Introduction to TSO/E REXX

Why Use REXX?

- **Two Languages In One**
    - Logic and control language
    - Host command language

- **Simple But Powerful**
    - Integration into other products
    - Only a few keyword instructions
    - English words for instructions
    - Structured language
    - Free format coding - readability
    - Extensive parsing capabilities
    - Many built-in functions

- **Excellent Debugging Capabilities**
    - Error messages
    - Interactive tracing and debugging

About REXX

What is Rexx?

Rexx is a procedural programming language that allows programs and algorithms to be written in a clear and structured way. It is easy to use by experts and casual users alike. Rexx has been designed to make easy the manipulation of the kinds of symbolic objects that people normally deal with such as words and numbers. Although Rexx has the capability to issue commands to its host environment and to call programs and functions written in other languages, Rexx is also designed to be independent of its supporting system software when such commands are kept to a minimum.

General programming using Rexx

Rexx provides powerful character and arithmetical abilities in a simple framework. It can be used to write simple programs with a minimum of overhead, but it can also be used to write robust large programs. It can be used for many of the programs for which BASIC would otherwise be used, and its layout may look somewhat similar to that of a structured BASIC program. Note, however, that Rexx is **not** BASIC!

A simple program in Rexx looks like this.

```
   /* Count to ten and add the numbers up */
   sum = 0
   do count = 1 to 10
      say count
      sum = sum + count
   end
   say "The sum of these numbers is" sum"."
```

Macro programming using Rexx

Many applications are programmable by means of macros. Unfortunately, in the Unix world, almost every application has a different macro language. Since Rexx is essentially a character manipulation language, it could provide the macro language for all these applications, providing an easy-to-use and consistent interface across all applications. The best examples of such systems are on CMS (IBM's mainframe operating system which was the birthplace of Rexx) and on the Amiga. However, IBM's OS/2 is catching up, and now that Rexx is freely available on Unix it cannot be long before applications start to appear which have Rexx as their macro language. Two products already exist. They are the Workstation Group's Uni-XEDIT and Mark Hessling's THE (a link to which is displayed on my Rexx title page).

Rexx treats any instruction that it doesn't immediately recognise as an expression which it evaluates and passes to the host environment as a command. A simple XEDIT macro in Rexx looks like this.

```
   /* this XEDIT macro centres the line containing the cursor. */
   width = 72                        /* Width within which to centre the
line */
   "extract /cursor /curline"     /* location of cursor and current line
# */
   if cursor.3=-1                              /* if cursor is not on a
line... */
   then "emsg Cursor not in a data line"       /* then give an error
message */
   else do
      restore=curline.2-cursor.1             /* how far cursor is from
current */
      ":"||cursor.3                                  /* make cursor line
current */
```

```
      "extract /curline"                             /* get the current
line */
      "replace" centre(strip(curline.3),width)    /* centre the current
line */
      restore                                     /* restore old current
line */
   end
```

Other applications of Rexx

Rexx can be used as an "application glue" language, in a manner similar to that in which shell scripts are often used. Since Rexx is able to pass arbitrary command strings for execution by its environment, it can be used to execute Unix programs as well as providing the control language necessary for testing things such as parameters and return codes and acting accordingly.

Rexx is often executed by an interpreter, and this permits rapid program development. This productivity advantage makes the language very suitable for modelling applications and products - in other words, for prototype development. Rexx is also fairly easy to debug. Once a design has been shown to work satisfactorily, it can be easily recoded in another language if that is required for performance or other reasons.

The design of Rexx is such that the same language can effectively and efficiently be used for many different applications that would otherwise require the learning of several languages.

REXX Instructions

- **Keyword –** SAY, PARSE, IF…Then…Else, DO…
- **Assignment –** Variable assignment statements, e.g., VAR1 = 20
- **Label –** Used with subroutines and functions, e.g., MYLABEL:
- **Null –** comment lines (/*…..*/) and blank lines
- **Commands –** dependent on the environment, TSO, MVS, ISPF, USS

Always begin a REXX routine with a comment line containing the word REXX.

Example:

```
/* REXX   this is a REXX routine */
SAY 'Enter your name'
PULL name
IF name = '' THEN
   SAY 'No name entered'
ELSE
   SAY 'Hello' name
EXIT
```

TSO/E REXX Keyword Instructions

ADDRESS - issue host command / change host environment
ARG - retrieves argument strings in upper case - same as PARSE UPPER ARG
CALL - call a routine (internal, built in, or external)
DO - group instructions, repetition (looping)
DROP - "unassigns" variable(s)
EXIT - exit a program unconditionally - return to caller
IF - conditionally processes an instruction or group of instructions
INTERPRET - process dynamically built instructions after evaluation
ITERATE - skip to next loop iteration
LEAVE - leave loop
NOP - no operation - useful as the target of a THEN or ELSE clause
NUMERIC - set arithmetic operations options
OPTIONS - set language processor options
PARSE - parse string using a template
PROCEDURE - enter subroutine, new variable pool
PULL - read string from top of data stack - same as PARSE UPPER PULL
PUSH - add a string to top of data stack
QUEUE - add a string to bottom of data stack
RETURN - return from function / subroutine call
SAY - write a line to the terminal
SELECT - conditionally call one of several alternative instructions
SIGNAL - set error trap or branch to label
TRACE - set/reset tracing options
UPPER - translate variables to uppercase (see TRANSLATE function)

Assignments and variables

A variable is a character or group of characters that represents a value. Its value can change during the running of a REXX program. You change the value of a variable by assigning a new value to it.

A variable name, the part that represents the value, is always on the left of the assignment statement and the value itself is on the right. In the following example, the word "var1" is the variable name, and its value is "5".

```
var1 = 5
```

The value of a variable may be:

- A constant, which is a number that is expressed as:

  An integer (12)
  A decimal (12.5)
  A floating point number (1.25E2)
  A signed number (-12)
  A string constant (' 12') or ('C1'x)

- A string, which is one or more words that may or may not be enclosed in quotation marks, such as:

  ```
  THIS VALUE IS A STRING.
   'This value is a literal string.'
  ```

- The value from another variable, such as:

  ```
  variable1 = variable2
  ```

  In the above example, variable1 changes to the value of variable2, but variable2 remains the same.

- An expression, which is something that needs to be calculated, such as:

  ```
  variable2 = 12 + 12 - .6          /* variable2 becomes 23.4 */
  ```

Before a variable is assigned a value, its value is its own name translated to uppercase characters. For example:

```
var1 = orange
```

If _orange_ has not been assigned a value then _orange_ has the value **ORANGE** and _var1_ then has the value **ORANGE**.

Compound variables

Compound variables can be used in REXX to store groups of related data in such a way that the data can be easily retrieved. Compound variables create a one-dimensional array or list of variables in REXX. For example, a list of employee names can be stored in an array called **employee** and retrieved by number.

EMPLOYEE
(1) Adams, Joe
(2) Crandall, Amy
(3) Devon, David
(4) Garrison, Donna

In some computer languages, you access an element in the array by the number of the element, such as, employee(1), which retrieves "Adams, Joe". In REXX, you use compound variables.

A compound variable contains at least one period and at least two other characters. It cannot start with a digit or a period, and if there is only one period in the compound symbol, it cannot be the last character.

abc.d

is a compound variable. It is comprised of a stem, **abc.**, followed by simple variables called the tail. The derived name of a compound symbol is the stem of the symbol, in uppercase, followed by the tail, in which all simple symbols have been replaced with their values.

Examples:

```
/* rexx                                       */
d = 5     /* assigns '5' to the variable D    */
e = 6     /* assigns '6' to the variable E    */
abc.d = "Hello"      /* "Hello"  to  abc.5    */
abc.e = "there"      /* "there"  to  abc.6    */
SAY abc.d abc.6 abc.f
/* displays        Hello there ABC.F          */

/* rexx                                       */
a=3       /* assigns '3' to the variable A    */
z=4                 /*   '4'      to Z        */
c='Fred'            /*   'Fred'   to C        */
a.z='Fred'          /*   'Fred'   to A.4      */
a.fred=5            /*   '5'      to A.FRED   */
a.c='Bill'          /*   'Bill'   to A.Fred   */
c.c=a.fred          /*   '5'      to C.Fred   */
y.a.z='Annie'       /*   'Annie'  to Y.3.4    */

say  a  z  c  a.a  a.z  a.c  c.a  a.fred y.a.4
/* displays the string:                       */
/*    "3 4 Fred A.3 Fred Bill C.3 5 Annie"    */
```

You can use a DO loop to initialize a group of compound variables and set up an array.

```
DO i = 1 TO 4
   SAY 'Enter an employee name.'
   PARSE PULL employee.i
END
```

If you entered the same names used in the earlier example of an array, you would have a group of compound variables as follows:

```
employee.1 = 'Adams, Joe'
employee.2 = 'Crandall, Amy'
employee.3 = 'Devon, David'
employee.4 = 'Garrison, Donna'
```

When the names are in the group of compound variables, you can easily access a name by its number, or by a variable that represents its number.

```
name = 3
SAY employee.name              /* Displays 'Devon, David'   */
SAY employee.2                 /* Displays 'Crandall, Amy'  */
```

Stems

A **stem** is a symbol that contains just one period, which is the last character. It cannot start with a digit or a period.

Examples:

JOHN.
B.
ARRAY.
EMPLOYEE.

By default, the value of a stem is the string consisting of the characters of its symbol (that is, translated to uppercase). If the symbol has been assigned a value, it names a variable and its value is the value of that variable.

Example:

```
/* rexx */
a = stem.1
b = stem.any
stem.3 = "assigned"
stem.4 = assigned
say a stem.2 b stem.3 stem.4
/* displays      STEM.1 STEM.2 STEM.ANY assigned ASSIGNED  */
```

When a stem is used as the target of an assignment then all possible compound variables whose names begin with that stem receive the new value regardless of whether they previously had a value or not.

In the following example, all possible compound variables whose names begin **abc.** are assigned a value of "Anyone" by the first statement, and then some of the compound variables are reassigned.

```
abc. = "Anyone"
d = 5
abc.d = "Hello!"
abc.6 = "home?"
SAY abc.d abc.f abc.6 abc.7 abc.8 abc.anything
/* displays       Hello! Anyone home? Anyone Anyone Anyone  */
```

You can use stems with the EXECIO command when reading from and writing to a data set. This will be covered in a later section.

Stems can also be used with the OUTTRAP TSO/E external function when trapping command output. See the TSO/E REXX User's Guide for more information. There is also an example at the end of this document.

IF/THEN/ELSE

At least one instruction should follow the THEN and ELSE clauses. When either clause has no instructions, it is good programming practice to include NOP (no operation) next to the clause.

```
IF expression THEN
     Instruction
ELSE NOP
```

If you have more than one instruction for a condition, begin the set of instructions with a DO and end them with an END.

```
IF weather = rainy THEN
     SAY "Find a good book."
ELSE
     DO
       SAY "Would you like to play tennis or golf?"
       PULL answer
     END
```

Without the enclosing DO and END, the language processor assumes only one instruction for the ELSE clause.

When nesting IF/THEN/ELSE instructions within other IF/THEN/ELSE instructions it is important to match each IF with an ELSE and each DO with an END otherwise the results can be unpredictable.

Example:

```
weather = "fine"
tenniscourt = "occupied"

IF weather = "fine" THEN
     DO
       SAY "What a lovely day!"
       IF tenniscourt = "free" THEN
            SAY "Shall we play tennis?"
       ELSE NOP
     END
ELSE
     SAY "Shall we take our raincoats?"
```

And the result of executing this is:

```
What a lovely day!
```

Without the DO/END/ELSE NOP the code looks like this:

```
weather = "fine"
tenniscourt = "occupied"

IF weather = "fine" THEN
     SAY "What a lovely day!"
     IF tenniscourt = "free" THEN
       SAY "Shall we play tennis?"
ELSE
     SAY "Shall we take our raincoats?"
```

And the result of executing this is:

```
What a lovely day!
Shall we take our raincoats?
```

Moral of the story is to be sure to use DO and END to group instructions in an IF…THEN…ELSE set and when nesting IF…THEN…ELSE include an ELSE NOP where appropriate.

Looping

Looping through your code is controlled by the use of the DO and END instruction. We saw a simple DO…END in our earlier discussion of the IF…THEN…ELSE sequence. This DO…END sequence caused the instructions contained within to be executed only one time. Using DO with a repetitor expression we can cause the set of instructions to be executed any number of times. Examples are:

```
DO 5
  SAY "Hello!"
END
```

OR

```
number = 5
DO number
  SAY "Hello!"
END
```

This will display Hello! 5 times. Note: ***number*** must evaluate to a whole number otherwise you will receive an error.

The most common repetitive DO loop is of the form:

```
DO X = Start TO Fini BY Incr
```

Start, Fini*,* and ***Incr*** must evaluate to numbers or an error will occur. If ***BY Incr*** is not specified it defaults to +1.

Two other common DO loop forms are:

```
DO WHILE expression
```

and

```
DO UNTIL expression
```

In a DO WHILE loop the ***expression*** is evaluated <u>before</u> the group of instructions is executed. In a DO UNTIL the ***expression*** is evaluated <u>after</u> the group of instructions is executed. In other words, a DO UNTIL will always execute its group of instructions at least one time whereas a DO WHILE may not execute at all.

The last DO loop to be discussed is:

```
DO FOREVER
```

This is a continuous loop, as you would surmise. To exit this loop construction you must use either EXIT, which will end the routine all together, or the LEAVE instruction. LEAVE may be used in all DO loops to end the loop prematurely. The LEAVE instruction affects the innermost loop in a set of nested loops unless the control variable for a loop is specified as part of the instruction.

Example 1:

```
      count = 0
      DO FOREVER
        SAY "HERE WE GO LOOP-DE-LOOP"
        count = count + 1
        IF count > 5 THEN
             LEAVE
      END
```

The output from this execution looks like this:

```
HERE WE GO LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
```

Example 2:

```
      DO I = 1 TO 5
         SAY "HERE WE GO LOOP-DE-LOOP"
         DO J = 1 TO 2
              SAY "LOOP-DE-LOOP"
              IF I > 3 THEN
                 LEAVE J
              ELSE NOP
         END
      END
```

The execution output looks like:

```
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
```

Example 3:

```
      DO I = 1 TO 5
         SAY "HERE WE GO LOOP-DE-LOOP"
         DO J = 1 TO 2
              SAY "LOOP-DE-LOOP"
              IF I > 3 THEN
                LEAVE I
              ELSE NOP
         END
      END
```

The execution output looks like:

```
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
LOOP-DE-LOOP
HERE WE GO LOOP-DE-LOOP
LOOP-DE-LOOP
```

Parsing

Parsing is the act of separating data into one or more variables. There are several forms of the PARSE instruction but the most commonly used are:

PARSE PULL
PARSE ARG
PARSE VAR

Each of these may also include UPPER to force all character information to uppercase before putting it into the variables; e.g., PARSE UPPER VAR. The instructions PULL and ARG may substitute for PARSE UPPER PULL and PARSE UPPER ARG respectively. ARG is used to pass arguments to another exec, a subroutine or a function.

Parsing separates the data by comparing the data to a template or pattern of variable names. Separators in a template can be a blank (most common), string, variable, or number that represents a column position.

The simplest, and most common, template is a group of variable names separated by blanks. Each variable name gets one word of data in sequence except for the last variable which gets the remainder of the data.

Example:

```
/* REXX */
  ptst = "A B C D E"
  PARSE VAR ptst v1 v2 v3
  SAY "V1="  v1       /* v1 CONTAINS WORD 1 (A)                    */
  SAY "V2="  v2       /* v2 CONTAINS WORD 2 (B)                    */
  SAY "V3="  v3       /* v3 CONTAINS ALL REMAINING WORDS           */
  EXIT
```

When executed the result is:

```
V1= A
V2= B
V3= C D E
```

Another template can use a string, sometimes called a delimiter, to separate data as long as the data contains the string as well.

```
/* REXX  - PARSE with a delimiter */
  ptst = "AA BB:CC DD EE"
  PARSE VAR ptst v1":" v2
  SAY "V1="  v1
  SAY "V2="  v2
  SAY "V3="  v3
  EXIT
```

When executed the result is:

```
V1= AA BB
V2= CC DD EE
V3= V3
```

A string or delimiter may also be a variable in the event you don't know what the separator should be in advance. In this case the delimiter name must be enclosed in parentheses.

```
/* REXX  - PARSE with a delimiter as a variable */
  ptst = "AA BB:CC:DD EE"
  delim = ":"

  PARSE VAR ptst v1 (delim) v2 (delim) v3
  SAY "V1="  v1
  SAY "V2="  v2
  SAY "V3="  v3
  EXIT
```

When executed the result is:

```
V1= AA BB
V2= CC
V3= DD EE
```

Numbers can be used in the template to indicate at which column to separate data. An unsigned integer indicates an absolute column position and a signed integer indicates a relative column position.

Absolute column position - the unsigned integer in the template separates the data according to absolute position. The first segment begins at column 1 and continues up to but does not include the column number specified. Subsequent segments begin at the column number specified.

```
/* REXX  - PARSE using absolute column positioning */
  ptst = "AAAABBCCCCCDDDDDDEEE"

  PARSE VAR ptst v1 5 v2 7 v3
  SAY "V1=" v1        /* v1 CONTAINS ONLY COLUMNS 1-4               */
  SAY "V2=" v2        /* v2 CONTAINS ONLY COLUMNS 5-6               */
  SAY "V3=" v3        /* v3 CONTAINS ALL FROM COLUMN 7 TO THE END   */
  EXIT
```

When executed the result is:

```
V1= AAAA
V2= BB
V3= CCCCCDDDDDDEEE
```

Relative column positions – a signed integer in the template separates data according to the relative column position. The signed integer specifies the length of the data assigned to the variable preceding it in the template. Negative numbers are allowed, just be sure of what you are trying to do.

```
/* REXX  - PARSE using relative column positioning */
  ptst = "AAAABBCCCCCDDDDDDEEE"

  PARSE VAR ptst v1 +4 v2 +2 v3
  SAY "V1=" v1        /* v1 CONTAINS ONLY COLUMNS 1-4               */
  SAY "V2=" v2        /* v2 CONTAINS ONLY COLUMNS 5-6               */
  SAY "V3=" v3        /* v3 CONTAINS ALL FROM COLUMN 7 TO THE END   */
  EXIT
```

When executed the result is:

```
V1= AAAA
V2= BB
V3= CCCCCDDDDDDEEE
```

Parsing Templates: The Period as a Placeholder

A period in a template is a placeholder. It is used instead of a variable name, but it receives no data. It is useful:

- As a "dummy variable" in a list of variables
- Or to "collect" unwanted information at the end of a string

Examples:

```
  /* REXX - PARSE using a period in the parse template */
  ptst = "AA BB CC DD EE"
  SAY "PERIOD AS PLACEHOLDER"
  PARSE VAR ptst v1 v2 . . v3
  SAY "V1="  v1         /* v1 CONTAINS WORD 1 (AA) */
  SAY "V2="  v2         /* v2 CONTAINS WORD 2 (BB) */
  SAY "V3="  v3         /* v3 CONTAINS WORD 5 (EE) */
  DROP v1
  DROP v2
  DROP v3
  SAY
  SAY "PERIOD AS A COLLECTOR OF UNWANTED INFORMATION"
  PARSE VAR ptst v1 . v2 .
  SAY "V1="  v1         /* v1 CONTAINS WORD 1 (AA) */
  SAY "V2="  v2         /* v2 CONTAINS WORD 2 (CC) */
  EXIT
```

When executed the result is:

```
PERIOD AS PLACEHOLDER
V1= AA
V2= BB
V3= EE

PERIOD AS A COLLECTOR OF UNWANTED INFORMATION
V1= AA
V2= CC
```

```
/* REXX - parse example with periods as placeholders */
parse source . . . . . . . ENV . .
if ENV = 'OMVS' then do       /* are we running under UNIX ?  */
   (instructions)
end
if ENV = 'TSO/E' then do      /* are we running under TSO/E?  */
   (instructions)
end
if ENV = 'ISPF' then do       /* are we running under ISPF?   */
   (instructions)
end
if ENV = 'MVS'  then do       /* are we running under MVS?    */
   (instructions)
end
```

Subroutines and Functions

Subroutines and functions consist of instructions that can receive data, process data, and return a value. Routines are either:

- **Built-in** (part of the language definition)
- **Internal** (within the current exec)
- **External** (program written in REXX or other language)

The search order for functions is: internal routines take precedence, then built-in functions, and finally external functions.

Calling a subroutine:

To call a subroutine, use the CALL instruction. The subroutine call is an entire instruction:

```
CALL  subroutine_name  argument1, argument2, argument3,....
```

A subroutine does not have to return a value, but when it does, it sends back the value with the RETURN instruction.

```
RETURN value
```

The calling exec receives the value in the REXX special variable named RESULT.

```
SAY 'The answer is' RESULT
```

Calling a function:

A function call is part of an instruction, for example, and assignment instruction:

```
var1  = function(argument1, argument2, argument3,.....)
```

A function must return a value. The value replaces the function call, so that in the previous example, var1 = value.

```
SAY 'The answer is'  var1
```

Built-in REXX Functions (1 of 2)

ABBREV (Abbreviation) - determines if a substring begins a string
ABS (Absolute Value) - returns the absolute value of a number
ADDRESS - returns the name of the current command environment
ARG (Argument) - returns an argument string or information about argument strings
BITAND (Bit by Bit AND) - returns the result of logically ANDing string1 and string2
BITOR (Bit by Bit OR) - returns the result of logically inclusive ORing string1 and string2
BITXOR (Bit by Bit Exclusive OR) - returns the result of logically excl. ORing string1 and string2
B2X (Binary to Hexadecimal) - returns the hexadecimal equivalent of a binary string
CENTER/CENTRE - returns a string with its string argument centered in it
COMPARE - compares to strings for equality or returns first column they differ
CONDITION - returns information about a trapped condition
COPIES - returns specified number of copies of its string argument
C2D (Character to Decimal) - returns the decimal value of a string's binary representation
C2X (Character to Hex) - returns, in character format, a hexadecimal representation of a string
DATATYPE - returns the data type of its string argument
DATE - returns the system date in the format specified
DELSTR (Delete String) - returns a substring from a string
DELWORD (Delete Word) - removes a blank-delimited string (word) from a string
DIGITS - returns the current setting of NUMERIC DIGITS
D2C (Decimal to Character) - returns, in character format, a decimal number converted to binary
D2X (Decimal to Hexadecimal) - returns the hexadecimal equivalent of a decimal number
ERRORTEXT - returns the REXX error message associated with an error number
FORM - returns the current setting of NUMERIC FORM
FORMAT - returns a rounded and formatted number
FUZZ - returns the current setting of NUMERIC FUZZ
INSERT - inserts one string into another
LASTPOS (Last Position)- returns the position of the last occurrence of a string in another string
LEFT - returns a string of a specified length containing the leftmost portion of a string
LENGTH - returns the length of a string
MAX (Maximum) - returns the largest number from a list of numbers
MIN (Minimum) - returns the smallest number from a list of numbers
OVERLAY - writes a string onto another string starting at a specified position
POS (Position) - returns the position of one string within another string
QUEUED - returns the number of lines in the external data queue (TSO/E data stack)
RANDOM - returns a random whole number
REVERSE - returns the reverse of a string
RIGHT - returns a string of a specified length containing the rightmost portion of a string
SIGN - returns a number that indicates a number's sign (1/positive, -1/negative, or 0/zero).
SOURCELINE - returns the source of a line number within the program
SPACE - formats blank-delimited words in a string
STRIP - removes leading and / or trailing characters from a string (usually blanks)
SUBSTR (Substring) - returns a substring of specified length and position from within a string
SUBWORD - returns a substring of words from within a string of words
SYMBOL - returns information about a specified symbol

Built-in REXX Functions (2 of 2)

TIME - returns the system time in the format specified
TRACE - returns trace settings currently in effect and, optionally, alters the setting
TRANSLATE - returns a string with translated characters from an input string
TRUNC (Truncate) - returns the integer part of number and a specified number of decimal places
VALUE - returns the value of a specified symbol and, optionally, assigns it a new value
VERIFY - compares two strings to see if they contain the same characters
WORD - returns the specified blank-delimited string (word) within a string
WORDINDEX - returns the position of a specified blank-delimited string (word)
WORDLENGTH - returns the length of a specified blank-delimited string (word) within a string
WORDPOS (Word Position) - returns the word number of a substring in its string
WORDS - returns the number of blank-delimited strings (words) within a string
XRANGE (Hexadecimal Range) - returns a string that spans all hex values in a specified range
X2B (Hexadecimal to Binary) - returns the binary string equivalent of a hexadecimal string
X2C (Hexadecimal to Character) - returns the character equivalent of a hexadecimal string
X2D (Hexadecimal to Decimal) - returns the decimal equivalent of a hexadecimal string

There are six other non-SAA built-in functions that TSO/E provides: EXTERNALS, FIND, INDEX, JUSTIFY, LINESIZE, and USERID. If you plan to write REXX programs that run on other SAA environments, note that these functions are not available to all the environments.

EXTERNALS - returns the number of elements in the terminal input buffer (always 0 for TSO/E)
FIND - returns the word number of a substring in its string (for SAA use **WORDPOS**)
INDEX - returns the position of one string within another string (for SAA use **POS**)
JUSTIFY - returns a formatted string from blank-delimited words to justify both margins
LINESIZE - returns the current terminal line width minus 1
USERID - returns the TSO/E user ID

TSO/E External Functions

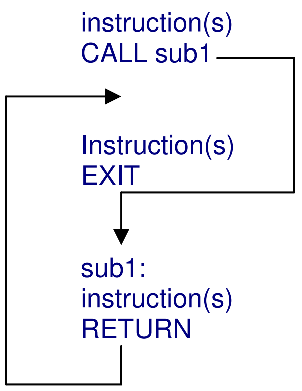
GETMSG - used to retrieve messages from a TSO/E CONSOLE session into variables
LISTDSI - returns information about a data set into variables
MSG - returns / sets the current value of displaying TSO/E messages
MVSVAR - returns information about a specified MVS system variable
OUTTRAP - used to trap TSO/E command output and store that output into variables
PROMPT - returns / sets the current value of TSO/E command prompting within an exec
SETLANG - returns / sets a code that indicates the language of displayed REXX messages
STORAGE - returns data of a specified length from a specified address in storage
SYSCPUS - returns information in a stem variable about online CPUs
SYSDSN - returns information indicating whether a data set exists and is available for use
SYSVAR - returns information about a specified system variable

You can use the MVSVAR, SETLANG, STORAGE and SYSCPUS external functions in REXX execs that run in any address space, TSO/E and non-TSO/E. You can use the other external functions only in REXX execs that run in the TSO/E address space.

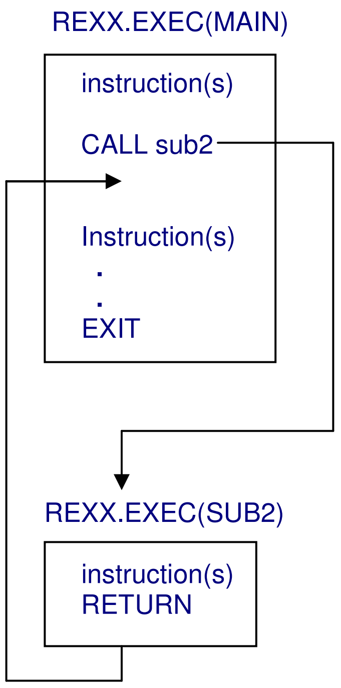
Writing a Subroutine

The instruction that invokes the subroutine is the CALL instruction. The CALL instruction may be used several times in an exec to invoke the same subroutine.

When the subroutine ends, it can return control to the instruction that directly follows the subroutine call. The instruction that returns control is the RETURN instruction.



Subroutines may be internal and designated by a label as in the previous example, or they may be external and designated by the data set member name that contains the subroutine.
The following illustrates an external subroutine.



User written external subroutines must reside in an execution library that is part of the implicit search order, for example SYSEXEC or SYSPROC, or be in the same PDS.

Writing a Function

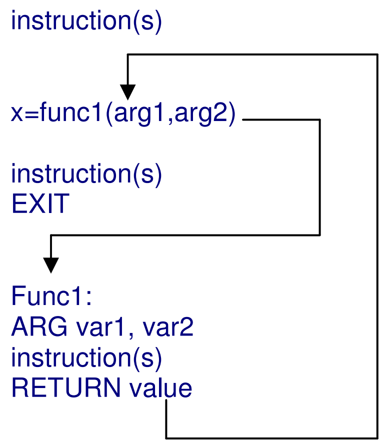
An exec invokes a user-written function the same way it invokes a built-in function -- by the function name immediately followed by parentheses with no blanks in between. The parentheses can contain up to 20 arguments or no arguments at all.

```
function(argument1, argument2,...)
      or
function()
```

A function requires a returned value because the function call generally appears in an expression.

```
x = function(argument1, argument2,...)
```

When the function ends, it uses the RETURN instruction to send back a value.



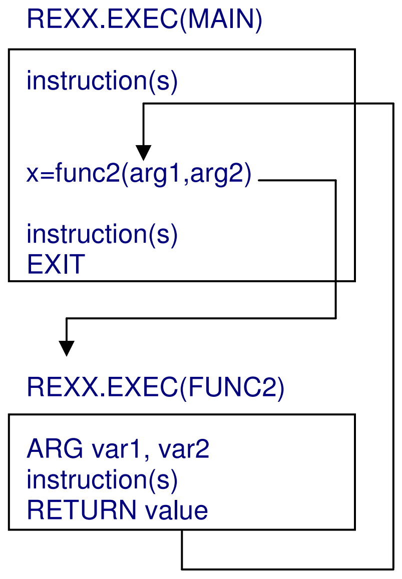
```
instruction(s)

x=func1(arg1,arg2)

instruction(s)
EXIT

Func1:
ARG var1, var2
instruction(s)
RETURN value
```

The previous example illustrated an internal function. The following illustrates an external function.



```
REXX.EXEC(MAIN)

instruction(s)

x=func2(arg1,arg2)

instruction(s)
EXIT

REXX.EXEC(FUNC2)

ARG var1, var2
instruction(s)
RETURN value
```

Here is an example of an internal function that adds three numbers. Note the commas (,) between the function's arguments which are the variables "number1", "number2", and "number3".

```
/* REXX  - internal "ADD3" function example */
v1 = add3(10,20,30)
v2 = add3(5,100,25)
SAY "V1="  v1
SAY "V2="  v2
EXIT  /* be sure to include for internal function calls */

Add3:
ARG number1, number2, number3
answer = number1 + number2 + number3
RETURN answer
```

When executed the result is:

```
V1= 60
V2= 130
```

NOTE: Because internal functions and subroutines generally appear after the main part of the exec, when you have an internal function or subroutine, it is important to end the main part of the exec with the EXIT instruction.

User written external functions must reside in an execution library that is part of the implicit search order, for example SYSEXEC or SYSPROC, or be in the same PDS. The following illustrates ADD3 written as an external function.

REXX.EXEC(MYPGM)

```
/* REXX  - external function call example */
v1 = add3(10,20,30)
v2 = add3(5,100,25)
SAY "V1="  v1
SAY "V2="  v2
EXIT
```

REXX.EXEC(ADD3)

```
/* REXX - ADD3 function to add 3 numbers */
ARG number1, number2, number3
answer = number1 + number2 + number3
RETURN answer
```

The TSO/E Data Stack

REXX in TSO/E uses an expandable data structure called a *data stack* to store information. Data stacks are used to temporarily hold data items until they are needed.

Data is placed into the data stack using either the PUSH or QUEUE instructions. Data is retrieved from the data stack using the PULL or PARSE PULL instruction. The difference between the PUSH and QUEUE commands is apparent when the data is retrieved from the data stack.

PUSH is used to place data on top of the stack. This equates to last in first out (LIFO) when the data is retrieved. QUEUE places data last in the stack, equating to first in first out (FIFO).

To determine the number of elements in the stack you use the built in function QUEUED(). This will return a number representing the number of elements in the stack.

When an exec issues a PULL instruction, and when it issues an interactive TSO/E command, the data stack is searched first for information and if that is empty, information is retrieved from the terminal.

The data stack has some unique characteristics, such as:

- It can contain a virtually unlimited number of data items of virtually unlimited size.
- It can contain commands to be issued after the exec ends.
- It can pass information between REXX execs and other types of programs in a TSO/E or non-TSO/E address space.

Because of the data stack's unique characteristics, you can use the data stack specifically to:

- Store a large number of data items for a single exec's use.
- Pass a large number of arguments or an unknown number of arguments between a routine (subroutine or function) and the main exec.
- Pass responses to an interactive command that can run after the exec ends. Note: Data left in the stack when the exec ends will be treated as TSO commands and will try to execute as such.
- Store data items from an input data set, which were read by the EXECIO command.

The following examples illustrate the difference when QUEUE and PUSH are used to place data in the data stack. PARSE PULL is used to retrieve information from the stack.

QUEUE example:

```
/* REXX */
  DO A = 1 TO 3
    QUEUE "A=" A            /* PUT DATA IN THE STACK (FIFO) */
  END
  SAY "NUMBER OF ELEMENTS IN THE STACK IS:" QUEUED()
  IF QUEUED() > 0 THEN
    DO QUEUED()    /* DISPLAY ALL THE ITEMS IN THE STACK */
      PARSE PULL V1
      SAY V1
    END
  EXIT
```

When executed the results are:

```
NUMBER OF ELEMENTS IN THE STACK IS: 3
A= 1
A= 2
A= 3
```

PUSH example:

```
/* REXX */
  DO A = 1 TO 3
    PUSH "A=" A             /* PUT DATA IN THE STACK (LIFO) */
  END
  SAY "NUMBER OF ELEMENTS IN THE STACK IS: " QUEUED()
  IF QUEUED() > 0 THEN
    DO QUEUED()    /* DISPLAY ALL THE ITEMS IN THE STACK */
      PARSE PULL V1
      SAY V1
    END
  EXIT
```

When executed the results are:

```
NUMBER OF ELEMENTS IN THE STACK IS:  3
A= 3
A= 2
A= 1
```

Other data stack functions that will not be covered here include:

MAKEBUF - Create a buffer in the data stack
DROPBUF- Drop a buffer in the data stack
QBUF - Find the number of buffers in a data stack
QELEM - Find the number of elements in a buffer
NEWSTACK - Create a new data stack
DELSTACK - Delete the most recently created data stack
QSTACK - Find the number of stacks that exist

Debugging REXX Execs

In order to aid in debugging an exec with an error condition or a logic problem, REXX provides a powerful TRACE instruction. You can use the TRACE instruction to display how the language processor evaluates each operation of an expression as it reads it (TRACE I) , or to display the final result of an expression (TRACE R). These two types of tracing are useful for debugging execs, however, there are also many other TRACE options (?,A,C,E,F,I,L,N,O,R).

Tracing Operations: To trace operations within an expression, use the TRACE I (TRACE Intermediates) form of the TRACE instruction. All expressions that follow the instruction are then broken down by operation and analyzed as:

`>V>`	- **V**ariable value - The data traced is the contents of a variable.
`>L>`	- **L**iteral value - The data traced is a literal (string, uninitialized variable, or constant).
`>O>`	- **O**peration result - The data traced is the result of an operation on two terms.
`>F>`	- **F**unction result - The data traced is the result of a function call.
`>C>`	- **C**ompound variable - The data traced is the name of a compound variable
`>P>`	- **P**refix operation - The data traced is the result of a prefix operation

Other trace output is as follows:

`*-*`	- a source line from the program
`+++`	- a trace message from REXX
`>>>`	- an expression result (TRACE R)
`>.>`	- a value assigned to a place holder

Example using TRACE I:

```
EDIT   USERID.REXX.EXEC(TRACEI)                        Columns 00001 00072
Command ===>                                              Scroll ===> CSR
***************************** Top of Data ****************************
000001 /* rexx */
000002 x = 9
000003 y = 2
000004 TRACE I
000005 IF x + 1 > 5 * y THEN
000006     SAY 'x is big enough.'
000007 ELSE NOP    /* No operation on the ELSE path */
**************************** Bottom of Data **************************
```

When executed the result is:

```
5 *-* IF x + 1 > 5 * y
  >V>   "9"
  >L>   "1"
  >O>   "10"
  >L>   "5"
  >V>   "2"
  >O>   "10"
  >O>   "0"
7 *-* ELSE
  *-*  NOP    /* No operation on the ELSE path */
```

First you see the line number (5 *-*) followed by the expression. Then the expression is broken down by operation as follows:

```
  >V>   "9"    (value of variable x)
  >L>   "1"    (value of literal 1)
  >O>   "10"   (result of operation x + 1)
  >L>   "5"    (value of literal 5)
  >V>   "2"    (value of variable y)
  >O>   "10"   (result of operation 5 * y)
  >O>   "0"    (result of final operation 10 > 10 is false)
```

Tracing Results: To trace only the final result of an expression, use the TRACE R (TRACE Results) form of the TRACE instruction. All expressions that follow the instruction are analyzed and the results are displayed as:

```
 >>>   Final result of an expression
```

If you changed the TRACE instruction operand in the previous example from an I to an R, you would see the following results.

```
5 *-* IF x + 1 > 5 * y
  >>>   "0"
7 *-* ELSE
  *-*  NOP    /* No operation on the ELSE path */
```

With interactive debugging, you also have the ability to single or multi-step though instructions. The language processor will pause between instructions to await debug input. While paused you can enter any commands or REXX statements or re-execute the last instruction.

To enable interactive debug prefix the TRACE action with a **?** (for example TRACE ?R). Using the previous example with TRACE ?R you would see the following:

```
5 *-* IF x + 1 > 5 * y
       >>>   "0"
+++ Interactive trace.  TRACE OFF to end debug, ENTER to continue. +++
```

After hitting ENTER, you would step though to the next result (if any) and the language processor would pause awaiting input.

```
7 *-* ELSE
  *-*  NOP    /* No operation on the ELSE path */
```

While paused you can continue tracing by entering a null line, type one or more additional instructions to be processed before the next instruction is traced, enter an equal sign (=) to re-execute the last instruction, or change trace options. For example, TRACE N (TRACE Normal) would turn back the default tracing options (trace failures only), TRACE O (TRACE Off) would turn off all tracing, and TRACE ? would turn off interactive tracing (but leave the current tracing options in effect).

Environments and Issuing Commands

Issuing commands to the surrounding environment is an integral part of REXX and one of the things that makes REXX such a powerful tool.

Environments

The system under which REXX programs run includes at least one host command environment for processing commands. An environment is selected by default on entry to a REXX program. In TSO/E REXX, the environment for processing host commands is known as the host command environment. You can change the environment by using the ADDRESS instruction. You can find out the name of the current environment by using the ADDRESS built-in function.

TSO/E provides several host command environments for a TSO/E address space (TSO/E and ISPF) and for non-TSO/E address spaces. For example, in REXX processing, a host command can be:

- A TSO/E command processor, such as ALLOCATE, FREE, or EXEC
- A TSO/E REXX command, such as NEWSTACK or QBUF
- A program that you link to or attach
- An MVS system or subsystem command that you invoke during an extended MCS console session
- An ISPF command or service

When you invoke a REXX exec in the TSO/E address space, the default initial host command environment is TSO (**ADDRESS TSO**). When you run a REXX exec in a non-TSO/E address space, the default initial host command environment is MVS (**ADDRESS MVS**).

The CONSOLE host command environment (**ADDRESS CONSOLE**) is available only to REXX execs that run in the TSO/E address space. Use the CONSOLE environment to invoke MVS system and subsystem commands during an extended MCS console session. To use the CONSOLE environment, you must have CONSOLE command authority.

The ISPEXEC and ISREDIT host command environments are available only to REXX execs that run in ISPF (**ADDRESS ISPEXEC** and **ADDRESS ISREDIT**). Use these environments to invoke ISPF commands and services, and ISPF edit macros. To use ISREDIT, you must be in an edit (or view) session.

The LINK, LINKMVS, and LINKPGM host command environments are available to any address space and let you link to unauthorized programs on the same task level (**ADDRESS LINK**, **ADDRESS LINKMVS**, and **ADDRESS LINKPGM**). The ATTACH, ATTCHMVS, and ATTCHPGM host command environments are also available to any address space and let you attach unauthorized programs on a different task level (**ADDRESS ATTACH**, **ADDRESS ATTCHMVS**, and **ADDRESS ATTCHPGM**).

To change the host command environment, use the ADDRESS instruction followed by the name of an environment. The ADDRESS instruction has two forms: one affects all commands issued after the instruction, and one affects only a single command.

- **All commands**

  When an ADDRESS instruction includes only the name of the host command environment, all commands issued afterward within that exec are processed as that environment's commands.

```
ADDRESS ISPEXEC     /* Change environment to ISPF */
"EDIT DATASET("dsname")"
        .
  instruction(s)
        .
"SELECT PANEL(TECH001)"
        .
  instruction(s)
        .
        .
ADDRESS TSO         /* Change environment back to TSO  */
"FREE FILE(WORK)"
```

- **Single Command**

  When an ADDRESS instruction includes both the name of the host command environment and a command, only that command is affected. After the command is issued, the former host command environment becomes active again.

```
/* Issue one command from the ISPF environment        */
ADDRESS ISPEXEC "EDIT DATASET("dsname")"
/* Return to the default TSO host command environment */
"ALLOC DA("dsname") F(SYSEXEC) SHR REUSE"
```

ADDRESS Example 1:

```
/* rexx */
SAY 'Do you know your PF keys?'
PULL answer
IF answer = 'NO' | answer = 'N' THEN
  ADDRESS ispexec "SELECT PGM(ISPOPT) PARM(ISPOPT3)"
ELSE
  SAY 'O.K. Never mind.'
```

ADDRESS Example 2:

```
/* REXX */
text = 'PROGRAM' pgname 'ABENDED. PLEASE ACKNOWLEDGE BY REPLYING!'
ltext = d2c(length(text),2)
wtor = ltext || text
ADDRESS LINKPGM "OPSWTOR wtor"
Exit 0
```

Commands

To send a command to the currently addressed host command environment, use a clause of the form:

```
  expression;
```

The expression is evaluated and submitted to the host command environment. The environment then processes the command and eventually returns control to the language processor, after setting a return code. The language processor places this return code in the REXX special variable **RC**. The return code can then be tested for conditional processing:

```
  "ALLOC FILE(INPUT) DA(PROGA.INPUT) SHR"
  IF RC \= 0 THEN DO
     SAY 'CAN''T ALLOCATE INPUT FILE'
     EXIT 16  /* exit and set RC=16 */
  END
  ELSE DO
          .
    instructions(s)
          .
  END
```

To differentiate commands from other types of instructions, enclose the command within single or double quotation marks. Any part of the expression not to be evaluated should be enclosed in quotation marks.

Here is an example of submitting a command to the TSO/E host command environment

```
  mydata = "PROGA.OUTFILE"
  "FREE DATASET("mydata")"
```

This would result in the string **FREE DATASET(PROGA.OUTFILE)** being submitted to TSO/E.

Many TSO/E commands, for example EXEC and ALLOCATE, use single quotation marks within the command. For this reason, it is recommended that, as a matter of course, you enclose TSO/E commands with double quotation marks.

```
"ALLOC DA('USERID.MYREXX.EXEC') F(SYSEXEC) SHR REUSE"

"EXEC 'USERID.MYREXX.EXEC(ADD3)' '25 78 33' EXEC"
```

When a variable represents a fully-qualified data set name, the name must be enclosed in two sets of quotation marks to ensure that one set of quotation marks remains as part of the value.

```
  name = "'SYS3.LINKLIB'"
  "LISTDS" name "STATUS"
```

Another way to ensure that quotation marks appear around a fully-qualified data set name when it appears as a variable is to include them as follows:

```
  name = SYS3.LINKLIB
  "LISTDS '"name"' STATUS"
```

Processing Data Sets With EXECIO

The EXECIO command can be used to:

- Read information from a data set
- Write information to a data set
- Open a data set without reading or writing any records
- Empty a data set
- Copy information from one data set to another
- Copy information to and from a list of compound variables
- Add information to the end of a sequential data set
- Update information in a data set one line at a time.

Before you can use the EXECIO command to read from or write to a data set, the data set must meet the following requirements. An I/O data set must be:

- Either sequential or a single member of a PDS.
- Previously allocated with the appropriate attributes for its specific purpose.

Allocation can be made by using the TSO ALLOCATE command for REXX execs that execute in a TSO/E address space, or in the case of batch REXX execution, the allocations can be made with JCL.

If you use EXECIO to read information from a data set and to the data stack, the information can be stored in FIFO or LIFO order on the data stack. FIFO is the default. If you use EXECIO to read information from a data set and to a list of variables, the first data set line is stored in variable1, the second data set line is stored in variable2, and so on. Data read into a list of variables can be accessed randomly. After the information is in the data stack or in a list of variables, the exec can test it, copy it to another data set, or update it before returning it to the original data set.

Reading Information from a Data Set

To read information from a data set to the data stack or to a list of variables, use EXECIO with either the DISKR or DISKRU operand. A typical EXECIO command to read all lines from the data set allocated to the ddname MYINDD, might appear as:

```
    "EXECIO * DISKR  myindd (FINIS"
```

Specify the number of lines to read: To open a data set without reading any records, put a zero immediately following the EXECIO command and specify the OPEN operand.

```
   "EXECIO 0  DISKR mydd (OPEN"
```

To read a specific number of lines, put the number immediately following the EXECIO command.

```
    "EXECIO 25 ..."
```

To read the entire data set, put an asterisk immediately following the EXECIO command.

```
    "EXECIO  * ..."
```

When all the information is on the data stack, either queue a null line to indicate the end of the information, or if there are null lines throughout the data, assign the built-in QUEUED() function to a variable to indicate the number of items on the stack.

Depending on the purpose you have for the input data set, use either the DISKR or DISKRU operand of EXECIO to read the data set.

DISKR - Reading Only - To initiate I/O from a data set that you want to read only, use the DISKR operand with the FINIS option. The FINIS option closes the data set after the information is read. Closing the data set allows other execs to access the data set and the ddname.

```
    "EXECIO  *  DISKR ... (FINIS"
```

Note: Do not use the FINIS option if you want the next EXECIO statement in your exec to continue reading at the line immediately following the last line read.

DISKRU - Reading and Updating - To initiate I/O to a data set that you want to both read and update, use the DISKRU operand without the FINIS option. You can update only the last line that was read therefore, you usually read and update a data set one line at a time, or go immediately to the single line that needs updating. The data set remains open while you update the line and return the line with a corresponding EXECIO DISKW command.

```
    "EXECIO  1  DISKRU ..."
```

In order to access a data set for I/O, it must first be allocated to a ddname. The ddname need not exist previously. You can allocate before the exec runs, or you can allocate from within the exec with the ALLOCATE command.

```
    "ALLOC DA(input.data) F(mydd) SHR REUSE"
    "EXECIO  *  DISKR  mydd  (FINIS"
```

You can specify a starting line number for reading a data set other than at the beginning. For example, to read all lines to the data stack starting at line 100, add the following line number operand.

```
  "EXECIO  *  DISKR  myindd  100  (FINIS"
```

To read just 5 lines to the data stack starting at line 100, write the following:

```
  "EXECIO  5  DISKR  myindd  100  (FINIS"
```

To open a data set at line 100 without reading lines 1 through 99 to the data stack, write the following:

```
  "EXECIO  0  DISKR  myindd  100  (OPEN"
```

To read the information to either a list of compound variables that can be indexed, or a list of variables appended with numbers use the STEM option. Specifying STEM with a variable name ensures that a list of variables (not the data stack) receives the information.

```
"EXECIO  *  DISKR  myindd  (STEM newvar."
```

In this example, the list of compound variables has the stem **newvar.** and lines of information or records from the data set are placed in variables **newvar.1**, **newvar.2**, **newvar.3**, and so forth. The number of items in the list of compound variables is placed in the special variable **newvar.0**. This makes it extremely easy to create loops that process input data. Example:

```
"ALLOC DA(old.data) F(indd) SHR REUSE"
"EXECIO * DISKR indd (STEM newvar."
DO i = 1 to newvar.0
   SAY newvar.i
END
```

To avoid confusion as to whether a residual stem variable value is meaningful, you should clear the entire stem variable prior to entering the EXECIO command. To clear all stem variables, you can either:

- Use the DROP instruction which sets all stem variables to their uninitialized state:

```
DROP newvar.
```

- Set all stem variables to nulls:

```
newvar. = ''
```

Writing Information to a Data Set

Use EXECIO with DISKW operand to write information to a data set from the data stack or from a list of variables. A typical EXECIO command to write all lines to the data set allocated to the ddname, MYOUTDD, might appear as:

```
"EXECIO * DISKW myoutdd (FINIS"
```

Specify the number of lines to write: To open a data set without writing records to it, put a zero immediately following the EXECIO command and specify the OPEN operand.

```
"EXECIO 0  DISKW  myoutdd ... (OPEN"
```

To empty a data set, issue this command to open the data set and position the file position pointer before the first record. You then issue **EXECIO 0 DISKW myoutdd ... (FINIS** to write an end-of-file mark and close the data set. This deletes all records in data set MYOUTDD. You can also empty a data set by issuing EXECIO with both the **OPEN** and **FINIS** operands like this:

```
"EXECIO 0 DISKW myoutdd ... (OPEN FINIS"
```

To write a specific number of lines, put the number immediately following the EXECIO command.

```
   "EXECIO 25  DISKW  ..."
```

To write the entire data stack or until a null line is found, put an asterisk immediately following the EXECIO command.

```
   "EXECIO  *  DISKW  ..."
```

When you specify *, the EXECIO command will continue to pull items off the data stack until it finds a null line. If the stack becomes empty before a null line is found, EXECIO will prompt the terminal for input until the user enters a null line. Therefore, if you do not want to have terminal I/O, queue a null line at the bottom of the stack to indicate the end of the information.

```
   QUEUE ''
```

If the data contains null lines and the data stack is not shared, you can use the built-in QUEUED() function to determine the number of lines to write in either of the two fashions below:

```
    n = QUEUED()
   "EXECIO" n "DISKW  outdd (FINIS"
```

or

```
   "EXECIO" QUEUED() "DISKW  outdd (FINIS"
```

Just as with reading a data set, before you can write to a data set it must first be allocated to a ddname. You can allocate from within the exec with the ALLOCATE command as shown in the following example, or you can allocate before the exec runs.

```
   "ALLOC DA(output.data) F(myoutdd) OLD REUSE"
   "EXECIO  *  DISKW  myoutdd  ..."
```

To write the information from compound variables or a list of variables beginning with the name specified after the STEM keyword. The variables, instead of the data stack, holds the information to be written.

```
   "EXECIO  *  DISKW  myoutdd  (STEM newvar."
```

In this example, the variables would have the stem **newvar.** and lines of information from the compound variables would go to the data set. Each variable is labeled **newvar.1**, **newvar.2**, **newvar.3**, and so forth. The variable **newvar.0** is not used when writing from compound variables. When * is specified with a stem, the EXECIO command stops writing information to the data set when it finds a null value or an uninitialized compound variable.

The EXECIO command can also specify the number of lines to write from a list of compound variables.

```
   "EXECIO  5  DISKW  myoutdd  (STEM newvar."
```

In this example, the EXECIO command writes 5 items from the **newvar** variables including uninitialized compound variables, if any.

EXECIO Examples

```
/* REXX - copy a sequential data set */
"ALLOC DA('userid.input') F(datain) SHR REUSE"
"ALLOC DA('userid.output') F(dataout) LIKE('userid.input') NEW"
"EXECIO * DISKR datain (FINIS"
QUEUE '' /* Add a null line to indicate the end of the information */
"EXECIO * DISKW dataout (FINIS"
"FREE F(datain dataout)"
```

```
/* REXX */
/**************************************************************/
/* This exec will read an input file of data set names and   */
/* create IDCAMS "DELETE NOSCRATCH" and "DEFINE NONVSAM"      */
/* control cards to indirectly catalog sysres data sets.      */
/**************************************************************/
arg catname
if catname = '' then
   catname = 'ICFCAT.MASTER'   /* <=== catalog name  */
/*****************************************/
/* allocate input and output files       */
/*****************************************/
"ALLOC F(INPUT)  DA('MY.PDS.CNTL(INPUT)')   SHR REUSE"
"ALLOC F(OUTPUT) DA('MY.PDS.CNTL(OUTPUT)')  SHR REUSE"
/*****************************************/
/* read input file into stem variables   */
/*****************************************/
"EXECIO  *  DISKR INPUT (STEM INREC. FINIS"
/*****************************************/
/* process input data                    */
/*****************************************/
do i = 1 to inrec.0
  inrec.i = word(inrec.i,1) /* remove leading and trailing blanks   */
  queue  ' DEL ('||inrec.i||') -'
  queue  ' NSCR CAT('||catname||')'
  queue  ' /*                          */ '
  queue  ' DEF NVSAM(NAME('||inrec.i||') -'
  queue  ' VOL(******) DEVT(0000)) -'
  queue  ' CAT('||catname||')'
  queue  ' /*********************************************/ '
end  /* do i */
queue '' /* null line to signal end of data stack */
/*****************************************/
/* write output file and exit            */
/*****************************************/
"EXECIO * DISKW OUTPUT (FINIS"
"FREE  F(INPUT OUTPUT)"
exit 0
```

```
/* REXX */
/******************************************/
/* Scan file for PARM string              */
/*                                        */
/* INPUT  DDNAME  - SCANIN                */
/* OUTPUT DDNAME  - SCANOUT               */
/******************************************/
parse arg SEARCH /* preserve arg case     */
if SEARCH = '' then do  /* no srch parm   */
  say ' ******************************'
  say ' *    NO SCAN PARM SUPPLIED   *'
  say ' ******************************'
  exit 12
end /* if search */
/******************************************/
/* allocate input and output files        */
/******************************************/
"ALLOC FI(SCANIN)   DA('SYS3.SYSLOG.G0521V00') SHR REUSE"
"ALLOC FI(SCANOUT)  DA(*) REUSE"
/******************************************/
/* Initialize counters                    */
/******************************************/
FOUND  = 0 /* records found counter       */
RECNUM = 1 /* current record number       */
/******************************************/
/* Process input file                     */
/******************************************/
do forever
  "EXECIO 1 DISKR SCANIN "RECNUM /* read 1 record     */
  if RC <> 0 then leave /* no more records, exit loop */
  RECNUM = RECNUM+1    /* bump up record count by 1   */
  parse pull INREC     /* pull record from data stack */
  if pos(SEARCH,INREC) <> 0 then do
    push INREC         /* write record to data stack  */
    FOUND = FOUND+1    /* bump up found counter by 1  */
    "EXECIO 1 DISKW SCANOUT" /* write 1 record        */
  end /* if pos(search,inrec) */
end /* do forever */
/******************************************/
/* close output files                     */
/******************************************/
"EXECIO 0 DISKR SCANIN  (FINIS"
"EXECIO 0 DISKW SCANOUT (FINIS"
/******************************************/
/* write totals and exit                  */
/******************************************/
RECNUM = right(RECNUM-1,7,'0') /* 7 digit number w/leading zeros */
FOUND  = right(FOUND,7,'0')    /* 7 digit number w/leading zeros */
say  RECNUM' RECORDS WERE READ FROM THE INPUT FILE.'
say  FOUND' RECORDS WERE FOUND WITH "'||SEARCH||'".'
if FOUND = 0 then exit 4  /* if no matches found, end with RC=4  */
  else exit 0
```

Running REXX Execs

Foreground REXX Execution

Interactive execs and ones written that involve user applications are generally run in the foreground. You can invoke an exec in the foreground:

- Explicitly with the EXEC command.
- Implicitly by member name if the PDS containing the exec was previously allocated to SYSPROC or SYSEXEC.
- From another exec as an external function or subroutine, as long as both execs are in the same PDS or the PDSs containing the execs are allocated to a system file, for example SYSPROC or SYSEXEC.
- From a CLIST or other program.

REXX routines may be executed explicitly by entering EX on the member line in a PDS enhanced member list (ISPF option 3.4), at the TSO READY prompt, under the ISPF Command Shell (option 6), or by entering EXEC followed by the data set name and member to be executed:

```
      EXEC `userid.REXX.EXEC(MYEXEC)' 'arg1 arg2' EXEC
```

The "EXEC" operand may be omitted as long as the REXX exec begins with a comment line containing the word REXX , otherwise it will be interpreted as a CLIST.

```
      EXEC `userid.REXX.EXEC(MYEXEC)' 'arg1 arg2'
```

REXX routines may be executed implicitly if the data set containing the exec is allocated to a system file (SYSPROC or SYSEXEC). When both system files are available, SYSEXEC is searched before SYSPROC. As with explicit execution, REXX routines in SYSPROC must begin with a comment line containing the word REXX, otherwise it will be interpreted as a CLIST. REXX routines in SYSEXEC do not have this restriction, however it is always a good idea to have the comment line with the word REXX as the first line in an exec.

You may also specify alternate libraries for implicit execution with the ALTLIB command.
The ALTLIB command gives you more flexibility in specifying exec libraries for implicit execution. With ALTLIB, you can easily activate and deactivate exec libraries for implicit execution as the need arises.

To implicitly execute your REXX exec you enter the command at the READY prompt, from the COMMAND option of ISPF, or on the command line of any ISPF screen as long as the member name is preceded by "TSO %command" (without the quotes). The percent sign (%) is optional, if there is not a TSO command with the same name. However, when you use this form, called the extended implicit form, TSO/E searches only the ALTLIB or SYSPROC libraries for the name, thus reducing the amount of search time.

```
      TSO %MYEXEC arg1 arg2
```

Since SYSPROC and SYSEXEC are usually allocated to your session via the LOGON PROC, and you may not have update access to those data sets, your best options for executing your execs is to use the explicit method outlined above or use ALTLIB to define an APPLICATION data set that contains your execs allowing you to use the implicit execution method. This will also keep your private execs from interfering with other execs that may have the same name.

The ALTLIB command lets you specify alternative libraries to contain implicitly executable execs. You can specify alternative libraries on the user, application, and system levels.

- The user level includes exec libraries previously allocated to the file SYSUEXEC or SYSUPROC. During implicit execution, these libraries are searched first.

- The application level includes exec libraries specified on the ALTLIB command by data set or file name. During implicit execution, these libraries are searched after user libraries.

- The system level includes exec libraries previously allocated to file SYSEXEC or SYSPROC. During implicit execution, these libraries are searched after user or application libraries.

Data sets concatenated to each of the levels can have differing characteristics (logical record length and record format), but the data sets within the same level must have the same characteristics.

The ALTLIB command offers several functions, which you specify using the following operands:

ACTIVATE	Allows implicit execution of execs in a library or libraries on the specified level(s), in the order specified.
DEACTIVATE	Excludes the specified level from the search order.
DISPLAY	Displays the current order in which exec libraries are searched for implicit execution.
RESET	Resets searching to the system level only (execs allocated to SYSEXEC or SYSPROC).

Here is an example of the ALTLIB command allocating a APPLICATION exec library:

```
ALTLIB ACT APPLICATION(EXEC) DSNAME('userid.REXX.EXEC')
```

If you are in split-screen mode in ISPF and you issue the ALTLIB command from a one-screen session, the changes affect only that screen session. The ALTLIB search order is not valid across split screens.

Background REXX Execution

When you run an exec in the foreground, you do not have use of your terminal until the exec completes. Another way to run an exec is in the background, which allows you full use of your terminal while the exec runs. You can run time-consuming and low priority execs in the background, or execs that do not require terminal interaction.

Running an exec in the background is the same as running a CLIST in the background. The program IKJEFT01 sets up a TSO/E environment from which you can invoke execs and CLISTs and issue TSO/E commands. For example, to run an exec named MYEXEC contained in a partitioned data set USERID.REXX.EXEC, submit the following JCL:

```
//USERIDA   JOB  'ACCOUNT,DEPT,BLDG','PROGRAMMER NAME',
// CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*
//TMP      EXEC PGM=IKJEFT01,REGION=4M
//SYSEXEC  DD DISP=SHR,DSN=USERID.REXX.EXEC
//SYSTSPRT DD   SYSOUT=*
//SYSTSIN  DD   *
   %MYEXEC arg1 arg2
/*
//
```

The EXEC statement defines the program as IKJEFT01. The REXX routine (and any called routines) must be in a PDS defined to the SYSEXEC or SYSPROC system file. You can assign one or more PDSs to SYSEXEC or SYSPROC. The SYSTSPRT DD allows you to print output to a SYSOUT class or a specified data set. In the input stream, after the SYSTSIN DD, you can issue TSO/E commands and invoke execs and CLISTs.

Any additional files needed to run the exec can be defined with the TSO ALLOCATE command within the exec or in the SYSTSIN input stream (prior to executing the exec), or by adding additional JCL DD statements.

Another way to run REXX execs in batch is to use the IRXJCL program. Running an exec in batch using IRXJCL is similar in many ways to running an exec in the TSO/E background using IKJEFT01, however, there are differences. One major difference is that the exec running in batch via IRXJCL cannot use TSO/E services, such as TSO/E commands (like ALLOCATE) and most of the TSO/E external functions.

```
//USERIDA   JOB  'ACCOUNT,DEPT,BLDG','PROGRAMMER NAME',
// CLASS=A,MSGCLASS=X,MSGLEVEL=(1,1)
//*
//*  RUN REXX EXEC IN BATCH
//*
//*  PARM VALUE IS THE CLIST AND ARGUMENTS TO BE EXECUTED
//*  SYSEXEC DD POINTS TO THE EXEC PDS
//*
//S1  EXEC PGM=IRXJCL,PARM='MYEXEC arg1 arg2'
//SYSTSIN  DD DUMMY
//SYSTSPRT DD SYSOUT=*
//SYSEXEC  DD DISP=SHR,DSN=USERID.REXX.EXEC
```

Since execs running in batch via IRXJCL cannot use TSO/E services, any additional files needed to run the exec must be defined by adding JCL DD statements.

Final Test / Discussion Question

Here is a small REXX exec. When this program is executed, what must be entered at the terminal in order to get the exec to respond "okie dokie"?

```
/* REXX */
Say 'Enter value for number:'
pull number
color = blue
if number = 7 but color is not green
  then say okie dokie
  else say get outta here
```

Writing ISPF Edit Macros with REXX

The information below is a general overview of edit macros. Creating edit macros can be complicated and the detail goes beyond the scope of this REXX introduction. For more information please see the OS/390 ISPF Edit and Edit Macros manual.

Edit macros are ISPF dialogs that run in the ISPF editor environment. You can use edit macros, which contain statements that look like ordinary editor commands, to extend and customize the ISPF editor. You can use edit macros to:

- Perform repeated tasks
- Simplify complex tasks
- Pass parameters
- Retrieve and return information

Edit macros can be either CLISTs or REXX execs written in the CLIST or REXX command language, or program macros written in a programming language (such as FORTRAN, PL/I, or COBOL). Our focus here will only be on REXX. Since REXX is such a powerful language, REXX edit macros have virtually endless possibilities.

In order to use an edit macro, it must reside in partitioned data set that is part of the implicit search order (see the previous section).

REXX edit macros are made up of REXX statements that fall into one of the following categories:

- Edit macro commands
- REXX command procedure statements and comments
- ISPF and PDF dialog service requests
- TSO commands

All statements are initially processed by the TSO command processor, which scans them and does symbolic variable substitution. It is important to recognize the different kinds of CLIST and REXX statements listed because:

- They are processed by different components of the system.
- They have different syntax rules and error handling.
- Their descriptions are in different manuals.

Since edit macros need to use ISPF services, they must run under the ISPEXEC or ISREDIT host command environment (ADDRESS ISPEXEC or ADDRESS ISREDIT). ISREDIT passes commands directly to the PDF editor and ISPEXEC is needed for using other ISPF services within an edit macro.

Any statement in an edit macro that begins with ISREDIT is assumed to be an edit macro command or assignment statement:

```
ADDRESS ISPEXEC
"ISREDIT FIND 'PGM='"    /* edit macro command              */
"ISREDIT BOUNDS = 1,50"  /* edit macro assignment statement */
```

If you have several edit macro commands within a REXX EXEC, you can change the host command environment default to the PDF editor with the **ADDRESS ISREDIT** instruction.

```
ADDRESS ISREDIT
"FIND 'PGM='"    /* edit macro command              */
"BOUNDS = 1,50"  /* edit macro assignment statement */
```

All edit macros must have an ISREDIT MACRO statement as the first edit command.

```
/* REXX - this is an edit macro */
ADDRESS ISPEXEC
"ISREDIT MACRO"
"ISREDIT CHANGE JANUARY FEBRUARY ALL"
```

The MACRO command is also used to pass parameters to an edit macro. Parameters are enclosed within parenthesis after the MACRO command. Multiple parameters are separated by a blank or comma. Example:

```
/* REXX - MYMAC edit macro */
ADDRESS ISPEXEC
"ISREDIT MACRO (PARM1,PARM2,REST)"
```

If you executed this edit macro with parameters like this:

```
Command ===> MYMAC FIRST SECOND THIRD FOURTH
```

PARM1 would be set to **FIRST**, PARM2 to **SECOND**, and REST would be assigned the value **THIRD FORTH**.

Here is a simple edit macro called "INCL". It can be used to exclude all lines in the editor except for a particular sting. The string to include is passed as a parameter to the macro. To use it you would type the following on the command line: `Command ===> INCL DSN=`

```
/* REXX */                                  /* REXX */
ADDRESS ISPEXEC                             ADDRESS ISREDIT
"ISREDIT MACRO (parm1)"        OR           "MACRO (parm1)"
"ISREDIT EXCLUDE ALL"                       "EXCLUDE ALL"
"ISREDIT FIND ALL" parm1                    "FIND ALL" parm1
```

You can use most primary commands in an edit macro if you precede it with ISREDIT. Some commands have additional operands permitted in a macro that cannot be used interactively. You cannot issue line commands directly from an edit macro. For example, you cannot use the M (move) line command within an edit macro. However, you can perform most of the functions provided by line commands by using a combination of primary commands, edit macro commands and edit assignment statements.

Edit macros use edit assignment statements to communicate between macros and the editor. An assignment statement consists of two parts, values and keyphrases, which are separated by an equal sign.

Data is always transferred from the right-hand side of the equal sign in an assignment statement to the left side. Therefore, if the keyphrase is on the right, data known to the editor is put into REXX variables on the left. In the following example, yyy would be a keyphrase, and xxx would be the value.

```
      ADDRESS ISPEXEC
      "ISREDIT xxx = yyy"
```

Edit assignment statements can be used to:

- Set or query edit parameters, for example CAPS, NULLS, and NUMBER
- Pass values back and forth between the editor and the macro
- Manipulate data (insert, delete, or replace lines)

Here are some examples:

```
ADDRESS ISPEXEC
"ISREDIT CAPS = OFF"       /* set CAPS off */
"ISREDIT (capmode) = CAPS" /* puts value of CAPS into var capmode */
"ISREDIT CAPS = "capmode   /* set CAPS to whatever is in capmode var */
```

In the first statement, CAPS are set to a value of OFF. The second statements queries the current value of CAPS and places it into variable **capmode**. The third statement sets the value of CAPS to whatever is in the **capmode** variable. If these instructions were executed in this order, the last statement would be equivalent to the first one.

```
ADDRESS ISPEXEC
"ISREDIT (lb,rb) = BOUNDS"
"ISREDIT BOUNDS = (lb,rb)"
```

In the first statement, the current left and right boundaries are stored into the variables **lb** and **rb**. In the second statement, the values from the variables **lb** and **rb** are used to change the current boundaries.

```
ADDRESS ISREDIT
"(linedat) = LINE "curline
"LINE" curline "= (linedat)"
```

In the first statement, the data from line number "curline" (a variable used in the exec) is stored into variable **linedat**. In the second statement, the data from variable **linedat** is used to change the contents of the editor at line number "curline".

Sample Edit Macros

This simple edit macro can be used to set up the ISPF profile defaults that you prefer.

```
/* rexx  MYPROF edit macro      */
ADDRESS ISREDIT
"MACRO"
"BOUNDS"
"TABS OFF"
"NULLS ON"
"HILITE AUTO"
"RECOVERY ON"
```

This edit macro, called "DELX" will delete everything but a specified string. The search can optionally be limited to specific columns. Example: DELX 'DISP=(NEW'

```
/* REXX - DELX edit macro  */
Address ISREDIT  /* ISREDIT host command environment       */
"MACRO (parm col1 col2)"       /* MACRO parms              */
"(lastln) = LINENUM .ZLAST"    /* find out how many lines  */
"SEEK "parm col1 col2" FIRST"  /* any thing found?         */
If rc = 0  then do             /* we found something       */
   "EXCLUDE ALL"               /* exclude all lines        */
   "FIND "parm col1 col2" ALL" /* UN-exclude lines         */
   "SEEK "parm col1 col2" ALL" /* get counts for later     */
   "DELETE ALL X"              /* delete excluded lines    */
   "(count,lines) = SEEK_COUNTS"  /* save counts for later */
   count = Format(count)  /* remove leading zeros          */
   lines = Format(lines)  /* remove leading zeros          */
   deleted = lastln-lines /* how many lines were deleted   */
   Upper parm   /* change parm to upper case for msg       */
   zedsmsg = deleted' LINES DELETED'  /* short msg         */
   zedlmsg = count 'OCCURRENCES OF "'parm'" WERE KEPT',
              'ON 'lines 'LINES - 'deleted 'LINES WERE DELETED.'
   "RESET"    /* re-display excluded lines               */
   Address ISPEXEC "SETMSG MSG(ISRZ000)" /* msg - no alarm */
   Exit 0   /* exit and set rc to 0                        */
End
Else do         /* string not found                        */
   Upper parm   /* change parm to upper case for msg       */
   zedsmsg = 'STRING NOT FOUND'
   zedlmsg = 'THE STRING "'parm'" WAS NOT FOUND IN THE FILE.'
   "RESET"  /* re-display excluded lines                   */
   Address ISPEXEC "SETMSG MSG(ISRZ001)" /* msg with alarm */
   Exit 12  /* exit and set rc to 12                       */
End
```

TSO/E OUTTRAP Example

This exec, called "TSOB" uses the OUTTRAP TSO/E external function to trap command output and store it in a compound variable. Note that OUTTRAP may not trap all of the output from a TSO/E command - it depends on the type of output that the command produces. For example, TSO/E commands that use the TPUT macro will not be trapped, but in general, the OUTTRAP function traps all output from a command.

This exec is very useful for browsing the output of RACF or TOP-SECRET commands, LISTCAT commands, and other commands where the output may exceed a single screen's rows or columns. You would invoke the exec in one of the following ways:

```
  TSO %TSOB cmd    (the exec must be in the implicit search order)
  TSO EXEC 'REXX.EXEC(TSOB)' 'cmd'
```

EXAMPLES:

```
  TSO %TSOB LISTC EN('VSAM.CLUSTER') ALL
  TSO %TSOB HELP ALLOC
  TSO EX 'REXX.EXEC(TSOB)' 'LISTA STA'
```

```
/* REXX - TSOB - trap and browse TSO/E command output          */
arg TSOCMD
/* Let this exec process return codes of 12 or higher          */
address ISPEXEC "CONTROL ERRORS RETURN"
address TSO         /* address TSO environment                 */
ddnm = 'DD'||random(1,99999)    /* choose random ddname        */
junk = msg(off)     /* turn off TSO/E messages                 */
/* Allocate data set to hold cmd o/p                           */
"ALLOC FILE("||ddnm||") UNIT(SYSDA) NEW TRACKS SPACE(9,9) DELETE",
" REUSE LRECL(172) RECFM(F B) BLKSIZE(8944)"
junk = msg(on)      /* turn on TSO/E messages                  */
/* issue TSO command and trap output                           */
junk=outtrap(LINE.) /* start trapping cmd o/p in "LINE." var */
TSOCMD              /* issue command                           */
junk=outtrap('off') /* stop trapping command output            */
/* write output of the command from stem variable "LINE."     */
/* to the previously allocated data set.                       */
"EXECIO" line.0  "DISKW" ddnm "(STEM LINE. FINIS"
/* Use ISPF LM services to browse                              */
/* the previously allocated data set.                          */
address ISPEXEC "LMINIT DATAID(TEMP) DDNAME("||ddnm||")"
address ISPEXEC "BROWSE DATAID("||temp")"
address ISPEXEC "LMFREE DATAID("||temp")"
/*                                                             */
junk = msg(off)     /* turn off TSO/E messages                 */
"FREE FILE("||ddnm||")"
```

CLIST to REXX Equivalents

CLIST	REXX
/* ANY COMMENT	/* ANY COMMENT */
WRITE	SAY
WRITENR &ZUSER WRITE IS AUTHORIZED	SAY ZUSER "IS AUTHORIZED"
PROC n	ARG or PARSE ARG
&VAR	VAR
SET X = &SUBSTR(3:8,&A)	X = SUBSTR(A,3,6) or PARSE VAR X 3 A 9
&STR()	"" or ''
&STR(&X)	X
&X&Y	X || Y
PRFX&MIDVAR.SUFFIX	"PRFX"MIDVAR"SUFFIX"
CONTROL CONLIST SYMLIST LIST	TRACE RESULTS
CONTROL END(ENDO)	(no equivalent)
ISPEXEC ispf service	ADDRESS ISPEXEC "ispf service"
WHEN (&A = &B)	WHEN A = B THEN
&NOP (as statement)	NOP
+ and - as continuation characters	, (comma)
SET A = &B &C	A = B C
SET A = &B.&C	A = B || C
SET A = &B.C	A = B"C"
READDVAL	PULL
OPENFILE, GETFILE, PUTFILE, CLOSEFILE	"EXECIO * ..." with PUSH, PULL
(a TSO command)	"a TSO command"
AND, &&	&
OR	|
GE, GT, EQ, NE, LE, LT	>=, >, =, ¬=, <= < (¬=, /=, \=, <>, >< are all valid ways of coding "not equal")

The following keywords have identical or nearly identical meanings in both CLIST and REXX:
IF, THEN, ELSE, DO, END, SELECT, WHEN, and OTHERWISE

REXX- GUIDELINES

Is REXX better than ?

Short answer: Yes. No. Maybe. Does it matter?

Long answer: This question wastes a lot of bandwidth in comp.lang.rexx and other newsgroups. Every language has its good points and its bad points. Some people love REXX, some people hate it. Use a language that suits your needs.

Why does my OS/2 REXX program run more quickly the second time?

When you run a REXX CMD file for the first time, a tokenized version will be stored on disk using the OS/2 extended file attributes. (You can see how big the tokenized version is by using the /N option on the DIR command.) If a tokenized version exists AND the file has not been modified, CMD.EXE will use the tokenized version instead of parsing the source.

Note that there is a 64K limit on the size of an extended attribute entry, so very large REXX programs do not benefit from this automatic tokenization.

How can I return multiple values from a function?

REXX does not provide any support for returning more than a single value from a function. If you wish to return multiple values you must devise an alternate scheme. A simple solution is to concatenate the values together into a single string and on return from the function use the PARSE instruction or the various string functions to split the string back into its elements.

Don't forget that you can use non-printable characters (such as '01'x) to separate the data -- REXX will correctly handle such strings. There may also be other alternatives available to you if you are using an external function library that lets you store data in separate memory pools or in disk files.

Here is one example of a way you can return multiple values from a routine and extract them easily.

```
Call MyFunction
Parse Var result With v1 v2
v1 = X2C(v1)
v2 = X2C(v2)
Exit
MyFunction:
 Return C2X(v1) C2X(v2)
```

Why does linein, lineout, charin or charout fail?

Most versions of REXX (ARexx is an exception) use implicit file opening. That is, each time you reference a file in a LINEIN, LINEOUT, CHARIN or CHAROUT function, REXX will open the file for reading or writing if the file is not already open. However, some operating systems, like DOS and OS/2, impose limits on the number of files that can be open simultaneously, usually around 20 or so.

After the limit has been reached, any further attempts to open another file will fail. That is why it is always good practice to close a file when you're done with it. In OS/2 this is done using the STREAM function, as follows:

```
call stream "c:\foo.out", "command", "close"
```

The STREAM function can also be used to open files, query their sizes and seek into the file. Consult your REXX documentation for specific instructions for your interpreter.

How do I iterate over all the tails in a stem variable?

One of the features REXX lacks is a function to return a list of defined tails. There are external libraries that provide functions to do so, but if that is not an option then the only solution is to maintain your own list of tails in a string and use the PARSE instruction or the WORDS function to traverse the list.

How do I REXX-enable my application?

REXX-enabling an application means being able to run REXX macros within an application. This information is very system-specific, so the best place to start is with the documentation provided with the REXX interpreter.

For OS/2, there are several sources of information. The most basic information is found in the OS/2 Toolkit, which includes the REXXSAA.H header file and the REXX Reference online document. The REXX Report includes a couple of articles on the subject.

Sample source code comes with the OS/2 Toolkit and is also available at rexx.uwaterloo.ca in the directory /pub/os2/vxrexx as VX-REXX Tech Notes #1 and #7 (vxtech01.zip, vxtech07.zip -- neither tech note requires that you own VX-REXX).

OS/2 technical conferences such as ColoradOS/2 or the IBM Technical Interchanges often includes sessions on this topic. For ARexx, a book was available from Commodore, but with the latter's demise it is unclear whether the book is still available.

How do I do inter-process communication in REXX?

Again, this is system-specific. The ARexx interpreter is built on a messaging model, making it very simple to do inter-process communication, but the OS/2 REXX interpreter has no such features, though in some cases queues can be used to achieve the desired effect.

How do I use global variables in my REXX programs?

The scope of variables is controlled by the PROCEDURE instruction. If a routine is declared with the PROCEDURE instruction, only those variables exposed using the EXPOSE instruction are available to the routine.If no PROCEDURE instruction is used, all of the caller's variables are available to the callee.

Here is a simple example:

```
a = 10
b = 20
call first
call second
call third
exit
first:
  say "first -- a is" a "b is" b
return
second: procedure
  say "second -- a is" a "b is" b
return
third: procedure expose a
  say "third -- a is" a "b is" b  b = 30
  call first
return
```

Running this program yields the following output

```
first         --         a        is         10         b         is         20
second         --         a        is         A          b         is         B
third         --         a        is         10         b         is         B
first -- a is 10 b is 30
```

Use the PROCEDURE instruction to keep variables local to a procedure, using EXPOSE to explicitly expose any "global" variables. The only catch is that you have to make sure you expose the variables inside every procedure.

One way to define and use global variables is to use a stem called "Globals." and define all your procedures like this:

```
Foo: procedure expose Globals.
```

Then at the top of you program initialize the Globals stem and assign appropriate values to your global variables

```
Globals. = ''
Globals.!NeedToSave = 0
Globals.!TmpDir = "D:\TMP"
```

The tail names in this example are all prefixed with '!', though you could also use an underscore ('_'). This is just a convention used to avoid this kind of problem:

```
Globals.TmpDir = "D:\TMP"
call Foo
```

```
say Globals.TmpDir
exit
Foo: procedure expose Globals.
  tmpdir = "foo"
  Globals.TmpDir = tmpdir
return
```

It's a subtle bug that has to do with how REXX interprets stem tails.

Why are my strings always in upper case?

Inadvertent upper case translation commonly happens for 2 reasons.

Use of an implicit literal instead of a quoted string.

```
X = Mixed Case /* instead of */
X = "Mixed Case"
```

Use of ARG or PULL instead of PARSE ARG or PARSE PULL