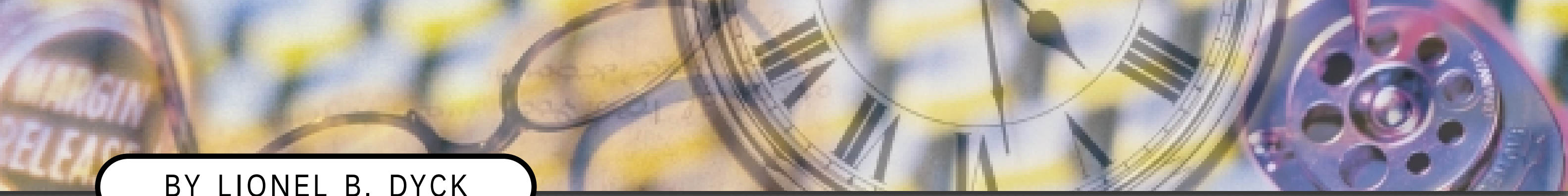

BY LIONEL B. DYCK

An Introduction to REXX for OS/390 Users

This article provides an OS/390 user with the necessary information to code a simple REXX program and to use the provided IBM documentation to build complex applications in REXX. Additionally, the author demonstrates how to write a program in REXX to read every member of a PDS and report on every member that contains a particular string.

REXX is a popular programming language that Michael Cowlishaw of IBM developed for the VM environment. The language was ported to MVS and OS/2, and today you can find versions of REXX for most platforms. There is also NetRexx, which can generate Java and Object REXX if you are into objects. Additionally, there is object-oriented (OO) REXX.

The definitive book for REXX beginners is *The REXX Language* by M.F. Cowlishaw (ISBN 0-13-780735-X Prentice-Hall, 1985, ISBN 0-13-780651-5 second edition, 1990). Note that on OS/390 the IBM softcopy publications are excellent reference guides.

THE BASICS

OS/390 REXX is a procedural language with a simple syntax. With most programming languages you have to spend a great deal of time learning the coding style rules, where to put parentheses, when to use brackets, how to use quotes (single or double), how to comment, and how to continue a statement across multiple records. With REXX, you will learn these things in less time than it takes to learn when to use braces in C or C++.

Note that OS/390 REXX is also an interpretive language. There is no compile, link, or execute required. Just put the REXX program in the proper DD concatenation and execute it. Although there are several REXX compilers available, including one from IBM, this article will not address them.

The goal of this article is to teach you how to code in REXX by demonstrating how to actually code a simple REXX program that will:

Note that OS/390 REXX is also an interpretive language. There is no compile, link, or execute required. Just put the REXX program in the proper DD concatenation and execute it.

- obtain a list of all members of a partitioned data set
- read each member
- report on each member that contains a specific string

Figure 1 shows the entire REXX code sample for this demonstration.

THE FIRST STATEMENT

The first statement in any REXX program should be a comment containing the word "REXX". This comment line is a standard in OS/390 REXX that lets the REXX program be included in the SYSPROC or the SYSEXEC DD concatenation. Figure 2 shows an example of valid first statements.

COMMENTS

Comments in REXX appear between the slash asterisk (/*) and the asterisk slash (*/). These delimiters can appear on the same record or you can separate the delimiters by multiple records.

Note: You can include additional slash asterisks (/*) within a comment; however, when the system encounters the first slash asterisk (*/), it ends your comment.

As shown in Figure 2, the first statement is just a comment.

HINT

Under ISPF Edit, enter Hilite and a pop-up panel appears. I like to set the language type explicitly to REXX. I set coloring to 3 so that both the IF and DO logic are colored. I then place a slash (/) next to each of the following: parentheses matching, highlight FIND strings, and highlight cursor phrase. This will make coding easier, as it will hilite any mismatched quotes and/or comments.

PICKING UP ARGUMENTS FROM THE COMMAND LINE AND VARIABLE NAMES

A REXX program executes like a CLIST because the user can specify options when issuing the command. For this program, you can specify two options: the name of the partitioned data set and the name of the string to report on. The code to capture the options is as follows:

```
Arg pds_dsname string
```

The ARG instruction tells REXX to access the command line arguments, and pds_dsname and string are the variables where you place the arguments. A variable name can be anything that begins with an alphabetic or national character (for example, $@#!?_) but not a number. The name can contain any alphanumeric or national character. I will discuss other variable naming conventions, including the stem variables, later.

Once you place the arguments into the variables, you need to verify the existence of both arguments. Figure 3 shows the code for verification.

Statements 1 and 7 are IF statements which test the variable for a null (“”). Statements 2 and 8 are THEN statements that follow the IF to indicate that some action will be performed. The DO statement indicates processing for a single pass DO loop. The DO loop ends with the END statement (statements 6 and 12). Within the DO loops, the SAY instruction writes output to the screen consisting of whatever follows. Statements 5 and 11 cause the REXX program to terminate immediately with a return code of 8. The indentation is not critical to REXX but is helpful to the human coder and reader.

VARIABLES AND LITERALS

By default, variables will be the value of the variable name. Thus, a variable name of pds_dsname will have a default value of PDS_DSNAME. Literals are any character string in either single or double quotes. Note that the case of the variable name is not relevant.

FIGURE 1: COMPLETE REXX PROGRAM SOURCE

```
/* - - - - - - - - - - - -  rexx procedure  - - - - - - - - - - - -*
 * Name:      DoAll                                                *
 * Syntax: %DOALL pds string                                       *
 * - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - */

arg pds_dsname string

If pds_dsname = “”
   Then do
        Say “Error. No arguments entered.”
        Say “Ending.”
        Exit 8
        End
If string = “”
   Then do
        Say “Error. No program name entered.”
        Say “Ending.”
        Exit 8
        End

if sysdsn(pds_dsname) <> “OK” then do
   say “Error:” pds_dsname sysdsn(pds_dsname)
   exit 8
   end

call outtrap “trap.”

“LISTD” pds_dsname “MEMBERS”

call outtrap “off”

do i = 1 to trap.0
   if trap.i = “—MEMBERS—” then leave
   end

call listdsi pds_dsname
dsname = sysdsname

i = i + 1
do j = i to trap.0
   parse value trap.j with  mem
   mem = strip(mem)
   say “Checking Member:”
   “Alloc f(xxin) ds(‘“dsname”(“mem”)’) shr reuse”
   “Execio * diskr xxin (finis stem in.”
   “Free  f(xxin)”
   hit = 0
   do z = 1 to in.0
      if pos(string,in.z) > 0
         then hit = 1
end
   if hit = 1 then
      say “Found in Member:” mem
 end
```

FIGURE 2: VALID FIRST STATEMENTS

```
/*  rexx */
/*  program.exec - a rexx program */
/*  program.exec - a rexx program
    that does something */
```

TESTING THE VALIDITY OF THE INPUT PDS

Next, it is helpful to verify that the input pds provided is a valid data set name. You can use the code shown in Figure 4 to do your verification.

Statement 1 uses the SYSDSN function, which uses a parameter of the input data set name to test if the data set exists. The <> is the same as saying, “not equal,” and the test is for a literal of OK. If the result of the SYSDSN test is not equal to OK, then a DO Loop is executed.

Statement 2 issues an error message starting with the literal of “Error:” followed by the input data set name. The results of the SYSDSN function call follow, which provide a short phrase describing the error.

Statement 3 will exit the REXX program with a return code of 8, and Statement 4 ends the DO Loop.

GETTING THE LIST OF PDS MEMBERS

To process every member of a partitioned data set you will use the TSO command LISTD. This code segment will demonstrate the use of a TSO command, the use of the OUTTRAP instruction to capture output normally written to the TSO screen, a somewhat more complex DO loop, the use of the PARSE instruction, the use of stem variables, and the use of the CALL instruction. The OUTTRAP instruction is shown in Figure 5.

The first statement in Figure 5 calls the OUTTRAP REXX function. The parameter to OUTTRAP is the stem where you place all trapped records. OUTTRAP can capture or trap anything written using a PUTLINE instruction, which many TSO commands use.

In Statement 2, execute the LISTD TSO command, which is enclosed in quotes to indicate it is a literal. The REXX interpreter will try to execute LISTD TSO as a TSO command because it is a statement of its own. The first parameter to LISTD is the name of the data set to process and the second parameter is the literal MEMBERS to tell LISTD to list all members.

Statement 3 calls OUTTRAP again with the instruction to cease trapping output.

Statement 4 is a DO LOOP that uses the variable "i", setting it to 1 to start with and processing until "i" is the same as the value in trap.0. The variable "i" is incremented by one until it matches the value in trap.0. OUTTRAP places the number of records captured into the .0 stem value. Stem trap.1 would be the first real record, trap.2 the next, and so forth.

Statement 5 looks for the string "—MEMBER—" in position 1 of the trapped record, as the member names start on the record immediately after this record. If the program has the LEAVE instruction, it will cause the current loop to terminate. When the DO loop terminates, the value of "i" will be the same as the record where the "—MEMBER—" string was found.

FIXING UP THE PDS DSNAME

The provided partitioned data set name now needs to be fixed so that the code can

FIGURE 3: CODE TO VERIFY ARGUMENTS

```
1. If pds_dsname = “”
2.    Then do
3.    Say “Error. No arguments entered.”
4.    Say “Ending.”
5.    Exit 8
6.    End
7. If string = “”
8.    Then do
9.    Say “Error. No string entered?”
10. Say “Ending.”
11. Exit 8
12. End
```

support either a fully qualified data set name (one enclosed in quotes) or a partially qualified data set name. A partially qualified data set name requires that the user TSO PREFIX be appended to the data set name. The quickest way to do this is to use the LISTDSI function as follows:

```
1. call listdsi pds_dsname
2. dsname = sysdsname
```

Statement 1 is the call of LISTDSI with a single parameter of the PDS data set name. LISTDSI has many options and other capabilities. LISTDSI will create a number of variables related to the data set. Check the documentation for the complete list.

Statement 2 sets the variable dsname with the fully qualified, but without quotes, data set name as returned from the LISTDSI function.

PROCESS THE MEMBERS

Figure 6 shows the code to process the members. Notice the indentation here, so that the statements within a specific DO LOOP are easily identifiable. This is a matter of style, and you should find a comfortable style for coding that is easily readable by others who may inevitably be maintaining this code.

Statement 1 increments the variable "i" by one. You can specify the variable "i" either in upper- or lower-case because it refers to the exact same variable. The variable "i" is incremented because it was last set with the record number of the record with the literal string "MEMBERS". This increment points to the next record from the LISTD results, which contain the first member name.

Statement 2 is a DO Loop using the variable J as the counter and counting from I to the value in trap.0, which is the number of records that were captured by the OUTTRAP function.

Statement 3 uses the PARSE function to extract the member name from the current record.

Statement 4 uses the REXX Strip function to remove leading and trailing blanks from the mem variable. The current record is found in trap.j, where j is the current record count of interest when working with the counter in the DO Loop.

PARSE is a very powerful function of REXX. In this case, the PARSE VALUE format tells REXX to parse using the Value of trap.j and separate that value into variable(s) mem.

Note that PARSE has a lot of other capabilities, including allowing multiple variables to be used which will take each distinct set of characters, separated by blanks, and insert the set into a respective variable. Since this example only specifies one variable, mem, this is where the first set of characters belong (in this case only).

To fully understand PARSE, refer to the publications listed in the References section at the end of this article.

Statement 5 uses the SAY function to inform the user on the screen about the processing status of the member name.

Statement 6 is the TSO command Alloc. Note that the command is defined as a literal as are all the keywords for Alloc. The only variables are the dsname and mem. Note the use of the double quotes to define the literals, leaving the single quote for use in qualifying the data set name.

Statement 7 is the REXX function EXECIO, which lets you read and write files. EXECIO does the following:

- the * indicates to process all records
- the diskr indicates to Read the records from disk
- the xxin is the DD Name where the records will be read from

- the parentheses indicate additional parameters for EXECIO to follow
- the finis keyword indicates to close the DD when the EXECIO finishes
- the stem indicates that the records are to be read into a stem variable
- the in. is the stem variable

As you can see with this simple example, the EXECIO is a very powerful function. Check out the References section for more information on EXECIO.

Statement 8 is the TSO command Free, which releases the allocation created in the Alloc command.

Statement 9 sets the variable hit to zero.

Statement 10 is the DO Loop that will process the member that has just been read into the in. stem variable using the variable z as the counter.

Statement 11 is a test. If the position of the value in the variable string is found in the text of the variable in.z at position greater than 0, then the variable hit is set to 1.

The Position function (POS) finds the relative position of the first string within the second string. A result of 0 (relative position 0) indicates that there was no match. As you can see from this statement, the THEN can be included on the same record. The THEN keyword indicates that when the previous test is true, THEN executes the following function (which may be a single function, a larger function, or a larger function contained within the DO/END or SELECT/END function).

Statement 12 is the END of the DO Loop started in Statement 10.

Statement 13 tests the variable hit for a value of 1. If the value is 1, then Statement 14 is processed, which uses the SAY function to tell the user on the screen that string searched for was found in the member currently being processed.

Statement 15 is the END of the DO Loop started in Statement 2.

CONCLUSION

I hope that this article has whetted your appetite for coding in REXX. As you can see, it is a very simple, yet very powerful language. A good source to learn about REXX is to look at REXX code from others. You can find a number of useful examples at my web site as well as on the CBT Tape web site, which contains several hundred files, several of which contain REXX code.

With REXX you can write ISPF dialogs, query DB2 databases, process SMF records, do involved math, access storage locations, and more using both standard functions provided with REXX or add-on functions provided with various products (e.g., ISPF). You can use REXX on many different platforms. The basic REXX language on OS/390 is an interpretive language; however, if you need the performance improvements of a compiled language there are several REXX compilers available.

FIGURE 4: USE OF SYSDSN TO VERIFY THAT A DATASET EXISTS

```
if sysdsn(pds_dsname) <> “OK” then do
   say “Error:” pds_dsname sysdsn(pds_dsname)
   exit 8
end
```

FIGURE 5: USE OF OUTTRAP

```
Call outtrap “trap.”
“LISTD” pds_dsname “MEMBERS”
Call outtrap “off”
do i = 1 to trap.0
   if trap.i = “–MEMBERS–” then leave
   end
```

FIGURE 6: THE ACTUAL CODE PROCESS

```
1. i = i + 1
2. do j = i to trap.0
3.    parse value trap.j with  mem
4.    mem = strip(mem)
5.    say “Checking Member:”
6.    “Alloc f(xxin) ds(‘“dsname”(“mem”)’) shr reuse”
7.    “Execio * diskr xxin (finis stem in.”
8.    “Free  f(xxin)”
9.    hit = 0
10. do z = 1 to in.0
11.    if pos(string,in.z) > 0 then hit = 1
12. end
13. if hit = 1 then
14.    say “Found in Member:” mem
15. end
```

REXX is a full language that I am sure you will find a place for in your toolkit.

REFERENCES

The following are excellent resources for finding out more information about REXX:

- The IBM OS/390 Softcopy Web Server: www.s390.ibm.com/os390/bkserv/
- The REXX Language Association: www.rexxla.org
- The IBM REXX home page: www2.hursley.ibm.com/rexx/
- Lionel Dyck’s home page: www.geocities.com/lbdyck
- Dave Alcock’s web site: www.planetmvs.com/freeware/fwrexx.html
- CBT Tape: Accessible using www.naspa.com

NaSPA member Lionel Dyck is a lead MVS systems programmer for a large HMO in California. He has been in systems programming since 1972, and has written numerous ISPF dialogs over the years. He is an active member of NaSPA and SHARE, and can be reached via email at Lionel.B.Dyck@kp.org.