REXX, the developer's best friend

By Paul Gallagher

Whenever I think of REXX, I picture a golden brown Basset hound. He's always by my side, panting and slobbering a bit, eager to run around and do some tricks for me. When I throw a ball, off he scampers, eager to please. His legs are a bit short, but boy do they pump him along.

Yep, that's m' boy, good ol' REXX. A developer couldn't have a more faithful companion.

Introduction

OS/2 provides a tremendous environment for the software developer. I'm sure we all know that by now! As the toothpaste advertisements go, "OS/2. Recommended by more software developers than any other operating system".

This article will focus on one feature of OS/2 that can really help improve the way you build software, perhaps more than any other -- **REXX**. Now, I'm not so much talking about developing in REXX as using REXX to assist your development in some other language (which could be REXX however, but could equally be C or C++).

We're going to do some basic to intermediate REXX scripting which I'll present in tutorial format. At the end we'll have a few fully functional developer's tools. That's only half the story though; I intend to take a few detours along the way and get philosophical about the software development process and developer tools.

I guess I have two audiences in mind. If you are fairly new to REXX, or have never used it before, then I hope the tutorial nature of the presentation will help you along the learning curve. Many developers could be reasonably familiar with REXX though. If you are in this camp, then if nothing else I hope I can stimulate your imagination somewhat, and perhaps encourage you to use REXX for tasks that you wouldn't have otherwise considered.

The REXX Renaissance

Some Background for Beginners

It seems interest in REXX has never been stronger, and I suspect that's largely thanks to OS/2 (remember, REXX is available on a wide range of platforms). The Internet newsgroup "comp.lang.rexx" is lively and EDM/2 is full of reviews of REXX-based tools. The IBM Employee Written Software (EWS) scheme has provided some great REXX support; there are now libraries that enable REXX to access a large portion of the OS/2 API, including networking features.

The key to REXX is its refreshing combination of simplicity and power. It is particularly strong in handling text strings and patterns. For example, the simple REXX statement

```
Parse Var tempstring word1 word2 therest
```

will split the contents of the variable "tempstring" into three variables ("word1", "word2" and "therest"). Everything in the source string up until the first space character goes into "word1", the text between the first and second space goes into "word2" and the remaining characters (if any) get put into "therest". Note also that our use of the space character as the delimiter is purely arbitrary - it could have been '#', '^' or any other character (or combinations of characters).

REXX is arguably one of the best text processing languages around. It is easier and more flexible than awk, quicker and easier to develop than C and other 3GLs, and more widely supported than some specialist text processing engines (e.g. the DOS Shareware program "Parse-O-Matic").

Developing software is a complex activity with many aspects. However, when you consider that producing **code**, particularly for 3GL languages, is largely a "text processing" activity, you perhaps get an inkling that REXX may have something to offer when it comes to helping your software development activities.

Developer productivity? ME???

Developer productivity - a personal issue

Most people reading this article would call themselves software developers, at least to some extent. Perhaps you are paid to write software; I'm sure many more wish they were, but only get to cut code at home while at work they have other responsibilities.

Any developer, whether working for profit or just for fun, must surely be interested in being as productive as possible. After all, the more time you save, the more work you can do (and as we all know, developing in GUI, networked environments usually takes a **lot** of work - in terms of lines of code).

Have you ever sat back and made an objective assessment of your 'productivity'? If you're paid to program, then I'm sure your managers make sure that the productivity issue is always in your face. But are you like me? Good habits at work, everything out the window when I'm sitting at my home computer?

Anyone answer in the affirmative? ;-)

Well, perhaps I'm not that bad (I hope;-), but I recognize that I tend to suffer from another ailment - "DOS-upsized-tunnel-vision". Like many of you, my first programming experience was under DOS. I am used to having low expectations of the operating environment. DOS batch files can only do so much. To do anything beyond copying and concatenating files etc, you need to turn to additional tools - tools you have to find (or buy) yourself. I guess partly in response to the paucity of standard tools, compiler products tended to come with a lot of functionality built-in (like Borland's IDE). Even adding Windows to the equation didn't help: it was a boon to the functionality that the average programmer could build into a program, but Windows itself didn't help us develop any better (far from it - productivity plummeted).

Now that I've "upsized" to a decent **operating** environment - OS/2 - I tend to forget that my **development** environment doesn't end at the edge of the IDE.

I would suggest that two features of OS/2 - the Workplace shell and REXX - set it apart from most other operating systems/environments when it comes to the software development process. The Workplace gives us a powerful space in which to organize the files and programs that "belong" to a project, and REXX gives us a supremely powerful scripting tool to automate many of the basic activities involved in setting up and working on a project.

My Scripting Style

This article includes a good half-dozen scripts, but they are all based on the same basic template. Rather than tediously describe the scripts in full, over and over, I'll take this opportunity to introduce you REXX scripts a la Paul Gallagher.

Here's the 8-point blueprint I use:

1. Header
This is basically a non-executable section containing comments, description and version history notes. I generally define three global variables here: programNameStr (a text description of the script), copyrightStr (appropriate copyright message) and versionStr (version message). These variables are mainly used in banner messages and help screens.

You will note a number of cryptic tags (beginning with "****" at around column 35) in the header - these are actually instructions for the version control software I use (TLib).They only pop up in a few places so I have not removed them for publication (they also let you know that these are **real** scripts!).

Example:

```
/*--------------------------------------------------------------------------*/
  '@echo off'
  programNameStr=    "Create new project (example only)"
  copyrightStr=      "Copyright (c) Paul Gallagher 1995"

/*                                   ***keywords*** "Version: %v  Date: %d %t"*/
  versionStr=        ""
/*
;                                    ***keywords*** "%l"
; LOCK STATUS       ""
;
;                                    ***keywords*** "%n"
; Filename          "newproj.cmd"
; Platform          OS/2 (REXX)
;
; Authors           Paul Gallagher (paulg@resmel.bhp.com.au)
;
; Description
;
; Revision History
;                                    ***revision-history***
;                                    ***revision-history***
;--------------------------------------------------------------------------*/
```

2. Load REXX utility functions [optional]
Next, I load the REXX utility package, if any of the functions will be required by the script. There tend to be a few schools of thought in relation to loading/unloading external functions. I prefer to load the entire standard utility package - and **not** unload functions at the end of the script. I do this since the load/unload is a global operation - not limited to the one process.

Example:

```
/*-----------------------------------------------------------------------------
; Load REXXUTIL
;-----------------------------------------------------------------------------*/
If RxFuncQuery('SysLoadFuncs') <> 0 Then
  If RxFuncAdd('SysLoadFuncs','RexxUtil','SysLoadFuncs') <>0 Then Do
    Say 'Unable to init REXX Utility function loader.'
    Exit
  End
Call SysLoadFuncs
```

3. Install selected error traps
As with any other programming language, it is important to ensure that programs fail as gracefully as possible as possible when severe errors are encountered. All error conditions cause program flow to jump to "ExitProc" - the point from which a reasonably clean exit can occur.

Example:

```
/*-------------------------------------------------------------------------
; Set error traps
;-------------------------------------------------------------------------*/
signal on failure name ExitProc
signal on halt name ExitProc
signal on syntax name ExitProc
```

4. Initial parse of command line [optional]
If the command arguments are of any interest at all, they are initially loaded into a variable called "params". The code that follows does a quick check for anything that looks like the user is after help - if that is the case, then the "HelpInfo" procedure is called prior to jumping to the script exit point.

Example:

```
/*-------------------------------------------------------------------------
; Do initial parse of command line and call help message if required
;-------------------------------------------------------------------------*/
                                    /* get the command line arguments */
Parse Arg params
                                    /* call help routine if required */
If POS(TRANSLATE(params),"-?"'00'x"/?"'00'x"-HELP"'00'x"/HELP") > 0 Then Do
  Call HelpInfo
  Signal ExitProc
End
```

The check for a "help" parameter illustrates a neat trick that is handy for command parsing. TRANSLATE(params) simply does an uppercase conversion. All of the valid "help" commands are concatenated as a string with intervening null characters. The POS function is then used to test whether the parameters passed to the program match any of the "help" commands. Thus, the parameters "-?", "/?", "/help", "-H" etc etc will all be recognized as calls for help.

5. Main program
This is where the "generic" stuff ends, and the real program starts

6. General Exit Procedure
The main exit point of the script is named "ExitProc". Normal execution of the main program will simply see control flow to this point, but error conditions will generally see execution jump to this point. Prior to exiting completely, variables are dropped (mainly to be neat than for any other reason) - but other tasks could be included here if required.

Example:

```
/*-------------------------------------------------------------------------
-
; General exit procedure
;-------------------------------------------------------------------------
*/
ExitProc:
  Drop params programNameStr copyrightStr versionStr
Exit
```

7. Standard help procedure
A standard help procedure is provided, and usually enhanced for each individual script. It is intended to simply display an appropriate message, but other features could be added if required.

Example:

```
/*-------------------------------------------------------------------------
; routine to display help message
;-------------------------------------------------------------------------*/
HelpInfo: Procedure Expose programNameStr copyrightStr versionStr
  Say
```

```
  Say "*=================================================================*"
  Say "    "programNameStr
  Say "    "versionStr
  Say "    "copyrightStr
  Say
  Say "*=================================================================*"
Return
```

8. Additional functions/procedures
The standard template ended at point 7. What follows are all the functions and procedures written for a specific application.

Coding and documentation standards
In addition to following the blueprint outlined above, the code adheres to some basic presentation guidelines:

1. Program blocks indented by 2 characters
2. Keywords typed in proper case (eg. "If" or "Select")
3. Function names typed in uppercase (eg. "TRANSLATE")
4. Variables name components in proper case, but name started with lower case (eg. "programName")
5. All comments that document algorithms are indented by 35 characters (ie. comments appear to the right of the page). Comments that precede a procedure or block of code begin at column 1.

Setting up your development environment

In this section I'll talk about a selection of REXX scripts that can help you setup and maintain a development environment.

INFICONS.CMD
A great deal of developer information comes packaged as INF files (just like EDM/2). As always, INF files tend to get scattered all over your disk - and searching for the right one can be a difficult task since they invariably have 8 character filenames (and if they come from IBM, the name will probably be particularly cryptic;-).

This is a fairly simple script searches for INF files on your disk. The script takes an optional parameter to specify the path and/or filemask used in the search for INF files. By default it will search C: drive, examining all sub-directories for *.INF files. I must admit that this is not the first script to be written with this purpose in mind - but it's the only one I know of that assigns the INF's true title to the icon created.

After parsing the command line to determine the appropriate search mask (variable "mask"), the main algorithm to search the disk and process files is as follows:

```
If SysFileTree(mask, 'file', 'FSO') > 0 then
                                  /* out of memory message */
  Say SysGetMessage(8)
Else Do
                                  /* if files found then process */
  if file.0>0 Then Do

    /* ... create folder for all INFs
       if it doesn't already exist */

                                  /* loop through all INFs found */
    Do i=1 to file.0

      /* ... for each INF, get its
         title & create icon */
```

```
    End
  End
  Else
                                    /* file not found message */
    Say SysGetMessage(2)
End
```

This is a fairly generic algorithm to search-and-process files that could be used in other situations.

Workplace Shell icons are created for all INFs found using the SysCreateObject function. Icons are created in a folder called "INF Files" on the desktop. The enclosing folder is created with the following command:

```
Call SysCreateObject "WPFolder", "INF Files", "<WP_DESKTOP>",,
                     "OBJECTID=<INF_FILES>"
```

Translated: this command creates a "WPFolder" on the desktop called "INF Files" and assigns it the object ID <INF_FILES>. Note the extra comma on the first line - this is the method REXX uses to indicate the command continues on the next line.

INF icons are created with the command:

```
Call SysCreateObject "WPProgram", title' ['path']', "<INF_FILES>",,
          "EXENAME=VIEW.EXE;PARAMETERS="name";STARTUPDIR="path";",,
          "UPDATE"
```

where "path" is the fully qualified path of the INF file, "name" is the filename, and "title" is its true title. Basically, an icon has been created for the VIEW.EXE program which is passed the INF filename as a parameter and is executed in the INF file's directory.

The "UPDATE" parameter to the SysCreateObject function requests that object attributes be updated, rather than additional objects created if a duplicate is encountered. In writing this script I discovered that if a WPProgram object had a multi-line name, then an object match was never detected and additional objects always created - something to watch out for.

The INF file's true title is obtained by peeking inside the INF file. The title is a null terminated string up to 48 characters long, starting at offset 6B(hex). The "peek" is easily achieved with two lines of code:

```
Call CHARIN file.i,1,X2D('6B')
Parse Value CHARIN(file.i,,48) with title'00'x
```

The first line "gobbles" the first 6B characters (NB: file.i is the INF file name). The next line reads the 48 characters that constitute the title field, and uses the Parse command to assign all characters up to the first null to the variable "title".

Once we have the INF filename ("name", located in directory "path") and true title ("title"), creating an icon for it requires a simple variation on the SysCreateObject we have seen already:

```
Call SysCreateObject "WPProgram", title' ['path']', "<INF_FILES>",,
             "EXENAME=VIEW.EXE;PARAMETERS="name";STARTUPDIR="path";",,
             "UPDATE"
```

NEWPROJ.CMD

Starting a new programming task or project involves a bit of administration to get you going. Aside from perhaps cleaning up your desk a bit, emptying the ashtray and so on, you need to set up an appropriate

computing environment (set up a project directory, check-out some standard libraries etc). I'm here to tell you that REXX can't help with the first tasks (unfortunately) - but it can certainly help with the later.

As an example, here's what I'll typically do when starting a new C or C++ project:

1. Create a project directory (I'll also create directories for the different target platforms - DOS, OS/2 etc)
2. I use TLib version control software, so I'll create a project-specific TLib configuration file, and create a project directory in my TLib archive area.
3. I may use TLib to check out a copy of my (personal) standard library for use in the project
4. Create a Workplace Shell folder for the project. I'll make this folder a "work area", and fill it up with some objects: a shadow of the project directory (from 1); an OS/2 Command Line icon with startup directory set to the project directory; icons for any compilers or tools I intend on using - with startup directories set to the project directory where appropriate.
5. Copy some program stubs as a starting point for coding.

Phew! I haven't even got started yet, but if I had to do this manually I would already have wasted a good half an hour. Fortunately its all pretty mundane stuff - easy to automate with REXX. Unfortunately, its also all very idiosyncratic - chances are, what I've describe is similar to what you do, but we'll always have our differences. For that reason, the NEWPROJ.CMD script I describe here is only an example - it will need some heavy customization for it to meet you own requirements.

After asking for the name of the new project, NEWPROJ.CMD does two things: creates some directories (the project's "home" directories); and then some Workplace shell objects.

Directories are created using the utility function SysMkDir:

```
Call SysMkDir prjDir
```

We could just have easily (but with a speed penalty) invoked the built-in command 'md' (ie. 'mkdir'):

```
'md' prjDir
```

To "personalize" this script, you may wish to create some basic project files at this stage: customized software configuration files; initialize some project log files etc etc.

The Workplace Shell objects are a bit more interesting. Firstly, the script creates a folder on the desktop called "REXX - The developer's best friend"; this folder will be a container for all the individual project folders.

```
Call SysCreateObject "WPFolder", "REXX - The developer's best friend",,
                     "<WP_DESKTOP>","OBJECTID=<REXX-TDBF>;"
```

After creating a project folder (as a workarea ie when this folder is closed, all child objects are also closed)..

```
SysCreateObject("WPFolder", n "Project", "<REXX-TDBF>",,
                "OBJECTID=<SPRJ-"n">;WORKAREA=YES;","UPDATE")
```

..it is populated with a selection of objects. Firstly, it creates icons for the Borland C++ compilers - OS/2, DOS and Windows version (if you use these products you should examine the NEWPROJ script for details of the required SysCreateObject commands). An OS/2 Command Prompt icon is also created, as well as a shadow of the project directory ("d"; "n" is the project name)..

```
Call SysCreateObject "WPProgram", "OS/2 Command Prompt", "<SPRJ-"n">",,
                "EXENAME=*;STARTUPDIR="d";","UPDATE"
```

```
  Call SysCreateShadow d, "<SPRJ-"n">"
```

As I have already mentioned, NEWPROJ.CMD is probably of little use as it stands, but is a good basis on which to build your own "New Project Initiation" script.

STUBS.CMD

Most developers have a pretty clear idea about how they like their source files organized. You may follow the more common conventions, or use a style of your own. Either way, you don't want to be typing all the same fluff for each new file created.

The most common solution to this problem is to keep a selection of templates. Rather than type a header, copyright info and so on - just copy the template and away you go. Two things could be improved though: you may end up with a clutter of templates, and the template may still need a bit of tweaking on a case-by-case basis.

STUBS.CMD allows you to automate your use of templates ("stubs"). Running STUBS first presents the user with a menu from which they can select the template to be generated. After providing a new filename, the script generates a file based on the selected template. Of course, being a REXX script it can include all kinds of smarts when generating the template - such as including a copyright notice with this years date (instead of the date from when you created the template;-).

So far its a good concept - but you may ask how I (as the guy who wrote STUBS.CMD) know how you want your templates produced. The answer is - I don't! In fact, STUBS.CMD comes as a basic menu shell but contains no actual templates. The one useful command it provides you (other than "exit") is "modify". Here is the basic menu structure:

```
Do Forever
                                    /* display menu */
  Say
  Say "Select from the following commands:"
/*FLAG1* DO NOT DELETE THIS LINE - New menu items inserted above*/
  Say "   ------------------------------------------------"
  Say "   MODIFY: add a new template to this file"
  Say "   EXIT: end processing"
  Call CHAROUT ,'> '
  Pull Cmd
                                    /* process menu option */
  Select
  When ABBREV("EXIT",Cmd) Then
    Signal ExitProc
  When ABBREV("MODIFY",Cmd) Then
    Call AdminMODIFY
/*FLAG2* DO NOT DELETE THIS LINE - New menu items inserted below*/
  Otherwise
    Nop
  End
End
```

As you can see, very simple. A menu is printed, command accepted and then processed. Using the ABBREV function to examine a command makes for a very user-friendly interface reminiscent of VMS (is that an oxymoron??). Users can type as few or as many letters of a command as they like; insofar as the letters they have typed match the actual keyword, then the command is recognized.

The "MODIFY" command allows you to add a new template to STUBS.CMD. You provide an example file, and the script reads it in, creating the necessary procedures and code modifications in STUBS.CMD to support the new template type. You will notice the hard-coded "tags" in the preceding code - these help

the script locate the menu code that requires extending. The code to write the new templates is encapsulated in procedures which are simply appended to the file.

When adding a new template, you are asked for three bits of information: the filename of the file to be used as the example template; the keyword to be used as the menu command; and a description of the template - this is used in the menu display.

To modify STUBS.CMD, you first need to locate the scripts true file name and path. This is easily done using the "Source" variant of the "Parse" command:

```
Parse Source . . SourceFile
```

Next, the SourceFile (STUBS.CMD) is read into a queue. The queue provides a temporary storage area prior to writing back a modified STUBS.CMD, and is much more convenient than messing about with temporary files and so on.

```
                                    /* read stubs.cmd to queue */
Do While LINES(SourceFile) > 0
  line = LINEIN(SourceFile)
  queue line
End
```

Once the file has been fully enqueued, write-back begins by first repositioning the file pointer to the start of the file...

```
Call LINEOUT SourceFile,,1
```

...an then pulling the source from the queue, writing each line back to file - inserting the new menu display and processing commands as we go:

```
Do While QUEUED() > 0
  Parse Pull line

                                    /* insert new menu item */
  if (POS("/*FLAG1",line)=1) Then Do
    Call LINEOUT SourceFile,'  Say "   'key': 'description'"'
  End
                                    /* write current line */
  Call LINEOUT SourceFile,line

                                    /* insert new menu-processing commands */
  if (POS("/*FLAG2",line)=1) Then Do
    Call LINEOUT SourceFile,'  When ABBREV("'key'",Cmd) Then'
    Call LINEOUT SourceFile,'    Call Create'key
  End
End
```

After that, the new template-writing procedure is appended to the source file. At its core, this is a simplified variation of the procedure used in the QUOTE.CMD script to read the example file and convert it to commands that can re-write the example (see the section on text filters for more details). In part:

```
                                    /* insert sample script */
pre="Call LINEOUT f,'"
post="'"
Do While LINES(sample) > 0
  line = LINEIN(sample)
  new=""
                                    /* replace conflicting quote chars in
```

```
                                        source line */
  Do While POS("'",line)>0
    Parse Var line frag"'"line
    new=new''frag"'"
  End
  new=new''line
                                  /* write command to write 'clean' line */
  Call LINEOUT SourceFile,'  'pre''new''post
End
```

In my experience, such self-modifying scripts appear to be quite safe. Although I don't know the details of the process, I suspect that the REXX command processor reads and semi-compiles/interprets a script before beginning execution (the compiled form of the script is also stored in extended attributes for re-use). This means a script can safely read and modify itself without altering the execution path of the current program instance.

How to best use STUBS? I suggest that you marshal your most commonly used templates and put a bit of effort into cleaning them up. Once they've checked out OK, use the MODIFY command to add them to STUBS.

At this stage, STUBS.CMD will be a self-contained script that can re-create your original templates on command. You can stop right there, however you may wish to delve in and customize the template-writing procedures to add extra capabilities.

For example, my personal STUBS.CMD can create C, C++ and REXX templates. I have converted all hard-coded dates (in the copyright legends for example) to calculated values so that the templates never go out of date. The C/C++ template procedures contain other enhancements. For example, after asking for a name stem, they produce both an include (*.h) and associated source files (*.c or *.cpp). The source files #include the matching header file.

More Ideas

I've already hinted that I use REXX to assist with version and project control activities (to great effect). In fact, my own variant of the NEWPROJ.CMD script is growing in scope and capabilities almost daily - it maintains version control configurations, checks-in/out and updates standard libraries, packages software for distribution and so on. As you would appreciate, I cannot really present these systems in this article because they are too specialized and peculiar to my personal development environment. But there's a lesson in there somewhere: I'm sure that most of you would be in a similar situation of having unique requirements, and I can only encourage you to examine what you do and consider whether REXX can give you a hand!

Text Filters

I made the point that REXX is particularly good at text processing, and indeed, it is a natural choice for implementing a whole raft of "text filter" type utilities.

Text filters are basically programs that read a text file from the standard input stream, modify or process it in some way (line by line), and write the resultant file to the standard output stream. This means that the standard operating system pipe and redirection functions can be used to select input and output files or device handles. Typical text filter usage is as follows, using the OS/2 SORT.EXE filter as an example:

```
type c:\config.sys | sort > lpt1:
```

The basic filter structure in REXX is a very simple 'Do' loop which reads from the standard input (using the LINEIN function) until no more lines are available (ie. LINES() = 0).

```
                                    /* loop until no lines available at
                                       standard input */
Do While LINES() > 0
                                    /* read current line */
  line = LINEIN()

  /* ... perform some processing */

End
```

LINE.CMD
This is the good old "print line number x" program (it actually prints line "x" and the 3 lines before and after). Implementing this requires only a few extra lines added to the basic filter script. Basically, we count lines and only print the ones we want.

TAIL.CMD
Again, an oldie but a goodie - print the last "x" lines of a file. When setting out to implement this specification, there are two obvious problems: we don't know how many lines are in the file - until after we've read it; and what's the best way of buffering the file so that you can come back and process it after you've worked out which lines to print.

The queue interface in REXX gives us an elegant way of solving the problems. As we read the source file, we write the lines to a first-in-first-out (FIFO) queue using the QUEUE keyword. Once we've read "x" lines, we PULL (discard) a line from the front of the queue for each new one we add - this way the queue becomes a sliding window containing the most recently read "x" lines.

```
Do While LINES() > 0
  line = LINEIN()
  lc=lc+1
                                    /* enqueue the new line */
  Queue line
                                    /* if we already have our quota, also
                                       discard a line from the front of
                                       the queue */
  If (lc>params) Then
    Pull line
End
```

By the time we get to the end of the file, the queue contains exactly what we need to print. A simple process:

```
Do While QUEUED() > 0
  Parse Pull line
  Say line
End
```

QUOTE.CMD
This script helps you prepare text for inclusion as print statements in C/C++ or REXX programs. For example, the line

```
I said "I'm here!"
```

may be converted to the C statement

```
printf("I said \"I'm here!\"");
```

In fact there are four output formats supported:

1. C stdio

```
printf("I said \"I'm here!\"");
```

2. C++ Iostreams

```
cout << "I said \"I'm here!\"" << endl;
```

3. REXX - Say keyword

```
Say 'I said "I''m here!"'
```

4. REXX - CHAROUT function

```
Call LINEOUT f,'I said "I''m here!"'
```

The QUOTE.CMD script uses the basic text filter model to process the input stream. Lines are prepared for output using five variables:

1. ("pre") a prefix is prepended to the line
2. ("post") a suffix is appended to the line
3 & 4. Nominated quote characters ("qchar") that appear within the text itself are substituted by a quote character replacement string ("reqchar").
5. The process of replacing quote characters requires another variable ("qqchar"), which takes the value of "qchar" in REXX-quoted format (more on that later!)

So, to produce a C++ Iostream compatible output, we define these 5 quantities as follows:

```
pre='cout << \"'
post='\" << endl;'
qchar='"'
qqchar=''''''''          /* the value of qqchar is '"' ie qchar */
reqchar='\"'
```

By using this generic "quoting" model, it is easy to add new variations.

The trickiest part of the procedure (shown below) is the replacement of embedded quote characters ("qchar") with a substitute string ("reqchar"). If it was to be a simple character for character replacement, then the TRANSLATE function could have come in handy, however there are situations where the quote character needs to be replaced with more than one character. The solution I have chosen is to repeatedly parse the input line into left and right fragments separated by the first occurrence of the quote character. The right-hand fragment becomes the subject of the next iteration - this process continues until the quote character can no longer be found in the remaining fragment. Because the delimiter between left and right portions is a variable quantity, we need to construct a command and request REXX to interpret it on the fly - this allows us to make the delimiter appear as a constant in the Parse template.

```
Do While LINES() > 0
  line = LINEIN()
  new=""

                                  /* replace qchar occurences in source text
                                     with reqchar */
  Do While POS(qchar,line)>0
    cmd = "Parse Var line frag"qqchar"line"
    interpret cmd
    new=new''frag''reqchar
  End
```

```
                                    /* tack on any remaining line to new */
    new=new''line
                                    /* print the quoted line */
    say pre''new''post
  End
```

You may not need the QUOTE.CMD itself - but this example of string substitution may be of assistance in other scripts you write. NB: I'm not altogether happy with this approach to string substitution; if anyone has a better suggestion I'd be glad to hear from them!

More Ideas

I've presented a few generic text filters. It's pretty obvious though that the sky's the limit when you start considering special purpose filters. Here are some suggestions...

... there are quite a few code formatters already available (mostly free) but these may not do exactly what you want. Perhaps you or the companies you work for have some special coding requirements. Developing a code formatter can be a fairly involved process (depending on how deep into the syntax you needed to peek to make the required modifications), but it would be easier in REXX than C!

... need to update copyright notices embedded in source files? If you use a consistent format, then a REXX script to do the update for you would be trivial.

... half way through a project you decide to change your variable or function naming conventions. If you have many source modules, a quick REXX script may be the easiest way of implementing the change.

Conclusion

REXX is a fantastic language for text processing - enormously powerful (yet simple) features such as the Parse command set it apart from its competitors.

Coupled with the ability to manipulate Workplace Shell objects, REXX clearly has a lot to offer the developer - not only as a language for coding the end product, but as I've explored in this article: as a tool for assisting development.

Hopefully the scripts presented - if not immediately useful to you - will give you ideas for other applications.

APPENDIX: Summary of REXX scripts presented

INFICONS.CMD
Invocation:
```
INFICONS [path][filemask]
```
Example usage:
```
INFICONS
INFICONS C:\OS2\BOOK
```
This script will search for INF files on disk, and create icons for them in a folder called "INF Files". The icons will be named according to the INF file's true title (not its filename).

LINE.CMD
Invocation:
```
LINE x
```
Example usage:
```
type file.txt | LINE 50
```

This script prints the specified line number of a file to the screen, along with the 3 lines immediately before and after.

NEWPROJ.CMD
Invocation:
```
NEWPROJ
```
Example usage:

QUOTE.CMD
Invocation:
```
QUOTE [c|c++|rexx|rexxf]
```
Example usage:
```
type file.txt | LINE 50
```
This script formats lines for inclusion in C/C++ or REXX procedures. The parameter indicates the formatting system used.

STUBS.CMD
Invocation:
```
STUBS
```

TAIL.CMD
Invocation:
```
TAIL x
```
Example usage:
```
type file.txt | TAIL 5
```
This script prints the last 'x' lines of a text file to the screen.