keys are returned in an unpredictable order – it depends on the hash algorithm and the order in which keys were added.

## 3.9 rxhash_setprop

```
int rxhash_setprop(rxhash_t tbl, PRXSTRING key, PRXSTRING val);
```

rxhash_setprop( ) sets a property of *tbl*. *Key* can be any name understood by the caller – properties are a mechanism for associating arbitrary data with an array for the use of the caller. Names starting with 'rxhash_' are reserved for the use of the RxHash library.

## 3.10 rxhash_getprop

```
PRXSTRING rxhash_getprop(rxhash_t tbl, PRXSTRING key);
```

rxhash_getprop( ) retrieves the value of the *key* property of *tbl*. If *key* is not found, it returns NULL.

As with rxhash_get( ), the value returned by rxhash_getprop( ) is valid until the property is dropped. Since there is no function for dropping properties, that means the value is valid until the *tbl* is deleted.