Creating REXX Functions

Author

Patrick Talley
Senior Systems Programmer
Altec Industries, Inc.
patalley@altec.com

Overview

In this article, I will describe how to create functions in REXX and in HLASM. To illustrate this, I have written the same function in both languages. This function is part of a utility application I wrote to assist me on a project to eliminate Shareoption(4) files. I needed to be able to distinguish COBOL keywords from variable names so I could properly identify the files being opened in a program.

Besides creating a function, there were other options for doing this – like hard coding several IF statements into a REXX procedure or using a large SELECT routine. I discarded these ideas since there are over 300 reserved COBOL keywords, and I wanted a way to do this identification efficiently.

My goal in the main REXX program was to pass a character string (which could be a COBOL/VSE reserved word or variable) and get a simple "yes" or "no" answer after performing a binary search. This is exactly what can be accomplished using a REXX function.

I originally was only going the write this function in REXX, but I ran into some problems getting it to work the way I wanted it to. I came back to it after I got an HLASM version to work. Thus I now can show the same function in REXX and HLASM for your benefit, as well as my own.

Required REXX Keywords

Writing a function in REXX is fairly easy, since there are only three keywords needed to create a skeleton for a function: **PROCEDURE**, **ARG** and **RETURN**.

The function is called by using it in place of a REXX expression. This can appear in different formats, but the simplest is:

```
answer=name(var1,var2).
```

The variables passed to the function can be any REXX expression, including a function call. However, you can not pass an array in the form of a stem variable. Thus `answer=name(stem.)` is invalid.

Internal and External Functions

There are two types of REXX functions you can define – internal and external. The main difference is where the function is defined. An internal function is in the same PROC member as the main calling routine. An external function is contained in a separate PROC member.

To start the definition of a internal function, use the PROCEDURE keyword in the format:

```
name: PROCEDURE
```

If you want to share data between the function and the calling routine, there is an optional parameter, `EXPOSE varname`, for the PROCEDURE keyword. This allows the function to use and update these variables, and should be used with caution.

The PROCEDURE keyword is omitted for an external function. Instead, the function begins with a `/* comment */` line like a regular REXX procedure.

The function name is the name of the PROC the function is contained within. Since there is no PROCEDURE keyword, you have to pass all of the data in the function call.

Receiving Values

To receive values that are passed to the function use, the `ARG` keyword in the format:

```
ARG var1, var2
```

Ending a Function

To end the function, use the `RETURN` keyword in the format:

```
RETURN(expression)
```

The expression is an optional parameter, where the result is passed back to the calling routine. The expression is typically a variable name, but it can be any REXX expression such as a calculation or a function call. The RETURN keyword does not have to be physically at the end of the function. It can be placed anywhere that the function can logically return to the calling routine, in multiple locations if necessary.

Recursive Calls

One of the nice things about writing a function in REXX is that it allows for recursive calls. The binary search routine I used is a recursive routine, so it lent itself to being implemented in REXX. However, this does require a slightly different mind set from the typical programming logic, especially if you are not accustomed to structured programming. My function calls itself for each table entry to be examined for the keyword passed to it.

Logic Issues

Some of the logic issues that I ran into were:

- How to keep all of the function logic out of the main program.
- Where to determine the address of the next table entry to be checked.
- Which data values needed to be passed forward to the next function call.

The solution to the first issue was to make my function *two* functions. My main program first called **rxcbrsv**, which initialized my array with COBOL/VSE reserved words and variables for determining the first entry to be checked in the array. Then this function called the second-level function **rxrsvrd**, which performed a binary search of the table. See Figure 28 on page 75.

Assembler Program RXCOBRSV

Loading the table was one of the problems I had with implementing the function in REXX. Since it is an interpretive language, the logic for loading the array is executed each time the main program calls the function. I could get around this by loading the array in the main program and using the EXPOSE option so it will be available to the function, but that defeats the idea of having a self-contained function and keeping that logic out of the calling program. This is what prompted me to first implement this function in HLASM (Figure 29 on page 76).

Remaining Issues

The other two issues are really related and probably could be resolved in several different ways. What I decided on was to calculate the size of the subset to be searched, the offset to the midpoint of that subset and the address of the entry to be checked in the calling function, and pass them forward along with the keyword being searched for.

I wish I could say I came to this decision after careful thought and planning, but in reality I arrived at this through trial and error. I consider this to be another advantage of writing a function in REXX. With the "say" and "trace" commands the debugging process is quick and easy.

Unless you are more comfortable with Assembler code than REXX, writing a function in HLASM is more difficult and time-consuming than it is for REXX. The good new is that the interface for getting the input data from the REXX program and returning the result is simple.

Finding an Example: The first step that I would recommend in creating a function in Assembler is to try and find an example that is close to what you are trying to write. I started by looking through the examples directory of the VSE Collection CD-ROM and found a sample REXX function in **RXPROG52**. This basically provided me the logic for handling the input and output routines for the function. I uploaded this sample program and used it as a skeleton program for my REXX function. I then used **IGY8RWRD.Z** to create a table of the reserved words for the lookup. I was not able to find any sample code for performing a binary search, so I had to develop that logic on my own.

Understanding Linkage Logic: The next step was to refer to the *VSE/REXX Reference* manual to understand the linkage logic used in the sample program. The REXX linkage data areas are created through three macros – **ARXEFPL**, **ARXARGTB** and **ARXEVALB**.

ARXEFPL Defines the external function parameter list (EFPL). There are two key addresses in the EFPL, EFPLARG which points to the argument list (the input data) and EFPLEVAL which points to the evaluation block (the result to be passed back).

```
EFPLARG                           DS A
EFPLEVAL                          DS A
```

ARXARGTB Defines the argument list. The list is an array of addresses and lengths ending with two fullwords of high-values marking the end of the array.

```
ARGTABLE_ARGSTRING_PTR            DS   A
ARGTABLE_ARGSTRING_LENGTH         DS   F
```

ARXEVALB Defines the evaluation block. EVALBLOCK_EVSIZE is the size of this data area in double words. EVALBLOCK_EVLEN is the length of the result the function returns. It is initialized to X'80000000'. EVALBLOCK_EVDATA is the area for returning the result to the calling REXX procedure.

The size of the EVALBLOCK_EVDATA is EVALBLOCK_EVSIZE * 8 - 16. This defaults to 256 bytes. If you need to return more than 256 bytes of data, you can use the ARXRLT routine to acquire a larger area.

Before returning back to the calling routine you will need to load the length of the data being returned to EVALBLOCK_EVLEN and of course move the result into EVALBLOC_EVDATA.

```
EVALBLOCK_EVSIZE               DS   F
EVALBLOCK_EVLEN                DS   F
EVALBLOCK_EVDATA               DS   C
```

Register Usage: Finally, the last thing you need to be concerned about is the register usage. Upon entry into the function:

- Register 0 points to the environment block (defined by ARXINIT) of the calling program,
- Register 1 points to the EFPL,
- Register 13 points to the register save area,
- Register 14 is the return address and
- Register 15 points to the entry point in the function.

Once you have the logic for handling input and output for REXX completed, all that is left is regular Assembler programming. In my case, I got a good start because I had already figured out the basic logic from writing the function in REXX. I did put in a little extra effort on defining the table so I could easily update or replace it.

Incorporating an External Function into a Function Package

The performance of an external function that is called frequently can be improved by incorporating it into a *function package*. This is because an external function is reloaded into the partition each time it is called, whereas a function package is loaded only once.

Creating a function package only requires a little additional work. First, you need to pick a name for the function package. There are two default names that can be used: **ARXFLOC** and **ARXFUSER**. If you choose to use another name, you will need to create a customized ARXPARMS module or call the ARXINIT routine to make it available to your REXX program. Also note that some of the software packages on your system already may use the default function name, so take this into consideration when you pick a name.

The next step is to assemble the HLASM function and catalog the object module that will be link-edited into the function package. This also can be done with a REXX function if you have a REXX compiler.

The final step is to create the function package directory. There is no macro to assist in the creation of the directory, but there is a macro, **ARXFPDIR.A** in PRD1.BASE, that supplies the field names and layout.

The directory has two sections, a header and the entries. Three of the fields in the header are constants:

1. **FPCKDIR_ID** is set to ARXFPACK,
2. **FPCKDIR_HEADER_LEN** is the length of the header and will always be set to X'00000018', and
3. **FPCKDIR_ENTRY_LENGTH** is the length of the directory entry and will always be set to X'00000020'.

The fourth field, **FPCKDIR_FUNCTIONS**, contains the number of directory entries in the function package.

```
FPCKDIR_ID                    DC CL8
FPCKDIR_HEADER_LEN            DC F
FPCKDIR_FUNCTIONS             DC F
FPCKDIR_ENTRY_LENGTH          DC F
```

The function package will have one directory entry for each function or subroutine in the function package. Each directory entry has three fields to be set:

1. **FPCKDIR_FUNCNAME** is the name of the function pointed to for the entry,
2. **FPCKDIR_FUNCADDR** is the address of the entry point for the function, and
3. **FPCKDIR_SYSNAME** is the name of the entry point for the function.

```
FPCKDIR_FUNCNAME              DC CL8
FPCKDIR_FUNCADDR              DC A
FPCKDIR_SYSNAME               DC CL8
```

FPCKDIR_FUNCADDR and FPCKDIR_SYSNAME serve the same purpose, and only one has to be specified. Please note that there are a few reserved fields in the header. For the complete layout of the directory and the entries that I have omitted from this article, refer to **ARXFPDIR.A** in PRD1.BASE or to the manual, *VSE/REXX Reference*. Figure 30 on page 78 shows my sample code for ARXFLOCL.

Performance

To illustrate the performance issues with the different ways you can define functions, I ran four tests. In each test, I called the function 33,903 times using the reserved word array along with three extra words to verify the logic of the function.

- Case one used the HLASM function.
- Case two used the HLASM function incorporated into a function package.
- Case three used an external REXX function.
- Case four used an internal REXX function and loaded the reserved word array once in the main program and passed it the function via the EXPOSE option.

As you can see below, the external REXX function has the worst overall performance. There is a significant I/O overhead associated with the HLASM function and the external REXX function. This comes from loading the function each time the function is called. The I/O overhead is eliminated for the HLASM function by it into a function package.

Table 1. Performance Comparison

Test Case	CPU Seconds	I/Os Performed
HLASM function	75.652	135,789
HLASM function package	27.224	218
External REXX function	2,627.15	1,186,721
Internal REXX function	480.169	152

Additional Information

When choosing which type of function you want to use, you should consider the following:

1. Does the function require any language specific logic?
2. How often will the function be called from the main program?
3. How many different REXX programs will call the function?

If your function requires specific logic or capabilities of REXX or HLASM, that will dictate the use of that language for the function. If the function will be called several thousand times from the same main program, you should rule out an external function. If only a few REXX programs need to call the function, you should consider an internal REXX function. If the function will be used heavily, the best choice would be a HLASM function incorporated into a function package.

Please note that a zipped file with my code is available via the VSE/ESA home page. To get this file, go to:

http://www.ibm.com/s390/vse/vsehtmls/s390ftp.htm

Look for **rexxfunc.zip**.

```
* $$ JOB JNM=RXCBRSV,DISP=D,CLASS=T,PRI=4
* $$ LST DISP=H,CLASS=P
// JOB RXCBRSV
// EXEC LIBR
ACCESS S=OEM.TEST
CATALOG RXCBRSV.PROC      DATA=NO REPLACE=YES
arg key_word
  /*****************************************************************/
  /* routine to test reserved routine                              */
  /*****************************************************************/
  rsvrd_word.0=339
  rsvrd_word.1='ACCEPT              '
  rsvrd_word.2='ACCESS              '

...

  rsvrd_word.338='ZEROES              '
  rsvrd_word.339='ZEROS               '
chk_index = ((rsvrd_word.0 +1) % 2)
search_size = rsvrd_word.0 - ((rsvrd_word.0 + 1) % 2)
mid_point=((search_size + 1) % 2)
return(rxrsvrd(key_word,chk_index,search_size,mid_point))
rxrsvrd: procedure expose rsvrd_word.
arg key_word,chk_index,search_size,mid_point
ANS='NO  '
search_size_n = search_size - mid_point
mid_point_n=trunc((search_size_n + 1) / 2)
if key_word = rsvrd_word.chk_index then do
  ANS='YES '
  return(ANS)
end
if search_size > 0 then do
   select
     when key_word > rsvrd_word.chk_index then do
       chk_index = chk_index + mid_point
       return(rxrsvrd(key_word,chk_index,search_size_n,mid_point_n))
     end
     when key_word < rsvrd_word.chk_index then do
       chk_index = chk_index - mid_point
       return(rxrsvrd(key_word,chk_index,search_size_n,mid_point_n))
     end
     otherwise do
     ANS='YES '
     return(ANS)
     end
   end
end
return(ANS)
/+
/&
* $$ EOJ
```

Figure 28. RXCBSRV

```
* $$ JOB JNM=RXCOBRSV,DISP=D,CLASS=T
* $$ LST DISP=D,CLASS=P
// JOB RXCOBRSV ASSEMBLE A REXX ASSMBLER FUNCTION
// OPTION CATAL
 PHASE RXCOBRSV
// LIBDEF PHASE,CATALOG=OEM.TEST
// EXEC ASMA90,SIZE=(ASMA90,300K),PARM='EXIT(LIBEXIT(EDECKXIT))'
          REGEQU
          CSECT RXCOBRSV
          USING *,R15
          SAVE  (R14,R12)
          BALR  R10,R0
          USING *,R10                      BASE REG 10
          LR    R2,R1                       LOAD REG 2 FROM 1
          USING EFPL,R2                    POINT TO PARMLIST
          L     R3,EFPLARG                 POINTER TO ARGUMENT LIST
          L     R9,EFPLEVAL                SAVE EVALUATION BLOCK POINTER
          USING ARGTABLE_ENTRY,R3
          L     R4,ARGTABLE_ARGSTRING_PTR POINT TO FIRST ARGUMENT
          L     R5,ARGTABLE_ARGSTRING_LENGTH LOAD ARGUMENT LENGTH
          CLRRTN KEYWORD
*  R0 = TABLE ENTRY SIZE
*  R1 = TABLE SEARCH SIZE
*  R2 = 2*TABLE ENTRY SIZE
*  R3 = POINTER INTO TABLE
*  R4 = PREVIOUS TABLE SEARCH SIZE
          LM    R0,R1,TABLE
          CR    R5,R0
          BH    NOTFND
          BCTR  R5,R0                     SUBTRACT 1 FOR EX
          EX    R5,EXMVC
          LA    R3,TABLES
          LR    R4,R1
TABHIGH   A     R1,=X'00000001'
          SRL   R1,1           CALCULATE MIDPOINT(R1/2)
          LR    R11,R1
          MH    R11,TABLE+2
          AR    R3,R11         GET POSITION INTO TABLE
          CH    R1,=X'0000'     IS OFFSET LESS THAN OR EQUAL TO TWO
          BNH   ENDCHK         YES, THEN CHECK LAST TWO ENTRIES
*                              TABLE ENTRIES?
```

Figure 29 (Part 1 of 3). RXCOBRSV

```
TABLOW   SR    R4,R1          SET UP R4 AND
         LR    R1,R4                 R1 FOR NEXT PASS
         CLC   0(L'TABLES,R3),KEYWORD
         BE    TABFND
         BL    TABHIGH
         A     R1,=X'00000001'
         SRL   R1,1          CALCULATE MIDPOINT(R1/2)
         LR    R11,R1
         MH    R11,TABLE+2
         SR    R3,R11        GET POSITION INTO TABLE
         CH    R1,=X'0000'    IS OFFSET LESS THAN OR EQUAL TO TWO
         BNH   ENDCHK        YES, THEN CHECK LAST TWO ENTRIES
         B     TABLOW
ENDCHK   SR    R3,R0         SHIFT BACK
         CLC   0(L'TABLES,R3),KEYWORD
         BE    TABFND
         A     R3,TABLE      SHIFT TO NEXT ENTRY IN TABLE
LSTCHK   CLC   0(L'TABLES,R3),KEYWORD
         BE    TABFND
NOTFND   LA    R7,NO                  SET R15 TO ONE
         B     SAVRTN
TABFND   LA    R7,YES                    SET R15 TO ZERO
SAVRTN   L     R8,0(R9)                   POINT TO EVALBLOCK
         USING EVALBLOCK,R8
         LA    R6,X'00000004'          SET REPLY LENGTH
         ST    R6,EVALBLOCK_EVLEN      STORE REPLY LENGTH IN EVALBLOCK
         MVC   EVALBLOCK_EVDATA(4),0(R7)
         SR    R15,R15                  SET R15 TO ZERO
         ST    R15,16(R13)              SAVE R15 IN SAVE AREA
         RETURN (R14,R12)               RETURN TO CALLING FUNCTION
EXMVC    MVC    KEYWORD(1),0(R4)
KEYWORD  DS     CL20
ADDTAB   DC     A(TABLES)
YES      DC     C'YES '
NO       DC     C'NO  '
TABLE    DC     A(L'TABLES)
ENTRIES  DC     A((TABLEE-TABLES)/L'TABLES)
TABSIZE  DC     A(TABLEE-TABLES)
         LTORG
```

Figure 29 (Part 2 of 3). RXCOBRSV

```
TABLES    DC     C'ACCEPT              '
          DC     C'ACCESS              '

...

          DC     C'ZEROES              '
          DC     C'ZEROS               '
TABLEE    EQU    *
          DS    0F
          ARXARGTB
          ARXEFPL
          ARXEVALB
          END
/*
// EXEC LNKEDT,PARM='AMODE=31,RMODE=ANY'
/*
/&
* $$ EOJ
```

Figure 29 (Part 3 of 3). RXCOBRSV

```
* $$ JOB JNM=ARXFLOC,DISP=D,CLASS=T
* $$ LST DISP=D,CLASS=P
// JOB ARXFLOC ASSEMBLE A REXX FUNCTION PACKAGE
// OPTION CATAL
 PHASE ARXFLOC
// LIBDEF *,SEARCH=TEST.BASE,CATALOG=OEM.TEST
// EXEC ASMA90,SIZE=(ASMA90,300K),PARM='EXIT(LIBEXIT(EDECKXIT))'
ARXFLOC   CSECT ARXFLOC        REXX FUNCTION PACKAGE DIRECTORY
FPCKDIR_ID          DC  CL8'ARXFPACK' FPCKDIR CHARACTER ID
FPCKDIR_HEADER_LEN  DC A(FPCKDIR_HEADER_END-FPCKDIR_ID)
FPCKDIR_FUNCTIONS   DC A((FPCKDIR_END-FPCKDIR_FUNCNAME)/(FPCKDIR_ENTRY_*
               END-FPCKDIR_FUNCNAME))
                    DC  F'0'     RESERVED
FPCKDIR_ENTRY_LENGTH DC A(FPCKDIR_ENTRY_END-FPCKDIR_FUNCNAME)
FPCKDIR_HEADER_END        EQU *
FPCKDIR_FUNCNAME    DC CL8'RXCOBRSV'   NAME OF FUNCTION OR SUBROUTINE
FPCKDIR_FUNCADDR          DC  V(RXCOBRSV)  ADDRESS OF THE ENTRY POINT
*                                      of the package code
                          DC  F'0'     RESERVED
FPCKDIR_SYSNAME           DC  CL8' '    NAME OF THE ENTRY POINT
*                                      corresponding to package code
FPCKDIR_SYSDD             DC  CL8' '   RESERVED              @VSEDH
FPCKDIR_ENTRY_END         EQU *
FPCKDIR_END               EQU *
          END
/*
 INCLUDE RXCOBRSV
// EXEC LNKEDT,PARM='AMODE=31,RMODE=ANY'
/*
/&
* $$ EOJ
```

Figure 30. ARXFLOC