Rexx Reference Manual (TSO)

by David Grund

Rev 10 – May 13, 2012
Rev 9 – July 20, 2011
Rev 8 – June 18, 2011
Rev 7 – April 13, 2011

Table of Contents

Rexx Reference Manual (TSO)

Rexx is the Restructured Extended Executor Language. New with TSO/E version 2, Rexx is a high-level procedural language that allows programmers to mix instructions with TSO commands, and build high-powered tools and utilities, called “exec”s.

Rexx is a programming language, and a scripting language. Rexx is a fascinating language. It is, from my viewpoint, IBM's answer to Basic. It is an English-like interpreted language. No compiler is needed. The computer reads the instructions, one at a time, and if it can interpret it, it will execute it.

The thing that Rexx is best suited for is to create data-manipulation tools, especially for one-time use or for development. Once you learn how to use the language, you can tailor data in ways you never dreamed.

Why learn Rexx? Knowing Rexx can give you a powerful advantage. Being able to manipulate data in esoteric and creative ways can be a tremendous aid to your productivity. Rexx can be very useful for creating and verifying test data, formatting output data, file-integrity-checking, and creating tools that help make your job easier. More than anything else now, Rexx is used to drive ISPF dialogues.

A Rexx program can be written far more quickly than one for COBOL or Assembler, for the same task at hand. You wouldn't want to use Rexx in production for high-volume files, though. That's the job of a compiled program. Rexx is for the "quickie", and low-volume tasks.

The reason I decided to write this book, is that with the reference I was using, it took too long to find information that I was looking for. The author of that book knew his stuff, but I felt he had no clue as to what to present, or how to present it. The organization of that book and lack of meaningful examples was frustrating, and simply not acceptable.

This reference discusses Rexx and its use with TSO, as opposed to CMS or personal computers. The intended audience for this book is all levels of mainframe programmers, and "computer-literate" users. You should be at least familiar with TSO, have a TSO UserID, and be able to log on to a mainframe.

I wrote this from the point of view of a Rexx user, and not a teacher. While I was writing this, I envisioned real-life situations that I could find myself in, and I then tried to illustrate the best way to handle it. The examples were not written for the sake of example; they were written to show *how* to solve a given problem. I added many examples from my real-life work experience. These are execs that I used to solve real problems.

So it is my hope that this reference is easy to use, has useful and pertinent examples, and can help you get your job done. This manual is the quickest way to get up, running, and productive in Rexx.

All of the examples in this book have been tested on an IBM mainframe, on Rexx370 Vers 3.48 01 May, 1992. Any errors resulting in the use of these examples would probably then be due to environmental differences, or the transfer of the example from this document.

If there is something about this book that really bugs you, or really pleases you, or if you have any other comments, criticisms, or suggestions, please feel free to e-mail me at: RexxManual@davidgrund.com.

This book is divided into five sections.

The first section is a reference, for the experienced programmer. I put this section first because I feel that this will be the one that is used the most. With this format, you don't have to worry about whether a Rexx component is a function, instruction, or anything else. Just flip through the alphabetically-sorted reference, find the keyword, read, and use!

The second section of this book is a short Rexx tutorial. This is where the beginner should start.

The third section of this book contains examples: *useful* examples.

The fourth section of this book is on the Rexx environment: how to establish and maintain it, and how to use it alone, and in conjunction with ISPF.

The fifth section of this book, the appendix, contains lists of instructions by class, and other Rexx references.

Section I - Reference
General Rules

Form: The Rexx language is generally free-form. You can put any number of spaces between instructions, operands, etc.

The elements of a Rexx exec are: Rexx instructions, Rexx functions, TSO external functions, and TSO commands.

These elements generally end at the end of a line or at the beginning of a comment, whichever comes first. They can be stacked on the same line if separated by semi-colons.

First Line: A Rexx exec is identified by the character string "REXX" (no quotation marks) in the first line of the exec, but only if allocated to SYSPROC. If the exec is part of a SYSEXEC library, then this is not necessary. Generally, it is recommended to start a Rexx exec off with a comment stating the name, short description, and "REXX" keyword, as follows:

```
/* MyFirst - MyFirst Rexx Program  */  or
/* Calc1 - My Calculator      Rexx */
```

Rexx is also case-insensitive. Use upper- or lower-case letters at your discretion. Note, however, that some functions look at the case of letters!

All values that appear in Rexx statements are translated to upper case unless they are enclosed in matching apostrophes or quotation marks.

In some cases, not all of the operands of an instruction are discussed. There are some operands that are highly esoteric, that I feel will be needed only in extremely specific situations. The appendix contains information on additional Rexx reference material.

The limit on the length of symbols (variable names) is 250 characters, although using one of that length is usually impractical.

The storage limit for any variable is 16MB.

Abbrev

Purpose: Return a 1 (TRUE) or zero (FALSE) based on the test that a word begins in a certain string. It is a subtle variation of the LEFT function.

Type: Rexx Function

Syntax: *Result* = ABBREV(*word,string,length*)

Usage: If the first *length* characters of *word* = *string*, then *result* will be TRUE.

Examples:
```
Result = ABBREV("America","Am",2)
/* TRUE; Result = 1 */

Result = ABBREV("America","mer",3)
/* FALSE; Result = 0 */
```

Abs

Purpose: Return the absolute value of a number: drop the sign, and format according to the current setting of NUMERIC DIGITS.

Type: Rexx Function

Syntax: *NewNum* = ABS(*OldNum*)

Example: NewNum = ABS(-436)
NewNum will be 436.

Address

Purpose: Return or change the setting of the environment that is currently receiving commands

Type: Rexx Function and Rexx Instruction

Syntax: 1) Address *Environ string* (instruction)
2) *Environ* = Address() (function)

Usage: Rexx passes to the environment any strings that are enclosed in quotation marks (or apostrophes), or any that it does not know what to do with.

1) The instruction form sets the environment that will receive these strings that are fed through by Rexx. This setting is "permanent" (for the duration of the current exec), unless it is supplied on the same line. If it is, then the setting that is specified is valid *only for the string on that line.* Rexx doesn't care what you set the environment to at the time you use this instruction. There is no validation at this point. The default is "TSO".
2) The function form simply returns the current environment setting

See Also: SubCom

Example 1: The following Rexx exec illustrates the use of the Address function and the Address instruction.

```
Say "Environ = " Address()
Address TSO
Say "Environ = " Address()
Address ISPEXEC
Say "Environ = " Address()
Address MVS
Say "Environ = " Address()
Address Junk
Say "Environ = " Address()
Address Dave
Say "Environ = " Address()
```

Will display:

```
Environ =  TSO
Environ =  TSO
Environ =  ISPEXEC
Environ =  MVS
Environ =  JUNK
Environ =  DAVE
```

Example 2: The following Rexx exec illustrates the *effects* of the use of Address:

```
1 "Browse Dataset(Rexx.Exec) "
2 address ispexec
3 "Browse Dataset(Rexx.Exec) "
4 address ispexec "Edit Dataset(Rexx.Exec) "
```

Explanation of the above exec:
1 This is a character string that Rexx does not understand, so Rexx passes it to the environment. Since the environment was not set, it remains as "TSO". TSO, in turn, does not know what to do with this character string, so the following displays:

```
COMMAND BROWSE NOT FOUND
     3 *-* "Browse Dataset(Rexx.Exec) "
       +++ RC(-3) +++
```

2 Rexx now sets the environment to "ISPEXEC" (the name for ISPF's environment).
3 ISPF receives this character string, and knows what to do with it, so it opens the dataset called "`Rexx.Exec`" for Browse.
4 This line is setting the environment at the same time as sending the string. ISPF then edits a dataset called "`Rexx.Exec`".

Example 3: This exec demonstrates the "temporary" environment setting.

```
1 address TSO
2 address ispexec "Edit Dataset(Rexx.Exec) "
3 "Browse Dataset(Rexx.Exec) "
4 address ISPEXEC
5 "Edit Dataset(Rexx.Exec) "
```

In the above exec,
Line 1 sets the environment to "TSO"
Line 2 will edit "Rexx.Exec", having set the environment temporarily to ISPEXEC.
Line 3 will err, because TSO does not recognize the command:

```
COMMAND BROWSE NOT FOUND
5 *-* "Browse Dataset(Rexx.Exec) "
  +++ RC(-3) +++
```

Line 4 will set the environment to ISPF
Line 5 will edit the dataset successfully.

Apostrophes

Purpose: To enclose a literal (character string).

See "Quotation Marks/Apostrophes" for documentation on this function.

Arg

Purpose: Retrieve data from the TSO command line or from a calling routine.

Type: Rexx Function and Rexx instruction

See "Parse" for documentation on this function.

Assignment Statement

Purpose: To assign a value to a variable. The value you assign to the variable can be any type: character, number, hex, binary, etc.

Syntax: Variable = ValueFormat

Variable The name of the variable being assigned. It can be up to 250 characters long, but I don't know why you would want to do that to anyone.

Value The value that you are assigning to the variable

Format The representation of the value. The default is character. Valid values are "X" for hexadecimal, and "B" for binary.

Examples:		
	A = 1	assigns the value '1' to the variable 'A'
	B = "F1F2F3F4"x	assigns the value '1234' to variable 'B'
	C = '11110010'B	assigns the value '2' (X'F2') to C

BitAnd

Purpose: Return a string that is the result of two strings that were logically AND'd together.

Type: Rexx Function

Syntax: Result = BitAnd(*string1*,*string2*,*padString*)
String1 and *String2* are the strings used in the AND operation.
padString is a string used for padding

Usage: To AND two strings is to multiply the bits of one string to the corresponding bits of the other string, and return the result. In English, it reads, "If the bit of the first string AND the corresponding bit of the second string are both on, then the resulting bit will be on. Otherwise, the resulting bit will be off."

padString is used to fill the shorter of the two strings (on the right) so the strings are the same length when being processed. If no padString is supplied, the operation works only for the length of the smaller string.

The sole purpose this function has is to do bit-level manipulation.

This function is the opposite of BitOr.

Example 1 The following example will convert a one-character reply from upper-case to lower case, by virtue of turning off bit 1:

```
ResultString = BitAnd('Y','10111111'B)
Say ResultString
```

The upper case 'Y' is X'E8', or B'11101000'.
The lower case 'y' is X'A8', or B'10101000'.

Example 2 The following example will convert all letters of a string to lower case (taking the above example a step further).

```
Sentence = "The Quick Brown Fox Jumps Over The Lazy Dog"
ResultString = BitAnd(Sentence,'10111111'B,'10111111'B)
```

Example 3 The following example does the exact same thing.

```
Sentence = "The Quick Brown Fox Jumps Over The Lazy Dog"
ResultString = BitAnd(Sentence,'BF'X,'BF'X)
```

Notice that the coding in this example is a little shorter, but not as clear to the reader: a binary 10111111 equals a hexadecimal BF. I prefer example 2 to example 3 because it is clearer.

BitOr

Purpose: Return a string that is the result of two strings that were logically OR'd together.

Type: Rexx Function

Syntax: Result = BitOr(*string1*,*string2*,*padString*)
String1 and *String2* are the strings used in the OR operation.
padString is a string used for padding

Usage: To OR two strings is to add the bits of one string to the corresponding bits of the other string (with no carry), and return the result. In English, it reads, "If either the bit of the first string OR the corresponding bit of the second string are on, then the resulting bit will be on. Otherwise, the resulting bit will be off."

padString is used to fill the shorter of the two strings (on the right) so the strings are the same length when being processed. If no padString is supplied, the operation works only for the length of the smaller string.

The sole purpose this function has is to do bit-level manipulation.

This function is the opposite of BitAnd.

Example 1 The following example will convert a one-character reply from lower-case to upper case, by virtue of turning on bit 1:

```
ResultString = BitOr('y','01000000'B)
```

The lower case 'y' is X'A8', or B'10101000'.
The upper case 'Y' is X'E8', or B'11101000'.

Example 2 The following example will convert all letters of a string to upper case (taking the above example a step further).

```
Sentence = "The Quick Brown Fox Jumps Over The Lazy White Dog"
ResultString = BitOr(Sentence,,'01000000'B)
```

String2 is padded to the length of *Sentence* with binary '01000000'.

Example 3 The following example does the exact same thing.

```
Sentence = "The Quick Brown Fox Jumps Over The Lazy White Dog"
ResultString = BitOr(Sentence,,'40'X)
```

Notice that the coding in this example is a little shorter, but not as clear to the reader: a binary 01000000 equals a hexadecimal 40. I prefer example 2 to example 3 because it is clearer.

BitXOr

Purpose: Return a string that is the result of two strings that were logically XOR'd together.

Type: Rexx Function

Syntax: Result = BitXOr(*string1*,*string2*,*padString*)
String1 and *String2* are the strings used in the AND operation.
padString is a string used for padding

Usage: To XOR two strings is to compare the bits of one string to the corresponding bits of the other string, one, by one, and return the result of the compare. In English, it reads, "If the bit of the first string AND the corresponding bit of the second string are the same, then the resulting bit will be off. Otherwise, the resulting bit will be turned on.

padString is used to fill the shorter of the two strings (on the right) so the strings are the same length when being processed. If no *padString* is supplied, the operation works only for the length of the smaller string.

If you XOR something to itself, the result will be hex zeroes.

The sole purpose this function has is to do bit-level manipulation.

You can use this instruction to do some rudimentary character-string encryption. See the example below.

Example 1 The following example will demonstrate the effect of this function.

```
ResultString = BitXOr('11111111'B,'01010101'B)
Say C2X(ResultString)
```

String1: 11111111
String2: 01010101
Result: 10101010 (X'AA')

Example 2 This example will further demonstrate the effect of this function.

```
ResultString = BitXOr('10101010'B,'01010101'B)
Say C2X(ResultString)
```

String1: 10101010
String2: 01010101
Result: 11111111 (X'FF)'

Example 3 This example demonstrates how to encrypt a character string. Use the exact same instruction to decrypt it.

```
Sentence = "The quick brown fox jumps over the lazy dog"
Say Sentence
Sentence = BitXOr(Sentence,,'BF'X)
Say Sentence
Sentence = BitXOr(Sentence,,'BF'X)
Say Sentence
```

Displays:

```
The quick brown fox jumps over the lazy dog
*:::::::::::: ::::::::::::::::¤:::::::::::
The quick brown fox jumps over the lazy dog
```

B2X

Purpose: Convert a binary string to a hexadecimal representation

Type: Rexx Function

Syntax: *Result* = B2X(*binarystring*)
Result is the hexadecimal representation of *binarystring*, which is a string of zeroes and ones.

Usage: Convert a binary to a hexadecimal number

Example 1 The following exec:

```
Say "B2X('11101111')=" B2X('11101111')
```

Will display the following:

```
B2X('11101111')= EF
```

Call

Purpose: To invoke, or transfer control to a subroutine (also commonly referred to as a procedure) or program, expecting to come back.

Type: Rexx Instruction

Syntax: Call *subroutine parameters*
Call *subroutine (parameters)*
Call *program*

Parameters Any number of variables that are intended to be used by the called subroutine.

Usage: A Call is used to facilitate structured programming. It is widely used to break the mainline processing up into blocks of code that are referenced by the mainline section. A lot of the examples illustrate structured programming and the use of Calls.

To call a Rexx exec or Clist *implicitely,* simply issue an "Address TSO" command, followed by the name of the Rexx exec or Clist, on separate lines.

A Call is also used to transfer control to a program, with the intention of regaining control. To call a program in the Linklist, you don't need to know the exact name of the library that the program resides in. Instead of issuing a Call, you issue the following command:
ADDRESS LINKMVS pgmname

Example 1

```
Call    Proc01          /* Call program section 1 */

Proc01:
        {code}
        {code}
Return
```

Example 2 The following code snippet is part of an exec that compares two disk files:

```
Address TSO
"Call 'SYS1.LINKLIB(IEBCOMPR)'"
If RC = 0 then
  Say "The modules are identical"
```

Example 3 The following code snippet is part of an exec that compares two disk files:

```
Address LinkMVS IEBCOMPR
If RC = 0 then
  Say "The modules are identical"
```

Example 4 Call a procedure, passing four variables.

```
Call Proc1(a b c) d
Call Proc1 e f g h
Exit

Proc1: Procedure
  Parse Arg p1 p2 p3
   Say "I am in Proc1. The parameters I was passed are " p1 p2 p3
Return
```

The output from this example code will be:

```
I am in Proc1. The parameters I was passed are A B C D
I am in Proc1. The parameters I was passed are E F G H
```

Call On

Purpose: Establish a subroutine to handle an error condition

Type: Rexx Instruction

Syntax: Call On *condition*

See "Trapping Errors" in the **Environment** section of this manual for a discussion of this instruction.

Center/Centre

Purpose: To center a string within a larger string

Type: Rexx Function

Syntax: Center(*string*,*length*,*pad*)

Usage: Center *string* within a larger string of *length* characters. If *pad* is present, it will be used as the pad character. If it is not, spaces will be used.

This function can be specified as either “Center” or “Centre”.

Example: The following excerpt of a Rexx Exec

```
Heading = "Tuesday"
Field = Center(heading,30,'-')
Say Field
```

will result in

```
-----------Tuesday------------
```

Clist

Purpose: Run a TSO command list the "old" way. This is what was used to accomplish the functions that Rexx Execs accomplish today.

Clists are mentioned here only because of their history and effect on today's Rexx language. I am in no way advocating using them. Anything you could do with a Clist can be accomplished with a Rexx exec, and usually cleaner.

Clists and Rexx execs alike are typically stored in a PDS (partitioned dataset). A Clist library is allocated to the DDName SYSPROC, while a Rexx exec library is allocated to the DDName SYSEXEC.

Comma

Purpose: To continue a Rexx statement

Example: The following Rexx Exec:

```
/* T1 - Example Rexx Program */
JanuarySales = 100
FebruarySales = 150
MarchSales = 5
AprilSales = 15
MaySales = 10
Total = JanuarySales + FebruarySales + ,
MarchSales + AprilSales + MaySales
Say 'The total sales = ' Total
```

will produce the result "280". Notice the continuation comma after *FebruarySales.*

Comments

Purpose: To document an exec, or annotate the lines within.

Syntax: Start with /* and with */. They can span any number of lines, but cannot be nested (supplied within another set).

Usage: Typically, you would comment each block of code with a comment line preceding that block of code. If you wish to comment one particular line, code the comment to the right of that line.

Example: /* This is a Rexx comment */

```
Say "Hello, World"      /* This is also a Rexx comment */

A = 1              /* Set the value of A to 1    */
B = 2              /* Set the value of B to 2    */
/* C = 3 */        /* This instr was commented out */
D = 4              /* Set the value of D to 4    */
```

Compare

Purpose: Compare two strings

Type: Rexx Function

Syntax: Result = COMPARE(string1,string2,pad)

Usage: Compare two strings, and return the number of the position where the inequality between the two strings starts. If the strings are equal, there is no inequality, and so the function returns a zero.

When one string is shorter than the other, it is first padded on the right with the pad character. The default pad character is a space.

Characters within quotation marks are treated with respect to their case. An upper-case letter will not equal a lower-case one.

Example:

```
Result = COMPARE("Apples","Oranges")
Say Result
```

Will yield 1, because the first position is unequal.

```
Result = COMPARE("Apples","Apple")
Say Result
```

Will yield 6, because the sixth position of the first string, "s", is unequal to the sixth position of the second string, which was padded to a blank.

```
Result = COMPARE("Apples","Apples   ")
Say Result
```

Will yield 0, because the strings after padding are identical.

```
Result =    Compare("Applesssssssss","Apples","s")
Say Result
```

Will yield 0, because the strings after padding are identical.

Comparison Operators

REXX comparison operations resolve to a 1 if the result of the comparison is true, and a 0 if the result of the comparison is false. REXX also uses an equality concept called 'strictly equal'.

Two values are 'strictly equal' if they match exactly, including imbedded blanks and the case of letters. Two values are 'equal' if they don't match exactly, but they resolve to the same quantity after REXX substitution and evaluation.

The following comparison operators can be used in REXX expressions:
== strictly equal
= equal
\== not strictly equal (can also use not sign, X'5F')
\= not equal (can also use not sign, X'5F')
> greater than
< less than
>< greater than or less than (same as not equal)
>= greater than or equal to
<= less than or equal to
\< not less than
\> no greater than

REXX Comparison Operators Order of Precedence:
\ - (not)
|| - concatenation
& - AND
| && - logical OR and EXCLUSIVE OR

Concatenation

Purpose: To combine two or more strings or literals into one variable.

One way concatenation is achieved by the use of "Or" bars. These are the vertical bars that can be found on the keyboard to the right of the +/= key. This is the preferred way, since it is *explicit*. If you use this method, all blanks between the two values that are being concatenated will be suppressed. If you want spaces between your variables, you must concatenate them as well. See example 1 below.

Another way to achieve concatenation is to simply put two variables *of different types* next to each other (juxtaposition). (Note that juxtaposition is accomplished by simply *not* using the "or" bars). Two or more intervening blanks will be compressed down to one. Again, if you want spaces between your variables, you must concatenate them as well. See example 2 below.

In summary, use the "Or" bars if you wish to strip out all intervening spaces. Use juxtaposition if you wish to keep just 1.

Examples using "Or" bars:

```
Say "Example 1"             ||    "Hello World"
Say "Example 1" || "Hello World"
Say "Example 1"||"Hello World"
```

All of the above will result in the same thing:

```
Example 1Hello World
```

Notice that all intervening spaces were removed by Rexx.

```
Say "Example 1"||" "||"Hello World"
```

will result in:

```
Example 1 Hello World
```

Notice the intervening space (between "1" and "Hello").

Examples using juxtaposition:

```
Say "Example 2""Hello World"
```

Notice that in this example, there is no legitimate concatenation. The quotation marks intended to define literals (variables of the same type). Instead, Rexx interpreted this as one string, and by its rules, translated two quotation marks into one.

```
Say "Example 2" "Hello World"
Say "Example 2"                "Hello World"
```

Both of the above examples will result in:

```
Example 2 Hello World
```

Condition

Purpose: Retrieve the setting information for the currently trapped REXX condition.

Type: Rexx Function

Syntax: *String* = CONDITION('*code*')
String is the returned setting. *Code* is supplied to request the type of information. The default is I.
Codes:
C- Return the name of the current condition
D- Return the descriptive string associated with the condition
I- Return the name of the actual instruction that was executing when the condition occurred
S- Return the status of the condition trap. This will be either ON, OFF, or DELAY.

Usage: This function is used in error trapping.

Example: In the following exec, we try to add Increase to Salary, neither of which has been defined:

```
      Signal On NoValue
      Salary = Salary + Increase
      Say "My salary = " Salary
      exit
      NoValue:
      Say "Undefined variable on line" SIGL
      Say "The current trapped condition is"
condition("C")
      Say "The variable is" condition("D")
      Say "The name of the instruction is"
condition("I")
      Say "The instruction is:" sourceline(SIGL)
      Say "The status of the condition trap is"
condition("S")
```

Will result in the following display

```
Undefined variable on line 4
The current trapped condition is NOVALUE
The variable is SALARY
The name of the instruction is SIGNAL
The instruction is: Salary = Salary + Increase
The status of the condition trap is OFF
```

Continuation

Purpose: To code an instruction that requires more than one line.

Syntax: Instructions are continued with a comma.

See **Comma** for documentation on this subject.

Copies

Purpose: Copies a string to itself a specified number of times.

Type: Rexx Function

Syntax: *Result* = COPIES(*string,quantity*)

Usage: Set *result* to *quantity* sets of *string.*

Example: Line = COPIES('*',75)
Will result in the variable "Line" containing 75 asterisks.

C2D

Purpose: Convert a string to its decimal equivalent

Type: Rexx Function

Syntax: *Result* = C2D(*string*)

Usage: Internally, the function first converts the string to its hexadecimal equivalent. Then it converts that hexadecimal value to decimal. It is the inverse of D2C.

Example:
```
result = C2D("  ")        /* Two spaces */
```
After execution of the previous instruction, *result* will contain 16448, the decimal representation of X'4040'

```
result = C2D("6")
```
result will contain 246, the decimal representation of X'F6'

C2X

Purpose: Convert a string to its hexadecimal equivalent

Type: Rexx Function

Syntax: *Result* = C2X(*string*)

Example:
```
result = C2X("  ")        /* Two spaces */
result will contain 4040

result = C2D("6")
result will contain F6
```

DataType

Purpose: This is a Rexx built-in function that will allow you to test to see the type of data a variable contains. There are two forms of this function.

Type: Rexx Function

Syntax 1: *Result* = DATATYPE(*string*)
If *string* was a number, *result* would contain "NUM". Otherwise, it would contain "CHAR".

Syntax 2: *Result* = DATATYPE(*string*,*type*)
Using this form, *result* will contain a one (TRUE) if *string* corresponds to *type*. Otherwise, it will contain a zero (FALSE).

Types:

Type	Description
A	Alphanumeric: A-Z, a-z, 0-9
B	Binary: 0 or 1 only
D	Double-byte character set
L	Lower-case letters
M	Mixed-case letters
N	Number
S	Symbol: valid Rexx symbol
U	Uppercase letters
W	Whole number
X	Hexadecimal number: 0-9, A-F

Examples: The following excerpt:

```
If datatype("Dave",M) then
      Say "Dave is mixed case"
else
      Say "Dave is not mixed case"
```

will display:

```
Dave is mixed case
```

Date

Purpose: This is a REXX built-in function that will provide you with the current date, in a variety of different formats.

Type: Rexx Function

Syntax: Result = DATE(option)
Based on the specification of the Options below, "result" will contain the date in the corresponding format, if the current date was April 8, 1997:

Usage: see the chart below

Option	Meaning	Format	Example
(blank) N	European	dd Mmm yyyy	8 Apr 1997
B	Basedate: Number of complete days since January 1, of the year 1.	nnnnnn	729121
C	Century: Number of days in this century	nnnnn	35527
D	Days: Number of days so far this year	nnn	98
E	European	dd/mm/yy	08/04/97
J	Julian date	yyddd	97098
M	Name of the current month	Mmmmmmmm	April
O	Ordered, suitable for sorting	yy/mm/dd	97/04/08
S	Ordered, suitable for sorting	yyyymmdd	19970408
U	USA format	mm/dd/yy	04/08/97
W	Name of current weekday	Ddddddd	Tuesday

option is not case sensitive. You can use either upper or lower case.

Examples: If today was April 8, 1997:

```
Today's date is date( )
```

will yield:

```
Today's date is 8 Apr 1997
```

```
Today is Date(M)
```

will yield:

```
Today is April
```

DelStack

Purpose: To delete the most recently-created TSO stack in preparation for use of it.

Type: TSO Command

Syntax: DELSTACK

Usage: Use this instruction right before you begin adding items to the TSO stack. This ensures that you don't inadvertently process data that was left on the stack by a previous program.

This is typically used in conjunction with the (Parse) Pull instruction.

DelStr

Purpose: Delete characters from a string

Type: Rexx Function

Syntax: *Newstring* = DELSTR(*string*,*start*,*length*)

Usage: Remove characters from *string* starting with position *start*, and for a length of *length*. The resulting string will be placed in *newstring*. The default for length is the entire remainder of the string.

Example:

Result = DELSTR("ABCDEFGHIJKLMNOPQRSTUVWXYZ",3,20)

After execution of this instruction, *result* will contain "ABWXYZ".

DelWord

Purpose: Delete words from a string

Type: Rexx Function

Syntax: *Newstring* = DELWORD(*string*,*start*,*quantity*)

Usage: Remove *quantity* words from *string* starting with word number *start*. The resulting string will be placed in *newstring*.

Example:

Result = DELWORD("FourScore and seven years ago, our fathers ",3,4)
After execution of this instruction, *result* will contain
"FourScore and fathers".

Digits

Purpose: Specify the number of digits that Rexx carries in arithmetic operations (precision).

Type: Rexx Function

Syntax: Numeric Digits n

See **Numeric Digits** for documentation on this function.

Do

Purpose: Execute a set of instructions, either under the control of a counter variable, or based upon current program conditions.

Type: Rexx Instruction

Syntax:
```
DO    expression
      variable=start
      TO limit
      BY increment
      WHILE expression
      UNTIL expression
      FOREVER
      (one or more statements to execute)
END   variable
```

Usage: There are several formats of the DO instruction. Each of the operands of the DO instruction as illustrated above are optional.

For the sake of explanation, the instructions in between the DO and END are commonly referred to as a DO Group.

If no operands are supplied, then the instructions in the DO Group are executed **one time**.

expression — Any valid REXX expression, but it must resolve to a positive whole number.

```
DO 19                       Do I = 19
(one or more instructions)  (one or more instructions)
END                         End
```

Both of the above examples would execute the instructions 19 times.

variable=start
TO limit
BY increment

Choose a control variable, and assign it a start value. This control variable is incremented by the BY amount on each iteration of the loop. The loop will stop when the control variable reaches the limit.

```
Do I = 1 TO UpperLimit BY 1
      (one or more instructions)
End
```

In the above example, I is the control variable. It starts with a value of 1, and the loop continues until I = UpperLimit.

BY can be a negative number if UpperLimit starts out to be less than the control variable.

WHILE expression
UNTIL expression

Continue to perform the instructions WHILE or UNTIL the expression is true. WHILE and UNTIL have opposite connotations. WHILE will test for a true condition *before* the do group is executed. UNTIL will test for a true condition a*t the end of* the do group. Using UNTIL assures you that the do group will execute at least one time.

```
Times = 1
StillIn = 'Y'
Do While StillIn = 'Y'
      Say "I am working on iteration number "Times
      Times = Times + 1
      If Times > 5 then StillIn = 'N'
End

Times = 1
StillIn = 'Y'
Do Until StillIn = 'N'
      Say "I am working on iteration number "Times
      Times = Times + 1
      If Times > 5 then StillIn = 'N'
End
```

The two examples above will produce identical results:

```
I am working on iteration number 1
I am working on iteration number 2
I am working on iteration number 3
I am working on iteration number 4
I am working on iteration number 5
```

Notice, however, that the only difference between the two examples is the expression following the conjunction (WHILE/UNTIL)

If you are in doubt as to which conjuntion to use, then apply the KIS principle (Keep It Simple). Use the one that makes the code easier to understand.

FOREVER Execute the do group continuously, until "told" to stop.

END Variable

A control variable name can be supplied to an END statement to clarify which DO group the END statement refers to.

```
Do I = 1 to 4
      Do J = 1 to 13
            (one or more instructions)
      End J
End I
```

Examples:

```
Do I = 1 to 25
      Say "Hello, world!"
End
```

The above example will print the message *Hello World!* 25 times.

```
Do I = 1 to 0
    Say "Hello, world!"
End
```
The above example will print nothing, because 0 < 1.

```
Do I = 1 to 10 by 2
      Say "Hello World #"I
End
```
The above example will print:
```
Hello World #1
Hello World #3
Hello World #5
Hello World #7
Hello World #9
```

```
Do I = 1 to 100 by 2 for 5
      Say "Hello World #"I
End
```
The above example will print:
```
Hello World #1
Hello World #3
Hello World #5
Hello World #7
Hello World #9
```
(Only five iterations)

```
/* Test1 - Example Rexx Program -   Rexx EXEC */
Do Forever
  Say "Tell me your name, or enter 0 to quit"
  Pull Answer
  If Answer = "0" then Leave
  Say "You told me that your name was" Answer"
End
```
The above Rexx exec will echo back whatever you type in, until you enter a zero :
```
You told me that your name was HOMER SIMPSON
```

Caution: Here is a common trap. After a do group completes, your index variable will be one higher than the limit. In the following example, assume you are traversing an array of records that you read in from a disk file, and that the disk file contained 114 records.

```
Do CurrRecNO = 1 to IPRec.0
      (processing…)
End
```
At this point, CurrRecNO will contain **115**, and not 114**.**

Drop

Purpose: "Unassign" a variable. This has the effect of converting a variable name to a literal (in upper case).

Type: Rexx Instruction

Example: The following excerpt from a Rexx exec:

```
Greeting = "Merry Christmas"
Say Greeting
Drop Greeting
Say Greeting
```

Will yield the following results:

```
Merry Christmas
GREETING
```

D2C

Purpose: Convert a decimal number to a character.

Type: Rexx Function

Syntax: *Result* = D2C(*number,length*)

Usage: Convert the decimal number to its internal hexadecimal format. It is the inverse of C2D.
Number must be a whole number or a variable containing a whole number. It must also be non-negative, unless *length* is specified.
Length is the length of the result, and is optional. If *Length* is not specified, *Result* will be left-zero-suppressed. If *Number* is negative, then *Length* is required.

Example:

The expression	displays
`Say D2C(240)`	`0 (X'F0')`
`Say D2C(240,5)`	`0 (right justified in a 5-byte field)`
`Say D2C(80)`	`& (X'50')`

D2X

Purpose: Convert a decimal number to a hexadecimal value.

Type: Rexx Function

Syntax: Result = D2X(number,length)

Usage: Convert a decimal number to its hexadecimal representation. It is the inverse of X2D.
Length- Length of the final result, in characters (optional)

Example:

The expression	displays
`Say D2X(240)`	`F0`
`Say D2C(80)`	`50`

The maximum value that can be converted is 999,999,999.

End

Purpose: Terminate a "DO" loop or block.

See the documentation on **Do** for more detailed information.

ErrorText

Purpose: This is a REXX built-in function that will return the English language text for an error code.

Type: Rexx Function

Syntax: Say ErrorText(RC)
where RC is the error code. Error codes are set by all Rexx errors.

Usage: To report back to the user, in English, what the problem is.

Example:

```
/* Test1 - Example Rexx Program -   Rexx EXEC */
Do I = 1 to 20
    Say "Error "I" is "errortext(I)
End
```

The above Rexx exec will display the following output:

```
Error 1 is
Error 2 is
Error 3 is Program is unreadable
Error 4 is Program interrupted
Error 5 is Machine storage exhausted
Error 6 is Unmatched "/*" or quote
Error 7 is WHEN or OTHERWISE expected
Error 8 is Unexpected THEN or ELSE
Error 9 is Unexpected WHEN or OTHERWISE
Error 10 is Unexpected or unmatched END
Error 11 is Control stack full
Error 12 is Clause too long
Error 13 is Invalid character in program
Error 14 is Incomplete DO/SELECT/IF
Error 15 is Invalid hexadecimal or binary string
Error 16 is Label not found
Error 17 is Unexpected PROCEDURE
Error 18 is THEN expected
Error 19 is String or symbol expected
Error 20 is Symbol expected
```

ExecIO

Purpose: Perform input/output operations.

Type: TSO Command

Syntax: "EXECIO *quantity operation ddname seq* (*options*"

where

quantity represents the number of records to read or write

operation DiskR for "read from disk"
DiskW for "write to disk"
DiskRU for "read for update"

ddname The ddname of the file for which I/O is to be performed.
The file must be allocated by TSO prior to its use.

seq Sequence number of the desired record, for disk read operations only

options STEM *stem.* FINIS
STEM is specified when reading records from or writing records to an array. s*tem* is the "name" of the array. If STEM is not specified, operations are performed on a disk file instead of an array.
Specify FINIS to close a disk file when done processing

Usage: If you perform a disk read operation, and you reach end-of-file, RC will be set to a 2.

Examples:

Read	Write	Ex #	Comments
Disk	TSO Stack	1	Read a disk file into the TSO stack
Disk	Array	2	Read a disk file into an array
TSO Stack	Array	3	Read the TSO stack into an array
TSO Stack	Disk	4	Read the TSO stack, write a disk file
Array	Disk	5	Read an array, write a disk file
Array	TSO Stack	6	Read an array into the TSO stack
Disk	Disk	N/A	Read one disk file; write another: 1) Read disk file into array 2) Write array to disk file
Disk	Disk	7	Copy a disk file, one record at a time
Disk	Disk	8	Disk update (update a record in place)
Array	Array	9	Copy one array to another
TSO Stack	TSO Stack	N/A	(Only one TSO stack is available)

Example 1: Read a disk file into the TSO stack

```
"Alloc fi(DDIn) Da(user.work) shr"
NewStack
"ExecIO * DiskR DDIn (Finis "
"Free Fi(DDIn)"
```

And then, to process the stack:

```
Do while queued() > 0
    Pull OneLine
    Say OneLine
End
```

Caution: If you read information into the stack, and then leave it there, whether intentionally or by an error in your Rexx exec, TSO will try to execute it.

Example 2: Read a disk file into an array

```
"Alloc fi(DDIn) Da(Rexx.exec(TestData)) shr"
"ExecIO * DiskR DDIn (Stem Lines. Finis "
"Free Fi(DDIn)"
```

And then, to process the array:

```
Say "The disk file contains " Lines.0 "lines. Here they are:"
Do I = 1 to Lines.0
    Say Lines.I
End
```

Example 3: Read the TSO stack into an array

```
/* If the queue is empty, say so and get out */
If queued() < 1 then do
    say "The TSO stack is empty!"
    Exit 16
End
/* Now read the stack into an array */
Lines.0 = queued()
Do I = 1 to queued()
    Pull NewLine
    Lines.I = NewLine
End
```

And then, to process the array:

```
Do I = 1 to Lines.0
    Say Lines.I
End
```

Example 4: Write the TSO stack to disk

```
If queued() > 0 then do
  "Alloc Fi(DDOut) da(work.data(test2)) shr"
  "ExecIO * DiskW DDOut (Finis "
  "Free Fi(DDOut)"
End
Else
  Say "The queue was empty; no file written!"
```

Example 5: Write an array to disk

```
"Alloc Fi(DDOut) da(work.data(test3)) shr"
"ExecIO * DiskW DDOut (Stem Recds. Finis "
"Free Fi(DDOut)"
```

Example 6: Read an array into the TSO stack

```
"Alloc Fi(DDIn) da(work.data(test1)) shr"
"ExecIO * DiskR DDIn  (Finis "
"Free Fi(DDIn)"
Say "I read "queued()" records into the TSO stack"
DelStack       /* Delete this stack when done */
```

Example 7: Copy a disk file, one record at a time

```
"Alloc Fi(DDIn)  da(work.data(test1)) shr"
"Alloc Fi(DDOut) da(work.data(test6)) shr"
RecsCopied = 0
Do Forever
    "ExecIO 1 DiskR DDIn"            /* Read a disk record*/
    If RC = 0 then do                /* Not end of file   */
        "ExecIO 1 DiskW DDOut"       /* Write a disk rec  */
        RecsCopied = RecsCopied + 1  /* Count the records copied */
    End
    Else Do                          /* End of file       */
        "ExecIO 0 DiskR DDIn  (Finis" /* Close the input file    */
        "ExecIO 0 DiskW DDOut (Finis" /* Close the output file   */
        Leave
    End
End
"Free Fi(DDIn,DDOut)"
Say "I copied "RecsCopied" records"
```

Example 8: Disk Update (update a record in place)

This is accomplished by reading a disk record (for update) into the stack, removing it from the stack into a variable, modifying it (in the variable), putting it back into the stack, and then writing the record back to disk, from the stack.

```
"Alloc Fi(DDUp)  da(work.data(test1)) OLD"
"NewStack"                           /* Establish a new stack    */
RecsUpdated = 0
"ExecIO 1 DiskRU DDUp 4 "            /* Read record number 4     */
Pull Record                          /* Read stack               */
Say "Record number 4 is" Record
Record = left(Record,10)||"* this asterisk is in column 11"
Say "The record was changed to:" Record
Push Record                          /* Put it back into the stack*/
"ExecIO 1 DiskW  DDUp   "            /* Write the record back    */
RecsUpdated = RecsUpdated + 1
"ExecIO 0 DiskW  DDUp (Finis"        /* Close the I/O file       */
"Free Fi(DDUp)"
"DelStack"                           /* Delete the new stack     */
Say "I updated "RecsUpdated" records"
```

Example 9: Copy one array to another

```
"Alloc Fi(DDin)  da(work.data(test1)) SHR"
RecsCopied = 0
"ExecIO * DiskR  DDin (stem Recs. Finis)" /* Read the disk into array */
Say "There are "Recs.0" records in the Recs array"
```

```
"Free fi(DDIn)"
Do I = 1 to Recs.0
    Recs2.I = Recs.I
End
Recs2.0 = Recs.0
Say "There are "Recs2.0" records in the Recs2 array"
Do I = 1 to Recs2.0
    Say Recs2.I
End
```

Exit

Purpose: Terminate a Rexx exec, and optionally set a return code.

Type: Rexx Instruction

Syntax: Exit *ReturnCode*
where *ReturnCode* is any code you wish to set.

Usage: Typically, the Exit instruction is coded at the end of a Rexx exec's processing, but it can indeed be used to prematurely terminate a Rexx exec. *ReturnCode* is the MVS return code, and can be tested by a calling program (another Rexx exec, for example), or by JCL.

To check for charcter strings instead of words, use **Index.**

Example:

```
Exit 16     /* Tell the caller I failed */
Exit 0      /* Tell the caller I processed ok */
```

Expose

Purpose: Make a local variable available to an external routine

Syntax: PROCEDURE EXPOSE variable

Usage: Typically, when an exec calls a procedure, it passes to the procedure all of the necessary values. The procedure, by rules of good coding, hides all of its local variables (by using the "Procedure" statement. If the procedure wants to pass one of those variables back, it can simply "Expose" the variable.

The Expose, used from inside a procedure, makes variables defined outside the procedure available to it.

Example: This is an example of a program that will calculate a bowling average for a five-game tournament.

```
/* Test1 - Example Rexx Program */
Call GA 157 202 170 160 144
Say "Your bowling average is " Result
Say "Your high game was " HighGame
Say "Your low  game was " LowGame
Exit

GA:
Procedure Expose HighGame LowGame
Arg Game1 Game2 Game3 Game4 Game5
  Total = Game1 + Game2 + Game3 + Game4 + Game5
  BowlAverage = Total / 5
  HighGame = 0
  If HighGame < Game1 then HighGame = Game1
  If HighGame < Game2 then HighGame = Game2
  If HighGame < Game3 then HighGame = Game3
  If HighGame < Game4 then HighGame = Game4
  If HighGame < Game5 then HighGame = Game5
  LowGame = 300
  If LowGame > Game1 then LowGame = Game1
  If LowGame > Game2 then LowGame = Game2
  If LowGame > Game3 then LowGame = Game3
  If LowGame > Game4 then LowGame = Game4
  If LowGame > Game5 then LowGame = Game5
Return BowlAverage
```

In the above example, the first line is a Call to procedure "GA". It passes five bowling scores.

The first thing that procedure "GA" does, is make the variables *HighGame* and *LowGame* available to the caller, by *Expos*ing them. Note that if the entire *Procedure* statement was removed, all of the variables would be available. In larger programs, that could be a problem.

See **Procedure** for more examples.

External

Purpose: Extract the number of terminal buffer or command stack elements that have been logically typed ahead by the terminal user.

See **PARSE EXTERNAL** for documentation on this subject.

Find

Purpose: Return the position of a word/words in a sentence

Type: Rexx Function

Syntax: *Result* = FIND(*sentence,words*)
where *result* is the word number where *words* appears in *sentence*. *Result* is 0 if *words* does not appear (as actual words) in *sentence*. (By definition, "word" is a character string enclosed by delimiters.)

Examples:

```
Position = FIND('Fourscore and seven years ago','years')
```
will result in 4. *years* is the fourth word in the sentence.

```
Position = FIND('Fourscoreandsevenyearsago','years')
```
will result in 0. *years* does not appear as a word in the sentence. (The sentence contains only one word.)

```
Say  FIND('Fourscore and seven years ago','and seven')
```
will result in *2*.

Form

Purpose: Returns the current setting of "Numeric Form".

Type: Rexx Function

Syntax: *CurrSetting* = Form()
where CurrSetting will contain either "SCIENTIFIC" or "ENGINEERING"

See also **Numeric Form** for documentation on this function.

Format

Purpose: To print a number

Type: Rexx Function

Syntax: *Result* = Format(*number,left-of-decimal,right-of-decimal*)
where *Result* is the formatted representation of number. *left-of-decimal* denotes how many digits to display on the left side of the decimal point, padded with blanks. *right-of-decimal* denotes how many decimal digits to display on the right side of the decimal point, zero-filled.

Usage: This function is used to display numbers so they line up with others being displayed, or to display a number in a certain way.

Example:

```
/* Test1 - Example Rexx Program -   Rexx EXEC */
Say "How much money did you have yesterday?"
Pull YAmount
Say "How much money do you have now?"
Pull NAmount
Diff = NAmount - YAmount
If Diff > 0 then DiffWord = "Gained"
else             Diffword = "*Lost*"
Diff = ABS(Diff)
Say "Yesterday, you had $" Format(YAmount,4,2)
Say "     Now, you have $" Format(NAmount,4,2)
Say "You "DiffWord"         $" Format(Diff,4,2)
```

In the above example, when the Rexx exec asked:

```
How much money did you have yesterday?
```

And you answered: `2`

And then the Rexx exec asked:

```
How much money do you have now?
```

And then you answered: `1.5`

The Rexx exec would display:

```
Yesterday, you had $    2.00
     Now, you have $   11.50
You *Lost*         $    9.50
```

Notice how the amounts line up. Without the formatting provided by the *Format* function, Rexx would display the following:

```
Yesterday, you had $2
     Now, you have $11.5
You Gained         $9.5
```

Fuzz

Purpose: Returns the current setting of "Numeric Fuzz".

Type: Rexx Function

Usage: This is an inquiry as to this setting: how many low-order digits Rexx should ignore in comparisons.

See **Numeric Fuzz** for documentation on this function.

If

Purpose: Test for certain conditions (via program expressions), allowing action to be taken based on the results of the test.

Type: Rexx Instruction

Syntax:

```
IF expression THEN
  one instruction
ELSE
  one instruction
```

```
If expression THEN DO
        one or more instructions
END
ELSE DO
        one or more instructions
END
```

Expression: Any valid program expression. If the statement is TRUE, the expression evaluates to a one. Conversely, if the expression is false, the expression evaluates to a zero. Rexx uses that value to determine whether it should execute the "THEN" instructions, or the "ELSE" instructions.

The operators that can be used in an expression follow:

>	is greater than
>=	is greater than or equal to
<	is less than
<=	is less than or equal to
/= \= <>	is not equal
=	equal: numerically equivalent; equivalent when padded with blanks
= =	**strictly** equal: exactly the same

Examples: The following statements are

```
TRUE              FALSE
1 < 2             1 > 2
2 > 1             2 < 1
3 <> 4            4 <> 4
"5b" = "5"        "5b" = = "5"      (b represents a space)
.02 = 0.02        .02 = = 0.02
```

Comparing strings: Do NOT use "If >" to compare strings. Before a compare is done, high-order blanks are removed. Therefore, the following statement

```
If " C4" < "BB3"
```

will result in *false*.

You can use the *Compare* function to compare strings, but only for equality or inequality. To compare the value of strings, convert each character with *C2D* first, as follows:

```
Do N = 1 to length(OldKey)
  If C2D(substr(OldKey,N,1)) < C2D(substr(NewKey,N,1)) then
      Leave
   If C2D(substr(OldKey,N,1)) > C2D(substr(NewKey,N,1)) then Do
     Say "The input file is out of sequence!"
     Exit
  End
End
```

If, Compound

Purpose: To allow more than one expression in an "IF" statement.

Type: Rexx Instruction

Syntax: If *expression* bo *expression* bo *expression*...
Where:
expression is as defined above
bo is a Boolean operator.

Boolean operator:

&	All expressions are true
|	At least one expression is true. (You must use the "OR" bar; you cannot use the word "OR"
&&	Only one of two expressions is true, and not both

Examples:

```
If month = "DECEMBER" | month = "JANUARY" | ,
month ="FEBRUARY" then
      season = "WINTER"

CandidateA = "Incumbent"
CandidateB = "Incumbent"
If CandidateA = "Incumbent" && ,
   CandidateB = "Incumbent" then
      Say "Input is okay"
Else
      Say "Dummy! They can't both be incumbents!"
```

The previous excerpt of code will call you a dummy, because you told the program that both candidates were incumbents.

If-Then-Do

Purpose: Execute one or more instructions one time based on some condition.

Type: Rexx Instruction

Syntax: IF *expression* THEN DO
END
where *expression* is any valid Rexx expression.

Example:
```
If A = B then do
    (one or more instructions)
End
```
In this example, `one or more instructions` is executed only if A = B.

Index

Purpose: Return the position of a character string in another

Type: Rexx Function

Syntax: *Result* = INDEX(*object*, *source*)
where *result* is the position number where *source* appears in *object*. *Result* is 0 if *source* does not appear in *object*.

Note: **Index** differs from **Pos** in that *object* and *source* are in opposite sequence in the command.

Examples:

```
Say  Index('Fourscore and seven years ago','and seven')
```
will return "11".

```
Say  Index('Fourscoreandsevenyearsago','andseven')
```
will return "10".

Insert

Purpose: Copy a string into another string.

Type: Rexx Function

Syntax: *Result-string* = INSERT(*new-string,old-string,where*)
where *result-string* is the string that will contain the *old-string* with the *new-string* inserted into it. *new-string* will be inserted into *old-string* after the *where* position. If *where* is greater than the length of *old-string*, then *old-string* wll be padded with enough blanks to accomodate the insertion operation.

Examples:

```
Say  Insert("Apple","Worm",2)
```

Will result in

```
WoApplerm
```

```
Say  Insert("Apple","Worm",7)
```

Will result in

```
Worm   Apple
```

(There are three spaces between "Worm" and "Apple").

Interpret

Purpose: To make Rexx process an expression as an instruction; that is, execute instructions that have been built dynamically.

Type: Rexx Instruction

Syntax: INTERPRET expression1 expression2 ...

Usage: This is one of those highly esoteric Rexx functions. I have never had a need for this instruction (which is not to say a person never will).

Example:

```
1 Instr = "Say"
2 Var = "Hello World"
3 Instr Var
4 Interpret Instr Var
```

Line 1 is simply setting the variable *Instr* to the character string "Say".
Line 2 is simply setting the variable *var* to the character string "Hello World".
Line 3 is being passed to TSO by Rexx, and the result is as follows:

```
COMMAND SAY NOT FOUND
     4 *-* Instr Var
      +++ RC(-3) +++
```

Line 4 tells Rexx not to pass these commands on to TSO, as it did with line 3, but to execute them instead. The result is:

```
HELLO WORLD
```

Iterate

Purpose: Pass through the remainder of the instructions in a "DO" loop without executing them.

Type: Rexx Intruction

Syntax: IF expression THEN ITERATE
where expression is any valid Rexx expression.

Usage: This is used to "skip" the remainder of a Do group.

Example:
```
/* Test1 - Example Rexx Program -   Rexx EXEC */
Say "Please tell me your name"
Pull YourName
Do I = 1 to length(Yourname)
  If I = 1 then iterate
  Say "The "I"th letter of your name is "I
End
```
The above example will print every letter of the name the user types in, *except the first*.

See the documentation on **Do** for more detailed information.

Justify

Purpose: Justify a string to both margins.

Type: Rexx Function

Syntax: *NewString* = JUSTIFY(*string*,*length*)
where *NewString* is the newly-created justified string; *string* is the character string being justified, and *length* is the length of *NewString*.

Usage: A new string is created by justifying the old string to both margins, and adding blanks between words.

If the specified length is less than the string, then the new string will be truncated on the right. Note that this should be viewed only as a side-effect, and not used purposely. Use the LEFT function instead when this is the desired effect.

If there is only one word in the string being justified, it will be justified on the left.

Examples:

The following excerpt from a Rexx exec:

```
NewString = JUSTIFY('Hello, world! I am terrific!',30)
Say NewString
```

will result in NewString containing the following:

```
Hello,  world! I  am terrific!
```

within a 30-character field.

The following excerpt from a Rexx exec:

```
NewString = JUSTIFY('Hello, world! I am terrific!',10)
Say NewString
```

will result in NewString containing the following:

```
Hello, wor
```

Notice that only 10 positions were kept.

Labels

Purpose: To provide a target for the "Signal" instruction.

Syntax: A label is immediately followed by a colon, with no intervening spaces.

Example:

```
Endit:
Exit
```

In the above example, "EndIt" is a label.

LastPos

Purpose: Return the position of the last occurrence of one string within another.

Type: Rexx Function

Syntax: *Position* = LASTPOS(*find-string,target-string*)
Where *position* is the position number of the last occurrence of *find-string* within *target-string.*

Examples: The following Rexx exec:

```
XMasGreeting = "We wish you a Merry Christmas"
Jingle = XMasGreeting || XmasGreeting || ,
XMasGreeting

Jingle = Jingle || "And a Happy New Year!"
Position = LASTPOS("Christmas",Jingle)

Say Position
```

will display "79".
XMasGreeting appears 3 times in *Jingle* (and is 29 chracters long). The last occurrence of the word "Christmas" appears in the 79th position of *Jingle.*

Leave

Purpose: Prematurely exit from a "DO" loop.

Type: Rexx Instruction

Syntax: (None)

Usage: “Do loops” can be written in a number of different ways. The example below illustrates just one of those ways. In this particular example, the only graceful way of exiting is by use of the *Leave* instruction.

Example:

```
/* Test1 - Example Rexx Program -   Rexx EXEC */
Do Forever
  Say "Tell me your name, or enter 0 to quit"
  Pull Answer
  If Answer = "0" then Leave
  Say "You told me that your name was" Answer"
End
```

The above Rexx exec will echo back whatever you type in, until you enter a zero.

Left

Purpose: Return the left "n" positions of a string.

Type: Rexx Function

Syntax: *NewString* = LEFT(*oldstring*,*quantity*)
Where *NewString* is the leftmost *quantity*th positions of *oldstring*

Example: In the following code,

```
First8 = LEFT("ABCDEFGHIJKLMN",8)
```

First8 will contain "ABCDEFGH"

See Also: Right

Length

Purpose: Return the length of a literal, string, or string variable

Type: Rexx Function

Syntax: Answer = LENGTH(variable)

Usage: This is a Rexx built-in function that will return the length of a literal, string, or string variable.

Example:

```
Answer = length("Merry Christmas and Happy New Year")
Say Answer
```

Would display

```
34
```

LineSize

Purpose: This is a Rexx built-in function that will return the terminal line width minus 1.

Type: Rexx Function

Syntax: *Result* = LineSize()

Usage: This is an inquiry-only function, and will usually return "79".

ListDSI

Purpose: Retrieve information about a TSO dataset.

Type: TSO external function

Syntax:

LISTDSI(*datasetname diropt*)
or
LISTDSI(*filename type diropt*)

datasetname- the name of the data set about which you want information.
diropt- an option that indicates whether or not you want PDS directory information returned.

DIRECTORY - return directory information. Note that this option must be specified if you want the PDS-specific variables below to contain the desired information (SYSADirBlk, for example).
NODIRECTORY - Do not return directory information. This is the default.

filename is the DD name if you pre-allocated the file

type Specify 'FILE' if the first operand is a DDName instead of a datasetname

Usage: This function will retrieve information about a dataset, and put it into variables.

The function is said to succeed if it can access the desired dataset information, and fail if it cannot. The function in reality does not fail, however, because if the dataset cannot be allocated, LISTDSI sets three variables that say why.

If the function succeeds, the return code is set to zero, and certain variables are set:

SYSADirBlk	For a PDS, this value will contain the number of directory blocks allocated. For a PDSE or sequential dataset, this value will be blank.
SYSALLOC	Total space allocation
SYSBLKSize	Block size of the dataset
SYSBLKSTrk	Blocks per track for the unit that this file is on
SYSCreate	Date the dataset was initially created; julian date format: yyyy/ddd
SYSDSName	Fully-qualified datasetname
SYSDSorg	DSORG of the dataset
SYSExDate	Expiration date of dataset. 0, if there is none.
SYSExtents	Number of extents used
SYSKEYLEN	Key length. 0 for non-keyed datasets
SYSLRECL	Logical record length

SYSMembers	Number of members in the PDS. This value is blank for PDSE's.
SYSPassword	The password assigned to the dataset, or "NONE"
SYSPrimary	Primary space allocation quantity
SYSRACFA	Level of RACF protection. Possible values are "NONE", "GENERIC", and "DISCRETE"
SYSRECFM	Record format of dataset
SYSRefDate	Date the dataset was last referenced; julian date format: yyyy/ddd
SYSSeconds	Secondary space allocation
SYSTrksCyl	The number of tracks per cylinder on the volume on which this dataset resides SYSUDirBlk. For a PDS, this value will contain the number of directory blocks used
SYSUnit	Generic unit of the volume, such as "3390"
SYSUnits	Units of allocation: "TRACK", "BLOCK", "CYLINDER", etc
SYSUpdated	Whether the dataset was ever updated: "YES" or "NO"
SYSUSED	Current space utilization: quantity of "SYSUnits" above. "N/A" for PDSE's
SYSVolume	The volume serial number on which this dataset resides

If the LISTDSI function fails, the return code is set to 16, and certain other variables are set:

SYSMSGLVL1	Primary, or generic error message
SYSMSGLVL2	Specific error message
SYSReason	An error number

Examples: Consider the following Rexx Exec:

```
/* Test1 - Example Rexx Program -   Rexx EXEC */
RC = listdsi(junk.data)
If RC = 0 then do
      Say "Allocation was successful."
      Say "SYSADirBlk="SYSADirBlk
      Say "SYSALLOC="SYSALLOC
      Say "SYSBLKSIZE="SYSBLKSIZE
      Say "SYSCreate="SYSCreate
      Say "SYSDSorg="SYSDSOrg
      Say "SYSDSName="SYSDSName
      Say "SYSExtents="SYSExtents
      Say "SYSExDate="SySExDate
      Say "SYSKEYLEN="SYSKEYLEN
      Say "SYSLRECL="SYSLRECL
      Say "SYSMembers="SYSMembers
      Say "SYSPassword="SYSPassword
      Say "SYSPrimary="SYSPrimary
      Say "SYSRefDate="SYSRefDate
      Say "SYSRACFA="SYSRACFA
      Say "SYSRECFM="SYSRECFM
      Say "SYSSeconds="SYSSeconds
      Say "SYSTrksCyl="SYSTrksCyl
      Say "SYSUnit="SYSUnit
      Say "SYSUnits="SYSUnits
      Say "SYSUpdated="SYSUpdated
      Say "SYSUSED="SYSUSED
      Say "SYSVolume="SYSVolume
End
Else do
      Say "Return code = " RC
      Say "SYSReason="SYSReason
      Say "SYSMSGLVL1="SYSMsgLvl1
      Say "SYSMSGLVL2="SYSMsgLvl2
End
```

Using the above exec, I performed a LISTDSI on an existing PDS, and the Rexx exec reported as follows:

```
Allocation was successful.
SYSADirBlk=
SYSALLOC=15
SYSBLKSIZE=32720
SYSCreate=1997/104
SYSDSorg=PO
SYSDSName=DGRUND.WORK.DATA
SYSExtents=1
SYSExDate=0
SYSKEYLEN=0
SYSLRECL=80
SYSMembers=
SYSPassword=NONE
SYSPrimary=15
SYSRefDate=1997/107
SYSRACFA=GENERIC
SYSRECFM=FB
SYSSeconds=1
```

```
SYSTrksCyl=15
SYSUnit=3390
SYSUnits=TRACK
SYSUpdated=YES
SYSUSED=N/A
SYSVolume=PCF011
```

Using the same exec, I performed a LISTDSI on an non-existent PDS, and the Rexx exec reported as follows:

```
Return code =  16
SYSReason=0005

SYSMSGLVL1=IKJ58400I LISTDSI FAILED.  SEE REASON CODE IN VARIABLE SYSREASON.

SYSMSGLVL2=IKJ58405I DATA SET NOT CATALOGUED. THE LOCATE MACRO RETURN CODE IS
0008
```

Literals

Purpose: Literals exist so variables can represent an unchanging value.
Rexx supports literals of a number of different types.

Usage: Typically, a literal is one that is enclosed by either a set of quotation marks or apostrophes.
"HELLO WORLD" and
'HELLO WORLD' represent the same character string.

Literals can be numeric, character, hexadecimal, and binary. "FOUR" is a character literal; "4" is a numeric literal.

The reason I say "typically", is because that is not always the case. (This is probably one of my biggest complaints about Rexx. I feel that if it was more stringent, it would be easier to figure out and explain.)

- A character literal doesn't *have to* be enclosed. If it isn't, it is changed to all upper case.
- A character literal that is not enclosed can be converted to a variable by using it on the left side in an assignment statement. (You will get a syntax error if you try to assign a literal that is enclosed).

Consider this example. The following excerpt is from a Rexx Exec:

```
Say "Hello, World"      /* Character string */
Say Hello  World   /* Two literals     */
World = "Dave"     /* Make "World" a variable */
Say Hello World
```

will yield the following results:

```
Hello, World
HELLO WORLD
HELLO Dave
```

A good rule of thumb to follow is always enclose literals. That way, if a character string appears in your output, you can bet it's an (uninitialized) unused variable.

Logical Operators

REXX supports the following logical (Boolean) comparison operators:

& AND - returns a 1 (true) if both comparisons are true, and a 0 (false) otherwise - performs a logical AND operation

| OR - returns a 1 (true) if at least one comparison of several is true, and a 0 (false) otherwise - performs a logical or operation

&& EXCLUSIVE OR - returns a 1 (true) if ONLY one of a group of comparisons is true, and a 0 (false) otherwise - performs a logical exclusive OR function

\ NOT - returns the reverse logical value for an expression - returns false if expression resolves to true, and true if the rexpression resolves to false

Math

Rexx performs math whenever it can recognize arithmetic operators. The valid Rexx operators are as follows:

Operator	Function
+	Add
-	Subtract
*	Multiply
/	Divide
%	integer divide
//	remainder of division
**	Exponentiation
( )	group items

Usage

The primary operations (+, -, *, /) are obvious, so not much further discussion is needed here.

%- Integer Divide — Any remainder is dropped

//- Remainder of Division — Yields the remainder in a division expression.

The following excerpt from a Rexx exec:

```
"EXECIO" 1 "DiskW SYSUT2"
OpCount = OpCount + 1
If OpCount // 1000 = 0 then
    Say OpCount "records written so far..."
```

will print a message line for every 1000th record written. This, of course, is useful in a long-running program.

**- Exponentation

( ) — Operations within expressions to make them take precedence over normal precedence.

Max

Function: Return the highest of a series of numbers.

Type: Rexx Function

Syntax: HighNum = MAX(num1, num2...)

Usage: A maximum of 20 numbers can be provided.

Example:

```
Say Max(1,3,5,17,9,6,4)
```

will yield "17" (without the quotation marks)

Min

Purpose: Return the lowest of a series of numbers.

Type: Rexx Function

Syntax: LowNum = MIN(num1, num2...)

Usage: A maximum of 20 numbers can be provided.

Example:

```
Say Min(8,3,5,17,9,6,4)
```

will yield "3" (without the quotation marks)

Msg

Purpose: Change or inquire as to the current TSO "MSG" setting.

Type: TSO external function

Syntax: Setting = MSG(*on/off*)
where *setting* is the current setting, *before* it is changed by what is in the parentheses;
on/off is either "ON", "OFF", or nothing.

Usage: The TSO "MSG" setting indicates whether TSO messages are printed during the execution of a Rexx exec or not. Specify the command with "ON" to turn message displays on, "OFF" to turn message displays off, and a null parameter (just the parentheses with nothing in them) to display the current setting.

Example 1: In the following example,

```
1 Say Msg()
2 MSetting = Msg(Off)
3 Say MSetting
4 Say Msg()
```

1 will display the current TSO message setting, either "ON" or "OFF"
2 will capture the current TSO MSG setting into the variable *MSetting*, and then set the setting to "OFF", regardless of what it was
3 will display the variable *MSetting*
4 will display the new current setting, which will be "OFF"

Example 2: In the following example,

```
1 Say Msg()
2 Say "About to allocate the first time..."
3 "Allocate FI(dummy) DA(junk2.data) shr"
4 Junk = Msg(Off)
5 Say "About to allocate the second time..."
6 "Allocate FI(dummy) DA(junk2.data) shr"
```

1 will display the current TSO message setting, either "ON" or "OFF"
2 tells the user that we are about to issue a TSO command
3 allocates the file, if possible
4 turns the MSG setting off
5 tells the user again that we are about to issue a TSO command
6 allocates the file again, if possible
The dataset *junk2.data* does not exist, so each attempt at allocating it will fail. Line 3 above will issue a message because the TSO MSG setting is on. Line 6 above would have issued a TSO message, but the TSO MSG setting was off.

NewStack

Purpose: Establish a new TSO stack

Type: TSO command

Syntax: NewStack

Usage: To tell Rexx that from here on, all stack operations are to be conducted on a newly-established TSO stack, instead of the one that existed when the instruction started. The "old" stack is left alone and unharmed by further operation, until a DelStack is issued to discard this newly-established stack.

See also: DelStack

NOP

Purpose: No operation

Type: Rexx instruction

Syntax: NOP

Usage: Allow you to use an instruction that performs no action in a place where an instruction (of any kind) is required.

Example: The following example is coded this way to avoid complicated negative logic.

```
If A = 1 | A = 2 then
      Nop               /* do nothing */
Else
      Say "answer was incorrect"
```

Numeric

Purpose: Set certain rules for Rexx's handling of numbers. It controls the waya Rexx exec carries out arithmetic operations.

Type: Rexx instruction

Syntax: Numeric *function*
Where *function* is either Digits, Form, or Fuzz.

Digits controls the precision to which arithmetic operations are evaluated.

Form directs which form of exponential notation Rexx uses of the result of arithmetic operations

Fuzz controls how many digits, at full precision, are ignored during a numeric comparison operation.

In many cases, these three functions work together to produce the desired results.

Numeric Digits

Purpose: Controls the precision to which arithmetic operations are evaluated.

Syntax: Numeric Digits *NoOfDigits*
NoOfDigits - Defaults to 9, and must be larger than the current NUMERIC FUZZ setting. There is no practical limit to the value for DIGITS, but keep in mind that higher values result in added processing time.

Example: The following Rexx exec snippet:

```
Numeric Digits 5 ; Say 1234.56 * 1
Numeric Digits 4 ; Say 1234.56 * 1
Numeric Digits 3 ; Say 1234.56 * 1
Numeric Digits 2 ; Say 1234.56 * 1
Numeric Digits 1 ; Say 1234.56 * 1
```

Will produce:

```
1234.6
1235
1.23E+3
1.2E+3
1E+3
```

Numeric Form

Purpose: Directs which form of exponential notation Rexx uses for the result of arithmetic operations

Syntax: Numeric Form *mode*
Where *mode* is either SCIENTIFIC or ENGINEERING
SCIENTIFIC notation adjusts the power of ten so there is a single non-zero digit to the left of the decimal point.
ENGINEERING notation causes the power of ten to be expressed as a multiple of 3.

Example: The following Rexx exec snippet:

```
Numeric Digits 2
Numeric Form Scientific
Say 123.45 * 1
Numeric Form Engineering
Say 123.45 * 1
```

Will produce:

```
1.2E+2
120
```

Numeric Fuzz

Purpose: Controls how many (low-order) digits, at full precision, are ignored during a numeric comparison operation. The *exact* way this function works is actually slightly complicated.

Syntax: Numeric Fuzz *ToIgnore*

ToIgnore - Defaults to 0. It must be smaller than the current setting of NUMERIC DIGITS.

Usage: During the numeric comparison, the numbers are subtracted under a precision of DIGITS minus FUZZ digits, and the difference is then compared to 0.

Example 1: The following Rexx exec snippet:

```
Value1 = 133456
Value2 = 123457

Numeric Digits 6
Numeric Fuzz 5
If Value1 = Value2 then Say "They are equal"
Else               Say "They are NOT equal"
```

Will produce:
They are equal
Digits (6) minus Fuzz (5) equals 1. That is the number of digits from the left that are compared. Since the first digit in each of Value1 and Value2 are identical, this comparison is true.

Example 2: The following Rexx exec snippet:

```
Value1 = 133456
Value2 = 123457

Numeric Digits 6
Numeric Fuzz 5
If Value1 = Value2 then Say "They are equal"
Else               Say "They are NOT equal"

Numeric Fuzz 4
If Value1 = Value2 then Say "They are equal"
Else               Say "They are NOT equal"
```

Will produce:
They are equal
Digits (6) minus Fuzz (4) equals 2. That is the number of digits from the left that are compared. The first two digits of Value1 (13) are compared to the first two digits of Value2 (12). This comparison is obviously false.

Operators

Arithmetic Operators- See the subject entitled "Math"
Comparison Operators- See the subject entitled "Compare"
Logical Operators- See the subject entitled "Logical Operators"
Concatenation Operators- See the subject entitled "Concatenation"

REXX Operator Precedence
The following list shows order of precedence for ALL REXX operators:

1) Expressions in parenthesis are evaluated first
2) prefix operators ==> -, + \
3) exponentiation ==> **
4) Multiplication and division in this order ==> *,/,%,//
5) Addition and Subtraction ==> + and -
6) concatenation ==> || or blank
7) comparison operators ==> ==,=,\==,\=,>,<,><,>=,<=,\<,\>
8) logical AND ==> &
9) logical OR and EXCLUSIVE OR ==> |, &&

OutTrap

Purpose: To turn on or off the capturing of TSO output.

Type: TSO external function

Syntax: *ReturnCode* = OUTTRAP(*stem.*,*max*)
ReturnCode = OUTTRAP('ON'/OFF')
where *stem.* is the name of the array into which the TSO output will be built, and *max* is the maximum number of records that will be written. Note that *stem* must end in a period. *ReturnCode* will be 0 if the function succeeds.

Usage: OUTTRAP("ON"): Turn on capturing of TSO messages and output, simply "swallow" it. Nothing wil be displayed at the terminal.
OUTTRAP("OFF"): Stop the capture of TSO messages and output, in which case they will start being displayed at the terminal again.
OUTTRAP(*stem.*,*max*): Turn on capturing of TSO messages and output, and write it all to an array named *stem*. *max* is the maximum number of records that will be written. Specify "*" to process all records, although that is the default.

Example: In the following example, we are trying to write all of the member names of a PDS to an array. As a byproduct of the TSO command that we are using, some unwanted information is written to the array as well.

```
Dummy = OutTrap("output_line.","*")
"LISTd work.data m"
NumLines = OutPut_Line.0
Say NumLines "lines were created"
Dummy = OutTrap("OFF")
Do I = 0 to NumLines
    Say "Output_Line."I"="Output_Line.I
End
```

After execution of this exec, the array called Output_Line looks like this:

```
Output_Line.0=8
Output_Line.1=DGRUND.WORK.DATA
Output_Line.2=--RECFM-LRECL-BLKSIZE-DSORG
Output_Line.3=  FB    80    32720   PO
Output_Line.4=--VOLUMES--
Output_Line.5=  PCF011
Output_Line.6=--MEMBERS--
Output_Line.7=  PROG01
Output_Line.8=  PROG02
```

There are only two members in the PDS, but the array contains all of the other output from the ListDS command. It's really simply to process around it, though, like this:

```
Do I = 7 to Output_Line.0
```

Overlay

Purpose: Move characters over (on top of) other characters.

Type: Rexx Function

Syntax: NewString = OVERLAY(source,object,position)

Usage: This function replaces the characters in object with the characters in source, starting at position. If object is less than position, it is padded with blanks.

Example 1: This is what happens when you use the command the wrong way:

```
NewString = OVERLAY("ABCDEFGHIJK",'X',4)
Say Newstring
```

NewString will contain:

```
X  ABCDEFGHIJK
```

Example 2: The following example decides, based on the day of the week, whose turn it is to make the coffee.

```
Say "Today is " Date(W)
CoffeeMaker = "Undecided"    /* default */
If Date(W) = "Monday" then CoffeeMaker = "Glenda"
If Date(W) = "Tuesday" then CoffeeMaker = "Alice"
If Date(W) = "Wednesday" then CoffeeMaker = "Thom"
If Date(W) = "Thursday" then CoffeeMaker =      ,
"Brucey"
If Date(W) = "Friday" then CoffeeMaker = "Chuck"
Message = "The person in charge of making coffee
-> today is"
Position = length(Message) + 2
Say OVERLAY(Coffeemaker,Message,Position)
```

Parse

Purpose: Take data from one of several origins, optionally break it up, and then drop it into variables.

Type: Rexx Instruction

Syntax: PARSE [UPPER] origin varname delimiter varname delimiter...

UPPER- Converts the data to upper case. This is the default.

origin- Places where REXX can get the data from:

- ARG- Command line
- VAR- A variable
- PULL- The TSO stack
- SOURCE- TSO info on how the program was executed
- VALUE- Literal
- EXTERNAL- Terminal
- VERSION- Version of Rexx interpreter

varname- One or more variables

delimiter- Delimiters for parsing the origin data

Note: The words "Parse Upper" are optional. When Rexx sees any of these origins, it assumes "Parse Upper".

Upper: "Upper" is optional, but it is the default. To *not* take the default, simply specify "Parse" without the word "Upper".

Action: Rexx will move variables one at a time from the implied origin into the variables specified after the origin keyword.

If there are more origin parameters than variables, Rexx will put all of the remaining parameters into the last variable. The last variable can be a period, in which case extra origin parameters will simply be discarded. I don't recommend this, however. Letting these drop into a variable would not hurt. You can always choose to ignore them, but the program will require no modification here if you later choose to look at these parameters.

If there are more variables than there are origin parameters, the variables are set to spaces.

Delimiters break the input up and cause it to be processed separately, under the guidelines specified above.

ARG: Take input from the command line. This is information that the user supplied to the exec when entering the command. See examples EX01 and EX02 below.

VAR: Take input from a variable. See example EX03 below.

A period is used as a placeholder. If you don't wish to use all of the arguments that are supplied to an EXEC, you can specify a period instead of a variable name, and that argument will be ignored.

PULL: Take input from the TSO stack. Use PULL to prompt the user for information. (Whatever the user types in is moved into the TSO stack.) See example EX04 below.

SOURCE: Take input from information that the system (TSO) maintains about your REXX program. It returns nine values. They are:
1. Operating System. In this case, it would be TSO.
2. How the program (Rexx exec) was called. It will be either of COMMAND, SUBROUTINE, or FUNCTION.
3. Name of the EXEC
4. DDName of command library; either SYSEXEC or SYSPROC
5. Datasetname containing the EXEC. It will be '?' if the command was invoked implicitly.
6. The name that the command was invoked by. It will be "?" if the command was invoked implicitly.
7. The initial address environment; generally TSO, MVS, or ISPEXEC
8. Environment: TSO, MVS, or ISPF
9. Reserved. Will be '?'
See example EX05 below.

VALUE: Take input from a literal. This function can be used to parse things like the current time. See example EX06 below.

EXTERNAL: Take input from the terminal.

varname: One or more variable names

Delimiters: Delimiters to determine where origin data is divided. These delimiters can be literals, variables, or column numbers.

Literal
Delimiters: Break input up at a specific character. See example EX03 below.

Variable
Delimiters: Break input up at a specific variable

Column number
Delimiters: Break input up under the control of column numbers.

Examples: I used the origins that I did for the sake of clear explanation only. The following examples apply to all of the origins.

In the following Rexx program,

```
/* EX01 - REXX Example Program */
Parse Upper Arg Var1 Var2 Var3 Var4 Var5
Say Var1 ; Say Var2  ; Say Var3  ; Say Var4  ; Say Var5
```

If the command line read

```
Ex01 a b c d e
```

Rexx would display

```
A
B
C
D
E
```

If the command line read

```
Test1 a b c d e f g h
```

Rexx would display

```
A
B
C
D
E F G H
```

If the command line read

```
Test1 a b
```

Rexx would display

```
A
B
```

(with three blank lines following)

If the command line read

```
Test1 "My name is Dave"
```

Rexx would display

```
"MY
NAME
IS
DAVE"
```

(blank line)

In the following Rexx program,

```
/* Ex02 - REXX Test Program */
Parse Arg Var1 Var2 Var3 Var4 Var5
Say Var1 ; Say Var2  ; Say Var3  ; Say Var4  ; Say Var5
```

If the command line read

```
EX02 a b c d e
```

Rexx would display

```
a
b
c
d
```

e

In the following Rexx program,

```
/* EX03 - REXX Test Program */
Parse upper arg datasetname
Parse var datasetname PDSName "(" MemName ")" junk
Say "The command line parameter was " DatasetName
Say "The PDSName is                  " PDSName
Say "The MemberName is               " MemName
Say "The junk variable is            " junk
```

If the command line read

```
EX03  user.session.jcl(copyfile)
```

Rexx would display

```
The command line parameter was  USER.SESSION.JCL(COPYFILE)
The PDSName is                  USER.SESSION.JCL
The MemberName is               COPYFILE
```

In the following Rexx program,

```
/* EX04 - REXX Example Program */
Newstack
Say "Please tell me your first and last name"
Pull FirstName LastName
Say "You told me your first name was" FirstName
Say "You told me your last  name was" LastName
DelStack
```

If the command line read

```
      EX04
```

Rexx would display

```
      Please tell me your first and last name
```

And if you replied

```
      George Washington
```

Rexx would display

```
      You told me your first name was  GEORGE
      You told me your last  name was  WASHINGTON
```

In the following Rexx program,

```
/* EX05 - REXX Example Program */
Parse Upper Source Stuff
Say Stuff
```

Rexx would display something like

```
TSO COMMAND EX05 SYSEXEC ? ? TSO ISPF ?
```

In the following Rexx program,

```
/* EX06 - REXX Example Program */
Parse Value Time() with Hrs ':' Mins ':' Secs
Say Hrs; Say Mins; Say Secs
```

If the time of day was 10:28:07, Rexx would display

```
10
28
07
```

```
/* EX07 - REXX Example Program */
Parse Version Me
Say Me
```

would display something like the following:

```
REXX370 VERS 3.48 01 May 1992
```

Pos

Purpose: This is a Rexx built-in function that will allow you to determine if a character is present in a string or variable, by returning its position in the string.

Type: Rexx Function

Syntax: *Position* = POS(*source*,*object*)
where *position* is the *position* of *source* within *object*. *Position* will be zero if *source* does not apear in *object*.

Note: **Index** differs from **Pos** in that *object* and *source* are in opposite sequence in the command.

Example: We will use the following Rexx exec for our examples:

```
        /* Test1 - Check for Coffeemakers - REXX exec */
Arg Person
        CoffeeMakers = "GLENDA ALICE THOM  BRUCEY CHUCK
DAVE "
        If Pos(Person,Coffeemakers) > 0 then
            say Person "is indeed one of our CoffeeMakers"
        Else
            say Person "does not drink coffee with us"
```

The following command:

```
Test1 Alice
```

will yield the following message:

```
ALICE is indeed one of our CoffeeMakers
```

because POS contains 8

The following command:

```
Test1 Randy
```

will yield the following message:

```
RANDY does not drink coffee with us
```

because POS contains 0

The following command:

```
Test1 Al
```

will yield the following message:

```
AL is indeed one of our CoffeeMakers
```

This is an error, not in the Rexx exec, but in our usage of it. We are checking only for the existence of the character string, and not whether that charcter string is a whole word.

Procedure

Purpose: Establish that the current block of code is a Procedure, and thereby hide all local variables

Type: Rexx Instruction

Syntax: PROCEDURE

Usage: This statement is needed only when you wish to hide the variables that appear in the local block of code. You can then "unhide" some of them by using the **Expose** function.

Variables defined outside the procedure are not visible from within the procedure. They need to be passed to the procedure.

Conversely, variables defined inside a procedure are not visible from outside the procedure, unless they are “exposed”.

This is an example Rexx Exec that demonstrates some variable usage and handling.

```
/* Define the same variable outside and inside a procedure */
/* The one inside the procedure is unique.                 */
Procvarl = "This is a global variable"
Say "Before call to Procl.  ProcVarl=" ProcVarl
Call Procl
Say "After  call to Procl.  ProcVarl=" ProcVarl
Say

/* Define the same variable outside and inside a procedure, */
/* and expose that variable from within the procedure.      */
/* The one inside the procedure takes precedence.           */
Procvar2 = "This is a global variable"
Say "Before call to Proc2.  ProcVar2=" ProcVar2
Call Proc2
Say "After  call to Proc2.  ProcVar2=" ProcVar2
Say

/* Define a variable outside the procedure,  and try to use */
/* it inside the procedure.                                 */
/* The one inside the procedure does not see the one        */
/* outside the procedure. */
Procvar3 = "This is a global variable"
Say "Before call to Proc3.  ProcVar3=" ProcVar3
Call Proc3
Say "After  call to Proc2.  ProcVar3=" ProcVar3
Say
```

```
Exit

Procl: procedure
  ProcVarl = "This is a local variable"
  Say "I am in Procl.  Procvarl=" Procvarl
Return 0

Proc2:  procedure    expose ProcVar2
  ProcVar2  =  "This is a local variable"
  Say "I am in Proc2.  Procvar2=" Procvar2
Return 0
```

```
Proc3: procedure
  Say "I am in Proc3.  Procvar3=" Procvar3
Return 0
```

The output from execution of this exec:

```
Before call to Procl.  ProcVarl= This is a global variable
I am in Procl.  Procvarl= This is a local variable
After  call to Procl.  ProcVarl= This is a global variable

Before call to Proc2.  ProcVar2= This is a global variable
I am in Proc2.  Procvar2= This is a local variable
After  call to Proc2.  ProcVar2= This is a local variable

Before call to Proc3.  ProcVar3= This is a global variable
  I am in Proc3. Procvar3= PROCVAR3
After  call to Proc3.  ProcVar3= This is a global variable
```

Prompt

Purpose: Change the setting of, or inquire as to the current setting of the TSO "Prompt" setting.

Type: TSO external function

Syntax: Answer = PROMPT("ON"|"OFF"|)

Usage: Rexx PROMPT functions only if the TSO PROFILE PROMPT setting is "ON")as opposed to "PROFILE NOPROMPT").

The "ON" parameter will cause Rexx to allow TSO commands to prompt for necessary information.

The "OFF" parameter will force TSO commands to bypass the normal step of stopping and asking for missing information.

In both of the above cases, the function will first return the current setting.

The empty parameter will simply return the current setting.

Example: The following exec is actually the same process run twice; once after turning the TSO Profile Prompt setting ON, and once turning it off. During each process, we will turn the Rexx Prompt setting on, issue the TSO "Delete" command, and then turn the Rexx Prompt setting off, and then issue the same TSO delete command. If you get confused, just remember that there is a difference bewteen the TSO Prompt command and the Rexx Prompt function.

```
"Profile Prompt"
Say "Here is the demo with the TSO prompt ON"
Dummy = prompt("ON")
Say "Rexx Prompt is " prompt()
"Newstack"
Delete
"Delstack"

Dummy = prompt("OFF")
Say "Rexx Prompt is " prompt()
"Newstack"
Delete
"Delstack"

"Profile NoPrompt"
Say "Here is the demo with the TSO prompt OFF"
Dummy = prompt("ON")
Say "Rexx Prompt is " prompt()
"Newstack"
Delete
"Delstack"
```

```
Dummy = prompt("OFF")
Say "Rexx Prompt is " prompt()
"Newstack"
Delete
"Delstack"
```

This exec will display:

```
Here is the demo with the TSO prompt ON
Rexx Prompt is  ON
ENTER ENTRY NAME -
```

At which point, the command waits for a datasetname to be entered. I entered "A".

Continuing the display...

```
ERROR QUALIFYING GRUND.A
** DEFAULT SERVICE ROUTINE ERROR CODE 20, LOCATE
ERROR CODE 8
LASTCC=8
Rexx Prompt is  OFF
MISSING ENTRY NAME+
LASTCC=12
MISSING ENTRYNAME TO BE DELETED, PASSWORD OPTIONAL
Here is the demo with the TSO prompt OFF
Rexx Prompt is  ON
MISSING ENTRY NAME+
LASTCC=12
MISSING ENTRYNAME TO BE DELETED, PASSWORD OPTIONAL
Rexx Prompt is  OFF
MISSING ENTRY NAME+
LASTCC=12
MISSING ENTRYNAME TO BE DELETED, PASSWORD OPTIONAL
```

In the above exec, I tried the following 4 scenarios:

TSO PROMPT	REXX PROMPT	Prompting Occurred?
ON	ON	YES
ON	OFF	NO
OFF	ON	NO
OFF	OFF	NO

In each case where prompting did not occur, TSO went along its merry way, trying to delete a dataset whose name wasn't supplied. Naturally, it failed.

Pull

Purpose: Get input from TSO

Type: Rexx Instruction

Syntax: Pull variable1 variable2...

Usage: This command will first look at the TSO stack. If the TSO stack is empty, the command will prompt the user.

Example:

```
1 NewStack
2 Push "Hello #1"
3 Pull Answer1
4 Say "I just learned" Answer1
5 Pull Answer2
6 Say "I just learned" Answer2
```

In this example,
1 Establishes a new stack
2 Puts the phrase "Hello #1" onto the stack
3 Gets (and removes) that phrase from the stack
4 Displays `I just learned HELLO #1`
5 Prompts the user for more input, since the stack is now empty
6 Displays whatever the user just typed in.

See **Parse** and **Stack** for documentation on this function.

Push

Purpose: Move data to the TSO stack.

Type: Rexx Instruction

Syntax: PUSH variable1 variable2 ...

Usage: Put things in the "input queue". This instruction works in LIFO format: last in, first out. It operates like a pile of plates in a diner. The plates put on top push the others down, and the first ones pulled off are the last ones put on.
Queue does the same thing as Push, but in FIFO format.

Example 1:

```
1 NewStack
2 Say "I have "queued()" lines on the stack"
3 Push "A" "B" "C"
4 Say "I have "queued()" lines on the stack"
5 Pull var1
6 Say "I pulled "Var1" off of the stack"
7 Say "I have "queued()" lines on the stack"
```

Line 1 established a brand new TSO stack to play with.
Line 2 tells us how many lines are on the stack. This should be "zero", since we just started a new stack.
Line 3 pushed three variables (one line) onto the stack.
Line 4 again tells us how many lines are on the stack. This should be "one".
Lne 5 pulls those three variables off the stack, so now the stack again contains zero lines.
Line 6 tells us the variables that the exec pulled off the stack
Line 7 again tells us how many lines are on the stack. This should be "zero".

Example 2:

```
Newstack
Say "I have "queued()" lines on the stack"
Push "A" "B" "C"
Push "D" "E" "F"
Say "I have "queued()" lines on the stack"
Pull var1
Say "I have "queued()" lines on the stack"
Say "I pulled "Var1" off of the stack"
Pull var1
Say "I pulled "Var1" off of the stack"
Say "I have "queued()" lines on the stack"
```

In this example, "A B C" is pushed onto the stack. Then "D E F" are pushed onto the stack. Since *Push* is a LIFO instruction, the program will first pull "D E F" off the stack, then "A B C".

QStack

Purpose: Determine the number of data stacks currently in existence

Type: TSO Command

Syntax: QStack

Usage: To see if the exec (or subroutines) had created any data stacks

See also: NewStack, DelStack

Example: The following Rexx exec snippet:

```
"QStack"              /* Returns a 1 in RC */
saverc = RC           /* Save the number of stacks */
Say "The number of data stacks is " saverc
"NewStack"            /* Create a new data stack */
"NewStack"            /* Create a new data stack */
"QStack"              /* Returns a 3 in RC */
saverc = RC           /* Save the number of stacks */
Say "The number of data stacks is " saverc
```

Will display:

```
The number of data stacks is  1
The number of data stacks is  3
```

Queue

Purpose: Move data to the TSO stack.

Type: Rexx Instruction

Syntax: Queue variable1 variable2 ...

Usage: Put things in the "input queue". This instruction works in FIFO format: First in, first out.
Push does the same thing as Queue, but in LIFO format.

Example 1:

```
1 NewStack
2 Say "I have "queued()" lines on the stack"
3 Queue "A" "B" "C"
4 Say "I have "queued()" lines on the stack"
5 Pull var1
6 Say "I pulled "Var1" off of the stack"
7 Say "I have "queued()" lines on the stack"
```

Line 1 established a brand new TSO stack to play with.
Line 2 tells us how many lines are on the stack. This should be "zero", since we just started a new stack.
Line 3 pushed three variables (one line) onto the stack.
Line 4 again tells us how many lines are on the stack. This should be "one".
Lne 5 pulls those three variables off the stack, so now the stack again contains zero lines.
Line 6 tells us the variables that the exec pulled off the stack
Line 7 again tells us how many lines are on the stack. This should be "zero".

Queued

Purpose: This is a Rexx built-in function that will return the number of lines that are currently available in the TSO stack.

Type: Rexx Function

Syntax: NumOfLines = Queued()

Example:

```
If Queued() > 0 then DelStack
```

In the above example, if there are any lines on the TSO stack, we will delete them.

Quotation Marks/Apostrophes

Purpose: To enclose a literal (character string).

Syntax: " " or ' '

Usage: Literals are enclosed by a matched set of either apostrophes or quotation marks. They can be used interchangeably, but must be used in matched pairs.

A character string containing apostrophes can be enclosed by quotation marks, or vice-versa.

The Rexx instruction:	Yields:
`Say "Hello, it's me!"`	`Hello, it's me!`
`Say 'Hello, it"s me!'`	`Hello, it"s me!`

(Although the punctuation is incorrect)

A character string containing apostrophes can be enclosed by apostrophes only if each of the contained apostrophes is represented by two.

The Rexx instruction:	Yields:
`Say 'Hello, it's me!'`	`Error: unmatched quote`
`Say 'Hello, it''s me!'`	`Hello, it's me!`

The first example (enclosing apostrophes in quotation marks) is cleaner, and is the recommended method.

Enclosing an expression causes Rexx to bypass the command, and pass it right through to the environment; in our case, TSO.

Example:

```
"Say 'Hello, World' "
```

Would display

```
COMMAND SAY NOT FOUND
     8 *-* "Say 'Hello' "
       +++ RC(-3) +++
```

Random

Purpose: Return a random number

Type: Rexx Function

Syntax: *Pick* = RANDOM(*min,max,seed*)
where *pick* is the number selected; *min* and *max* is the range of numbers, inclusive, from which the function can pick; and *seed* is the random number seed; it is optional.

Usage: This function will pick a number that is commonly referred to as pseudo-random. Specifying the same seed will produce the same random number.
Random

Example: This is an example of an Exec that thinks it can guess what the current temperature is.

```
MoNum = substr(Date(U),1,2)
If Monum =  1 then Do; Low =  0; High = 55; end
If Monum =  2 then Do; Low =  0; High = 60; end
If Monum =  3 then Do; Low = 15; High = 65; end
If Monum =  4 then Do; Low = 35; High = 80; end
If Monum =  5 then Do; Low = 45; High = 85; end
If Monum =  6 then Do; Low = 50; High = 90; end
If Monum =  7 then Do; Low = 55; High = 95; end
If Monum =  8 then Do; Low = 55; High = 95; end
If Monum =  9 then Do; Low = 50; High = 90; end
If Monum = 10 then Do; Low = 30; High = 85; end
If Monum = 11 then Do; Low = 10; High = 75; end
If Monum = 12 then Do; Low =  0; High = 60; end

Temp = Random(Low,High)
Say "The temperature right now is " Temp
```

RC

Purpose: Special variable set by TSO commands

Usage: This variable can be used to test the success/failure of a TSO command.

Example 1:

```
1 Say "This is a typical Rexx instruction"
2 Say "Return Code = "RC
3 Junk
4 Say "Return Code = "RC
5 Say "Hello, World"
6 Say "Return Code = "RC
7 Say A = B + C
8 Say "Return Code = "RC
```

Line 1 will simply display a message.
Line 2 wil display Return Code = RC. Line 1 was a Rexx instruction, and did not set RC. Since RC was never set (in this exec), it is stil undefined.
Line 3 is not a Rexx instruction, so it is passed on to TSO, and the following displays:

```
COMMAND JUNK NOT FOUND
      3 *-* Junk
        +++ RC(-3) +++
```

Line 4 displays: `Return Code = -3`
Lines 5-6 display:

```
Hello, World
Return Code = -3
```

Return code was set to -3 before, and is unchanged because these are both valid Rexx instructions.
Line 7 displays:

```
7 +++ Say A = B + C
Error running T1, line 7: Bad arithmetic conversion
```

The Rexx exec stops here, so line 8 never executes.

Result

Purpose: Special TSO variable set by the Return instruction

Usage: This variable is set by the Return instruction after a subroutine is called. If the subroutine returns an expression, Result will contain that expression. If not, Result is dropped (becomes uninitialized).

Example: The following exec:

```
Call Proc1
Say "Result is " Result
Call Proc2
Say "Result is " Result
Exit
Proc1:
Return "abc"
Proc2:
Return
```

Will display:

```
Result is  abc
Result is  RESULT
```

Return

Purpose: Go back to a caller

Type: Rexx Instruction

Syntax: RETURN variable

Usage: Use this command to return to a calling program, and optionally pass a variable. The variable that is passed back will be moved into the "RESULT" variable for use by the caller.

Example 1:

```
Call Multiply 2 3
Say "The answer is "Result
Exit

Multiply:
  Arg Factor1 factor2
  Product = Factor1 * Factor2
Return Product
```

The above example illustrates the use of the *Return* function and the *Result* variable. You could have specified *Product* instead of *Result*, but that would have violated good programming techniques, and depending how the subroutine is coded, may not give you the desired results. The illustrated way always will.

Reverse

Purpose: Reverses the order of the characters of a string.

Type: Rexx Function

Syntax: Result = REVERSE(string)

Usage: Use this function to turn a string around.

Example 1: The following Rexx EXEC:

```
Message = "Happy birthday to you"
NewMsg = REVERSE(Message)
Say "The original message was " Message
Say "The new message is       " NewMsg
```

Will display:

```
The original message was  Happy birthday to you
The new message is        uoy ot yadhtrib yppaH
```

The following Rexx EXEC:

```
Message = "Able was I ere I saw Elba"
NewMsg = REVERSE(Message)
Say "The original message was " Message
Say "The new message is       " NewMsg
```

Will display:

```
The original message was  Able was I ere I saw Elba
The new message is        ablE was I ere I saw elbA
```

I used a palindrome here to illustrate a point: the case of the letters will remain the same as they were.

Right

Purpose: Return the right "n" positions of a string.

Type: Rexx Function

Syntax: *NewString* = RIGHT(*oldstring*,*quantity*)
Where *NewString* is the rightmost *quantity*th positions of *oldstring*

Example: In the following code,

```
First8 = RIGHT("ABCDEFGHIJKLMN",8)
```

First8 will contain "GHIJKLMN"

See Also: **Left**

Say

Purpose: Display strings, literals, and numeric values

Type: Rexx Instruction

Syntax: Say *anything*

Usage: This command is probably the most commonly-used Rexx command. It is used to display information to the user at the terminal. You can mix literals and variables into the object that you are displaying.

Example:

```
Say "Hello, World. My name is Computer. What is your name?"
Pull YourName
Say "So, you say your name is" YourName"."
Say "How old are you, "YourName"?"
Pull YourAge
Say "Hmmmm..." YourAge", huh? That's pretty good. I used to be",
    YourAge "once, too!"
Say "Goodbye, "YourName", and have another wonderful "YourAge" years!"
```

The above example first asks you for your name, and then your age.

Select

Purpose: Rexx's implementation of the structured programming CASE construct.

Type: Rexx Instruction

Syntax:

```
SELECT
      WHEN expression THEN instruction
      WHEN expression THEN instruction
      WHEN expression THEN instruction
      WHEN expression THEN instruction
      OTHERWISE instruction
END
```

Example:

```
SELECT
      WHEN WeekDay = 1 THEN DOWWord = "Sunday"
      WHEN WeekDay = 2 THEN DOWWord = "Monday"
      . . . /* The rest of the days of the week */
      OTHERWISE DOWWord = "Invalid"
END
```

Semi-Colon

Purpose: To stack instructions on a line

Syntax: instruction **;** instruction **;** instruction

Usage: Use this command to place more than one instruction on a line, especially when they are "short" instructions. Stacking instructions on a line can compact the body of a routine so you can see more of it at one time. Sometimes, this can be a help instead of a deterrent.

Example 1: (Instruction not stacked)

```
Temperature = Random(1,100)
If temperature < 20 then do
      Weather = "Brutal"
    Like = "heck no!"
End
If temperature > 19 & temperature < 32 then do
    Weather = "Cold"
    Like = "no"
End
If temperature > 31 & temperature < 50 then do
    Weather = "Nippy"
    Like = "not really"
End
If temperature > 49 & temperature < 71 then do
    Weather = "so-so"
    Like = "so-so"
End
If temperature > 70 & temperature < 82 then do
    Weather = "warm"
    Like = "nice"
End
If temperature > 81 then do
    Weather = "hot"
    Like = "yes!"
End
Say "The temperature now is "temperature,
    " and the weather is "Weather"."
Say "Do I like it? "Like
```

In the above example, there are two short instructions in every If-then-do group. They each take two lines.

Example 2: (Instructions stacked)

```
Temperature = Random(1,100)
If temperature < 20 then do
    Weather = "Brutal"; Like = "heck no!"
End
If temperature > 19 & temperature < 32 then do
    Weather = "Cold";   Like = "no"
End
If temperature > 31 & temperature < 50 then do
    Weather = "Nippy";   Like = "not really"
End
If temperature > 49 & temperature < 71 then do
    Weather = "so-so";   Like = "so-so"
End
If temperature > 70 & temperature < 82 then do
    Weather = "warm";   Like = "nice"
End
 temperature > 81                      then do
    Weather = "hot";   Like = "yes!"
End
Say "The temperature now is "temperature,
    " and the weather is "Weather"."
Say "Do I like this weather? "Like
```

This is the same program as Example 1, except that we stacked the instructions on one line, and we saved 6 lines in the program. That made this routine more compact, and we can therefore see more of the program on one screen. This technique, more importantly, did not compromise the appearance or readability of this code.

Sigl

Purpose: Special TSO variable that contains the line number of the last instruction that caused a jump to a label.

Usage: This variable is very useful for tracing and debugging purposes. It can tell you exactly where you came from, without having to "drop breadcrumbs".

Example: The following exec:

```
Say "Hello. I am line 3"
Say "Hello. I am line 4"
Call Proc01
Say "Hello. I am line 6"
Signal Tag01
Tag01: Say "Hello. I am line 9; Sigl="Sigl
Exit
Proc01:
Say "Hello. I am line 13; Sigl="Sigl
Return
```

Will display:

```
Hello. I am line 3
Hello. I am line 4
Hello. I am line 13; Sigl=5
Hello. I am line 6
Hello. I am line 9; Sigl=7
```

Sign

Purpose: Return the arithmetic sign of a number

Type: Rexx Function

Syntax: Result = sign(number)

Usage: This function returns a 1 if the number is positive, and a negative 1 if it is negative. It will return a zero if it is neither (a zero is considered neither positive or negative).

Example 1:

```
Number = -3
Say "The sign of this number is " sign(Number)
Number = -1
Say "The sign of this number is " sign(Number)
Number = 0
Say "The sign of this number is " sign(Number)
Number = +1
Say "The sign of this number is " sign(Number)
Number =  2
Say "The sign of this number is " sign(Number)
Number = +3
Say "The sign of this number is " sign(Number)
```

The above example yields the following displays:

```
The sign of this number is  -1
The sign of this number is  -1
The sign of this number is  0
The sign of this number is  1
The sign of this number is  1
The sign of this number is  1
```

Signal

Purpose: To unconditionally branch (transfer control) to another part of the program.

This instruction lends to "spaghetti code", and should therefore be used only when it would make the code clearer. "Bailing out" of a complicated routine is a good example.

Type: Rexx Instruction

Example:
```
Signal Endit            /* An error has occurred */

Endit:
        Say "Program ending now due to error"
Exit
```

Note: I have found the signal instruction to be unreliable in some cases. In these cases, for some reason, the signal statement simply fails to function. When this happens, the use of switches to control processing is recommended. An example follows.

```
ErrorSw = 'N'      /* Initialize the error switch */
Call Proc01       /* Perform routine 01 */
If ErrorSw = 'N' then
    Call Proc02         /* Perform routine 02 */
If ErrorSw = 'N' then
    Call Proc03         /* Perform routine 03 */
```

If an error occurred in either Proc01 or Proc02, instead of performing a "Signal" to the end of the program, you could simply set the error switch to 'Y', and then conditionally perform the rest of the program routines upon return.

Signal On

Purpose: Turn on error trapping.

Syntax: Signal On *condition*

See "Trapping Errors" in the **Environment** section of this manual for a discussion of this instruction.

SourceLine

Purpose: Return the text of the program source

Type: Rexx Function

Syntax: Result = SOURCELINE(number)

Usage: This function will return the actual program text of the line number supplied.

Example 1:

```
1 /* Test1 - Rexx Example Program */
2 Say "Hello World #1"
3 Say "Hello World #2"
4 Say "Hello World #3"
5 Say "Hello World #4"
6 Say "Hello World #5"
7 Say "Hello World #6"
8 Say "Hello World #7"
9 Say "Line three of the program is "SourceLine(3)
```

The above example will display the following:

```
Hello World #1
Hello World #2
Hello World #3
Hello World #4
Hello World #5
Hello World #6
Hello World #7
Line three of the program is Say "Hello World #2"
```

Space

Purpose: Adds blanks to or removes blanks from between words in a string.

Type: Rexx Function

Syntax: *NewString* = SPACE(*OldString*,*quantity*)
where *NewString* is the result of putting *quantity* blanks between every word in OldString.

Usage: If *quantity* is "0", this function will *remove* all blanks from the string. The function does not take into consideration how many spaces are already between words. It sets the string to the quantity you supply. Therefore, this instruction can be used to nicely format a sentence.

Example 1:

```
Greeting = "Merry  Christmas    to  one and     all"
NewGreeting = space(Greeting,0)
Say NewGreeting
NewGreeting = space(Greeting,1)
Say NewGreeting
NewGreeting = space(Greeting,2)
Say NewGreeting
NewGreeting = space(Greeting,3)
Say NewGreeting
```

This exec will display the following:

```
MerryChristmastooneandall
Merry Christmas to one and all
Merry  Christmas  to  one  and  all
Merry   Christmas   to   one   and   all
```

Stack

Purpose: Serve as an "input queue" for TSO commands in a Rexx Exec

Usage: The Stack (or TSO stack, as it is more commonly called) is a storage area used to hold TSO commands that are about to be executed. These TSO commands were moved into the stack by either an individual keying them in at the terminal, or by a Rexx program.

When a Rexx exec needs information, it first looks for it on the stack. If the stack is empty, TSO will prompt the user (see example 1).

If you wish to read TSO commands directly, and bypass the stack, use *Parse External*.

More than one TSO stack can be created. The number of TSO stacks is limited only by the core available. Only the current TSO stack, though, is the one that is the subject of operations.

The TSO stack can be shared by subroutines and by called programs.

If you read information into the stack and leave it there, then after your Rexx exec ends, TSO will try to execute each item in the stack (see example #2).

Several commands operate on or manipulate the stack:

Push	Adds items to the stack
Pull	Removes items from the stack
Queue	Adds items to the stack
NewStack	Establishes a new stack
DelStack	Deletes the current (newest) stack
ExecIO	Reads/writes a file or array into/from the stack

Each of the items above is documented in this manual in detail as their own subjects.

Example 1:

```
1 NewStack
2 Push "Hello #1"
3 Pull Answer1
4 Say "I just learned" Answer1
5 Pull Answer2
6 Say "I just learned" Answer2
```

In this example,
1 Establishes a new stack
2 Puts the phrase "Hello #1" onto the stack
3 Gets (and removes) that phrase from the stack
4 Displays `I just learned HELLO #1`
5 Prompts the user for more input, since the stack is now empty

6 Displays whatever the user just typed in.

Example 2:

```
NewStack
Push "Hello #1"
Push "Hello #2"
Push "Hello #3"
Push "Hello #4"
Push "Hello #5"
```

This example will display the following:

```
COMMAND HELLO NOT FOUND
COMMAND HELLO NOT FOUND
COMMAND HELLO NOT FOUND
COMMAND HELLO NOT FOUND
COMMAND HELLO NOT FOUND
```

Strip

Purpose: Removes leading or trailing spaces from a string.

Type: Rexx Function

Syntax: *NewString* = STRIP(*OldString*,*option*,*char*)
where *NewString* is the result of removing *char* from *OldString* based on the setting of *option*.

Usage: The function will remove from the old string:
Leading *char* (option = "L"),
Trailing *char* (Option = "T"), or
Leading and Trailing *char* (Option = "B")
The third parameter, *char*, specifies the character to be removed. If specified, it must be exactly one character long. The default is blank.

Example 1:

```
Greeting = "   Happy New Year to you           "
NewGreeting = Strip(Greeting,"L")
Say NewGreeting
NewGreeting = Strip(Greeting,"T")
Say NewGreeting
NewGreeting = Strip(Greeting,"B")
Say NewGreeting
```

This exec will display the following results:

```
Happy New Year to you
   Happy New Year to you
Happy New Year to you
```

SubCom

Purpose: Poll TSO to see if a particular environment is available.

Type: TSO command

Syntax: Subcom *environment*

Usage: This command can be used to test to see if an environment is available before issuing commands to it. For example, before you invoke the ISPF editor on a dataset, it may be a good idea to first check to see if the system has ISPF available (although this would be a good assumption).

This is the strongest reason that I could come up with for using this command, which probably demonstrates why I have never used it in any of my execs. In certain situations, there may indeed be a good reason to use it.

See also: Address

Example:

```
"SubCom TSO"
If RC = 0 then Say "TSO is available"
Else           Say "TSO is not available; RC=" RC
"SubCom ISPF"
If RC = 0 then Say "ISPF is available"
Else           Say "ISPF is not available; RC=" RC
"SubCom Junk"
If RC = 0 then Say "Junk is available"
Else           Say "Junk is not available; RC=" RC
"SubCom ISPEXEC"
If RC = 0 then Say "ISPEXEC is available"
Else           Say "ISPEXEC is not available; RC=" RC
"SubCom ISREDIT"
If RC = 0 then Say "ISREDIT is available"
Else           Say "ISREDIT is not available; RC=" RC
"SubCom CMS"
If RC = 0 then Say "CMS is available"
Else           Say "CMS is not available; RC=" RC
```

The above exec will display the following:
TSO is available
ISPF is not available; RC= 1
Junk is not available; RC= 1
ISPEXEC is available
ISREDIT is available
CMS is not available; RC= 1

SubStr

Purpose: This is a Rexx built-in function that will return a portion of a string or variable.

Type: Rexx Function

Syntax: var = SUBSTR(string,begin,length)
var Any variable name
string The object string (can be a literal also)
begin The beginning position of the string you wish to refer to
length Then length of the string you wish to refer to

Example: *Section* = substr(*alphabet*,4,5)
Where *alphabet* is a string containing all of the letters of the alphabet
After this instruction executes, the variable SECTION will contain "DEFGH"

SubWord

Purpose: Returns a subset of a sentence

Type: Rexx Function

Syntax: *NewString* = SUBWORD(*OldString*,*start*,*quantity*)
where *NewString* is the result of copying *quantity* words from *OldString*, starting at word number *start.*

Usage: Extract a fixed number of words from a sentence.

Example 1:

```
        Phrase = "Fourscore and seven years ago, our
fathers..."
        Extract = SUBWORD(Phrase,2,3)
        Say Extract
        Extract = SUBWORD(Phrase,7,3)
        Say Extract
```

This example will display the following:

```
and seven years
fathers...
```

Symbol

Purpose: Tells if a character string is a variable, literal, or neither

Type: Rexx Function

Syntax: *Result* = SYMBOL(*charstring*)

Usage: According to "the book", this function will test a character string, and return one of the following:

VAR If the character string is a valid variable name
LIT If the character string is a valid literal
BAD If neither of the above

I have found that this function will return only "LIT" or "BAD", based on whether the supplied character string can comprise a valid variable name.

Example:

```
Result = SYMBOL(Myname)
Say Result
Myname = 4
Result = SYMBOL(Myname)
Say Result
Result = SYMBOL("**")
Say Result
```

Will display:

```
LIT
LIT
BAD
```

SYSDSN

Purpose: Return the status of a datasetname

Type: TSO external function

Syntax: *Result* = SYSDSN(datasetname)

Usage: This function can tell you whether a dataset appears in the catalogue, whether a member name appears in a PDS, etc. It is not quite as comprehensive as LISTDSI.

Consult the following chart for possible results.

Result	Reason
DATASET NOT FOUND	The datasetname was not in the catalogue
ERROR PROCESSING REQUESTED DATASET	
INVALID DATASETNAME	The datasetname was invalid: Length > 44 chars, invalid chars, etc.
MEMBER NOT FOUND	Looking for a member of a PDS, but it was not found
MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED	Looking for a member of a PDS, but the dataset is not a PDS
MISSING DATASETNAME	SYSDSN( ): no datasetname supplied
OK	Disk dataset, in catalogue
PROTECTED DATASET	
UNAVAILABLE DATASET	
VOLUME NOT ON SYSTEM	Tape dataset, in catalogue

See also: LISTDSI

Example:

```
MyDSN = "'"dsn"'"
RC = SYSDSN(MyDSN)
If RC = "OK" then
  Say MyDsn "was found"
Else
  Say RC
end
```

SYSVAR

Purpose: Return information about the system

Type: TSO external function

Syntax: *Result* = SYSVAR(infoRequest)

Usage: This function can tell you the current TSO user signed on to the system, the name of the logon proc being used, and many other things.
Consult the following chart for a list.

InfoRequest	Description
SYSCPU	The number of CPU seconds used in this TSO session so far
SYSENV	Thre environment you are currently executing in: FORE for foreground; BACK for background (via JCL)
SYSHSM	This will be the HSM release number. If HSM is not available, this will be blank.
SYSICMD	The name of the command or Rexx exec
SYSISPF	*ACTIVE* if the ISPF dialogue manager is active. Test this variable in your exec if it depends on ISPF services being available.
SYSLRACF	RACF level, or spaces if not available
SYSLTerm	Number of lines available on the terminal screen.
SYSNEST	*YES* if executed from another exec or CLIST; *NO* if executed from TSO.
SYSPCmd	The most recently-executed TSO command from this exec. It will be *EXEC* if there was none.
SYSPREF	The prefix that TSO puts in front of unqualified datasetnames.
SYSPROC	The name of the procedure that was used to log on to TSO
SYSRACF	AVAILABLE, NOT AVAILABLE, or NOT INSTALLED
SYSSCmd	The most recently-executed TSO sub-command. This is "the book" explanation, but I find it to be always blank.
SYSSRV	How many SRM units were used so far
SYSTSOE	TSO/E level
SYSUID	The TSO UserID of the currently-logged on user
SYSWTerm	Number of columns available on the terminal screen. This is LINESIZE+1

Time

Purpose: This is a REXX built-in function that will provide you with the current time, in a variety of different formats.

Type: Rexx Function

Syntax: Result = Time(option)
Based on the specification of the Options below, "result" will contain the time in the corresponding format, if the current time was 1:05pm (plus a few seconds).

Option	Meaning	Format	Example
(blank)	normal (same as 'N')	hh:mm:ss	13:05:13
C	Civil	hh:mm xm	1:05pm
E	Elapsed (seconds and microseconds)	ssssssss.mmmmm	111111.222222
H	hour, 24-hour format	hh	13
L	long	hh:mm:ss.dddd	13:05:13.090191
M	Number of minutes since midnight	nnnn	785
N	normal	hh:mm:ss	13:05:13
R	Reset elapsed time		0
S	Number of seconds since midnight	nnnnn	47113

If you use an unsupported option, for example "A", you will see an error message similar to the following:

```
    5 +++ Say "The time now is " Time(A)
Error running AskTime, line 5: Incorrect call to routine
```

This command can also be used for measuring elapsed time. The first time this command is issued with either the 'E' or 'R' option, the elapsed time counter is started. Every subsequent issuance of the command with either of these options will return the elapsed time since the first issuance of Time('E') or the last issuance of Time('R'). Issuing the command with option 'R' will reset the elapsed time counter, but only *after* it returns the elapsed time.

The following example demonstrates the use of elapsed time.

```
Dummy = Time(E)             /* Start time  */
Say "I am waiting for you to hit enter!"
Pull Answer
Duration = Time('E')
Say "Point1:" Duration "seconds!"
Duration = Time('E')
Say "Point2:" Duration "seconds!"
Duration = Time('R')
Say "Point3:" Duration "seconds!"
Duration = Time('E')
Say "Point4:" Duration "seconds!"
Say "Point5:" Time('E') "seconds!"
```

This exec will display something like this:

```
I am waiting for you to hit enter!
Point1: 1.200962 seconds!
Point2: 1.203493 seconds!
Point3: 1.205070 seconds!
Point4: 0.001185 seconds!
Point5: 0.002150 seconds!
```

Trace

Purpose: List instructions as they are executed; variables as they are set

Type: Rexx Function

See “Debugging” for a discussion on this subject

Translate

Purpose: Convert characters to other characters

Type: Rexx Function

Syntax: *Result* = TRANSLATE(*ObjectString*,*String2*,*String1*)

Usage: Convert all occurrences of *ObjectString* that appear in *String1* to the corresponding character in *String2.*

Example 1: I find this a difficult command to conceptualize, to explain, or to remember, so a very detailed example is necessary here.

```
Say TRANSLATE("ABCDEFGHIJ","1234567890","DAVE")
```

Would result in:

```
2BC14FGHIJ
```

Because:

String1 = `"DAVE_____"`

String2 = `"1234567890"`

ObjectString = `"ABCDEFGHIJ"`

Result = `"2BC14FGHIJ"`

In *ObjectString*, the first character, *A*, appears in *String1*. So that *A* in *ObjectString* is replaced by *2*, which is the character in *String2* that corresponds to the character in *String1*.

The next character in ObjectString does not appear in String1, so it is not converted. The same applies to the third.

The fourth character in *ObjectString* (D), however, does appear in *String1*. So that *D* in *ObjectString* is replaced by *1*, which is the character in *String2* that corresponds to the character in *String1*.

To visualize how this command works, and how to make it work for you, just lay *String1* on top of *String2*, like I have here.

Example 2: In this scenario, it turns out that the English teacher mistakenly gave the class the wrong test: it was one grade level too high. So now, she wants to push everyone's grade up one notch, instead of making everyone re-take the test. First, let's lay out String 1 and String 2:

String1 = 'BCDF'
String2 = 'ABCD'

Then code the Rexx exec, as follows:

```
OldGrades = "BBCCBDFDDFD"
NewGrades = TRANSLATE(OldGrades,"ABCD","BCDF")
Say "The old grades were" OldGrades
Say "The new grades are " NewGrades
```

which will result in:

```
The old grades were BBCCBDFDDFD
The new grades are  AABBACDCCDC
```

Example 3: This command converts 1 to A and 2 to B

```
String = Translate(String,”AB”,”12”)
```

Trunc

Purpose: Return a number with a specified number of decimal places

Type: Rexx Function

Syntax: *NewNumber* = TRUNC(*Number*,*DecimalPlaces*)
where *NewNumber* is *Number* with *DecimalPlaces* decimal places.

Usage: This command could have been called *Decimal Places*, because that applies more than *Trunc*. The command will add or remove positions based on the specification of decimal places.

Examples:

```
Say Trunc(1.12345,0)
Say Trunc(1.12345,4)
Say Trunc(1,4)
```

Will display:

```
1
1.1234
1.0000
```

Upper

Purpose: Convert a character string to upper case
Caution: This is NOT a function. It can NOT be used on the right side of an expression.

Type: Rexx Instruction

Syntax: UPPER variable1 {variable2} {variable3}...

Examples:
```
fname='George';  lname='Bush'
Upper fname lname
Say lname ',' fname /* displays "BUSH , GEORGE" */
```

See also: Parse Upper Arg

UserID

Purpose: Return the TSO UserID of the resource who is currently logged on to the system

Type: Rexx Function

Usage: This is commonly used to determine access priviledges.

Examples:

```
Say "Your userID is" UserID()
```

Could display:

```
Your userID is DGRUND01
```

Value

Purpose: Returns the contents of a variable after resolving it. The main purpose for this function is to resolve a dynamically-created variable.

Type: Rexx Function

Syntax: NewVar = VALUE(variable)

Usage: There is a subtle difference between using VALUE(variable) and just the variable itself. Value will convert the contents of a variable to upper case while resolving it.

Simple example:

```
Name = "Dave"
Say "My name is "value(Name)
Say "My name is "Name
```

The above exec will display:

```
My name is DAVE
My name is Dave
```

Example of resolving a dynamically-created variable:

In one particular Rexx exec, I create ten arrays, named Array01, Array02, ... Array10. We wish to perform the same processing on each array, so we use a subroutine, or what is more commonly known as a procedure.

Variables

Purpose: To retain values for use later in the program. A variable can hold any type of value: character, numeric, hex, binary, etc.

Syntax: A variable must start with a character (never a number), and certain special characters. The rest of the variable can contain alphabetic characters, numbers, and certain special characters.

Some special characters that can appear in a variable name are as follows:
@ __ # $!

Some special characters that can *not* appear in a variable name are as follows:
% &

For any other special characters, you're on your own. Try it out; it can't hurt.

A variable name can be up to 250 characters long.

Usage: A variable in Rexx does not get declared. It is assigned a value by using it on the left side of an assignment statement, or with the use of certain Rexx instructions. A variable is not a variable until it is given a value. Note that until a variable is given a value, it is a literal. If a variable's value is removed (with the "Drop"), it is then converted back to a literal.

If you happen to see a variable name appear in your output unexpectedly, there is a good chance that you misspelled either it, or the one that you initialized.

Example: My_name = "John Smith"

Variables, Compound

Purpose: To act as a variable, with an added benefit. The same variable name can be used to contain any number of similar values. This is a very powerful feature of Rexx, and is very simple to implement. This feature is commonly used to construct an array.

Syntax: Same as regular variables, but with a period and a suffix added to the end.

Usage: Simply assign a value to the nth element of the array. The "0th" element is used to contain the number of elements in the array.

Example: The following excerpt from a Rexx exec:

```
Name.1 = "Mary"
Name.2 = "Joe"
Name.3 = "Alice"
Name.4 = "Smokey"
Name.0 = 5              /* Establish no. of elements */
Say "There are "name.0" elements in this array"
Do I = 1 to Name.0
    Say  Name.I
End
```

Will yield the following results:

```
Mary
Joe
Alice
Smokey
NAME.5
```

Verify

Purpose: Tells whether certain characters are contained in a given character string.
Note: It is not looking at words. It is looking at individual characters.

Type: Rexx Function

Syntax: *Result* = VERIFY(*FindString*,*ObjectString*)
Result is the first position of *FindString* that does not appear in *ObjectString*.

Usage: If *Result* is zero, then all of *FindString* appears somewhere in *ObjectString*. Both strings are case-sensitive: a lower-case letter will not match an upper-case, and vice-versa.

Example:

```
Say Verify('I','TEAM')
Say Verify('Scienc',"ConSciencious")
Say Verify('fat',"indefatigable")
Say Verify('hillary','hilarious')
```

Will display:

```
1       (There is no "I" in "TEAM")
0
0
7
```

Word

Purpose: Returns the nth word of a string.

Type: Rexx Function

Syntax: *Result* = WORD(*phrase,n*)
Result is the *n*th word of *phrase.*

Usage: If n is greater than the number of words in the phrase, result will contain blanks. If n is zero, the function will err out.

Example:

```
Say Word("Merry Christmas and Happy New Year",2)
Say Word("Merry Christmas and Happy New Year",7)
Say Word("Merry Christmas and Happy New Year",0)
```

Will display:

```
Christmas

3 +++ Say Word("Merry Christmas and Happy New Year",0)
Error running Test1, line 3: Incorrect call to routine
```

WordIndex

Purpose: Return the character position of a word in a string

Type: Rexx Function

Syntax: Position = WORDINDEX(string,n)
where *Position* is the character number of the *n*th word in *string*.

Usage: This function will return the character position where a particular word starts in a string.

Example:

```
Answer = WordIndex("Merry Christmas and Happy New Year",5)
Say Answer
```

Would display

```
27
```

The 5th word of the string is New, which starts at character position number 27.

WordLength

Purpose: Return the length of a word in a string

Type: Rexx Function

Syntax: *Answer* = WORDLENGTH(string,n)
where *Answer* is the length of the *n*th word in *string*.

Usage: This function returns the length (number of characters) of a word in a string.

Example:

```
Answer = WordLength("Merry Christmas and Happy New Year",5)
Say Answer
```

Would display

```
3
```

The 5th word of the string is "New", whose length is 3.

WordPos

Purpose: Return the position of a word or phrase in a string

Type: Rexx Function

Syntax: *Answer* = WORDPOS(phrase,string)
where *Answer* is the length of the *n*th word in *string*.

Usage: This function returns the word position of a phrase in a string.

Example:

```
        Answer = WordPos('and Happy',"Merry Christmas and
    Happy New Year")
        Say Answer
```

Would display

```
3
```

Words

Purpose: Returns a count of the words in a string

Type: Rexx Function

Syntax: *Answer* = WORDS(*string*)

Example:

```
Answer = Words("Merry Christmas and Happy New Year")
Say Answer
```

Would display

```
6
```

XRange

Purpose: Return a string of characters between two characters in the ASCII character set.

Type: Rexx Function

Syntax: Result = XRange(startchar,endchar)
startchar-The first ASCII character that will be returned. The default is low-value (X'00').
endchar- The last ASCII character that will be returned. the default is high-value (X'FF').

Usage: This function will return all of the characters in the ASCII Character set between *startchar* and *endchar*, inclusive. If *startchar* is greater than *endchar*, then the string that is returned will wrap around through the beginning.

Example 1 The following example will not return the letters in the alphabet as a string. This is unfortunate, because the function would be a little more useful if it considered only valid characters.

```
Alphabet = XRange('A','Z')
Say Alphabet
```

The reason for this is that the leters of the alphabet do not appear continuously in the ASCII character set. What would be returned would be the ASCII characters represented by X'C1' through X'E9', inclusive:
ABCDEFGHI.....JKLMNOPQR.....STUVWXYZ

Example 2 The following example will return the alphabet.

```
Alphabet = XRange('A','I')XRange('J','R')XRange('S','Z')
Say Alphabet
```

X2C

Purpose: Converts a hexadecimal string to character

Type: Rexx Function

Syntax: *CharString* = X2C*(hexstring)*

Example:

```
Answer = X2C('C4C1E5C560F1F6F1F6')
Say Answer
```

Would display

```
Dave-1616
```

X2D

Purpose: Converts a hexadecimal string to decimal

Type: Rexx Function

Syntax: *Number* = X2D*(hexstring)*

Example:

```
Answer = X2D('FF')
Say Answer
```

Would display

```
255
```

```
Answer = X2D('FFFF')
Say Answer
```

Would display

```
65535
```

The maximum value that can be converted is X'3B9AC9FF', which equals 999,999,999 in decimal.

Instructions Not Covered

Certain instructions, commands, and functions are seldom, if at all, used in applications. These are used by system administrators and system programmers. These instructions, commands, and functions are listed here. Why are they even mentioned, if we are not going to document them?

They are listed, for the most part, to let you know (and to remind me) that they are indeed available, in case we want to use them or learn more about them. Documentation of these instructions, commands, and functions is beyond the scope of this manual. Check the appendix for additional sources of documentation.

DropBuf Delete a data stack buffer

ExecUtil Control Rexx processing options for the current Rexx environment

MakeBuf Add a buffer to the data stack

Options This instruction is used for DBCS (Double-Byte Character Set) character and data operations support.

Qbuf determine the number of data stack buffers that exist

Qelem Determine the number of data stack elements that exist

Storage Retrieve a number of bytes from a main storage address, or store a number of bytes into a main storage address.

Section II -A Starter Rexx Tutorial

Follow this tutorial by keying in the example Rexx execs and reading the associated commentary. If your results are not identical to those of the tutorial, try to find out *exactly* why. Each example builds on the previous ones, so it is important that you understand each before you move on.

```
/* Rexx Exec Tutorial #1 */
Say "Hello World"
```

This is one of the shortest Rexx execs ever written. All it does is display the famous programmer's primer message.

```
/* Rexx Exec Tutorial #2 */
Say "What is your name?"
Pull Answer
Say "So, your answer is " Answer". That is swell!"
```

This exec will ask you your name, and if you reply, it will echo it back, something like this:

```
So, your answer is  JOHNNY. That is swell!
```

```
/* Rexx Exec Tutorial #3 */
Say "What is your name?"
Pull Answer

Say "So, "Answer", how old are you?"
Pull Age
AgeIndays = Age * 365
Say "If you didn't lie to me, you are about" AgeInDays "days old."
```

This exec will ask you your name, and then perform a calculation. Notice that I used an apostrophe within a string that was enclosed in quotation marks. The exec's last display would look like this:

```
If you didn't lie to me, you are about 8030 days old.
```

From here, the possibilities are endless. Rather than waste your (and my) time by making you go through endless and pointless exercises, I will stop here, and let you get started with playing with some ideas of your own. Just remember: have fun!

Section III - Rexx Examples

I believe strongly in examples. No matter what someone is trying to say, it is clearer if it can be illustrated with a good example. A person can then glean an interpretation from that example.

The easiest way to write a Rexx exec is to take one that exists, and tailor it for your own use. Remember that like with any programming language, if you copy someone's source code verbatim, it's not ethically cool to put your name on it. If you use a major portion of source code that is provided to you for free, it is only fair to at least give credit to the author somewhere in your program. Please respect an author's inventiveness and hard work. Since Rexx execs are distributed with the source, if you publish any new Rexx execs that you created using an existing one as a basis, you are requested to at least credit the author. All of these examples were written by David Grund, and are free to use.

The examples provided here vary in purposes, but can be tailored to most specific needs that you have. They don't necessarily demonstrate the *best* way to write a Rexx exec in all cases. They *do,* however, demonstrate different techniques.

In some cases, some of the execs depend on data from ISPF libraries. That data is not included.

Disclaimer: All examples are provided for the sake of example only. There is no guarantee that these work as desired, or are entirely bug-free. You are free to, and encouraged to, develop and improve any or all of these examples.

The examples provided here are as follows:

ALLOCEIO	Allocate O/P dataset; write Rexx array to it
CAPTSO	Capture TSO command output
CHGBLKC	Insert a COBOL change block
CHGDATA	Modify a data file
CHGSTEP	Change steps in JCL
COMMANDS	List available commands
COMPCO	Compare two files of order numbers
COMPDS	Compare two sequential datasets
COMPPDS	Compare two PDS's
DD	Add a DD Statement
DELDUPS	Delete duplicate records
DURATION	Time an EXEC
FIXJCL	Fix Job Control
FX	File name cross-reference
HD	Hex Dump
INIT	Establish my TSO environment
INITSPF	Establish my ISPF environment
JOBCARD	Create a jobcard
LA	List TSO allocations
LISTDSI	List dataset information
LOTTERY	Pick lottery numbers
LPDSIX	List a PDS Index to a Sequential File
PROCSYMS	Perform symbolic substitution
PTS	PDS-to-Sequential; member name is prefix
PTS2	PDS-to-Sequential; member name is inserted
REXXMODL	Rexx Exec Model
SCALE	Display a Scale
SDN	Sorted Directory w/Notes; directory annotator
SHOWDUPS	Show duplicate records
STACK	Start another ISPF session
TIMEFMTS	Show all time formats
TIMETOGO	Display time until an event

ALLOCEIO - Allocate O/P dataset; write array to it

This is a code snippet that will allocate a TSO dataset, and then write a Rexx array to that dataset. The TSO dataset is deleted first, in case it already exists.

```
"Delete "MapDSN
"Allocate DD(FiCvtDS) DA("MapDSN") new space(1 1) tracks",
          "LRECL(80)  Block(6160)  recfm(f b) RETPD(0)"

"ExecIO" MapArray.0 "DiskW FiCvtDS (STEM MapArray. FINIS"
"Free DDNAME(FiCvtDS) DA("MapDSN")"
```

CAPTSO - Capture TSO command output

Using this exec, you can capture the output from just about any TSO command. The purpose, of course, is to dump it into a dataset and edit it.

```
/* CapTSO - Capture TSO Output - Rexx Exec */
/* Written by David Grund                  */

Dummy = OutTrap("output_line.","*")
"LISTd 'GRUND.ASSEMBLY.DATA' m"
NumLines = OutPut_Line.0
Say NumLines "lines were created"
Dummy = OutTrap("OFF")

"Delete CAPTSO.List"
"Allocate DD(CapTSO) DA(CAPTSO.List) new space(15 15) tracks",
          "LRECL(80)  Block(6160)  recfm(f b) RETPD(0)"

"ExecIO" OutPut_line.0 "DiskW CapTSO (STEM OutPut_Line. FINIS"
"Free DDNAME(CapTSO) DA(CAPTSO.List)"

ADDRESS "ISPEXEC" "EDIT Dataset(CAPTSO.List) "
```

CHGBLKC - Insert a COBOL change block

This Rexx exec is an ISPF edit macro, used to insert a program modification comment block into a program. By using this exec, you can make the comment block will look the same for every program, hence an increase in productivity. This technique, of course, can be used for any language. I have created one for Easytrieve and another for Assembler.

```
/* ChgBlkC - Insert COBOL Change Block - ISPF Edit Macro (REXX EXEC) */
ADDRESS "ISREDIT" "MACRO PROCESS"
J11= "000001*-------------------------------------------"
J12= "--------------------*"
J21= "000002*      PROGRAM MODIFICATION LOG             "
J22= "                     *"
J31= "000003* LOG #   DATE   WHO          REASON        "
J32= "                     *"
J41= "000004*   9   06/09/95 DAVID GRUND   change descrip"
J42= "tion line 1          *"
J51= "000005*                             change descrip"
J52= "tion line 2          *"
address "ISREDIT" "LINE_AFTER 0 ="  "'"J11""J12"'"
address "ISREDIT" "LINE_AFTER 1 ="  "'"J21""J22"'"
address "ISREDIT" "LINE_AFTER 2 ="  "'"J31""J32"'"
address "ISREDIT" "LINE_AFTER 3 ="  "'"J41""J42"'"
address "ISREDIT" "LINE_AFTER 4 ="  "'"J51""J52"'"
address "ISREDIT" "LINE_AFTER 5 ="  "'"J11""J12"'"
ADDRESS "ISREDIT" "Cursor = 1 0"
address "ISREDIT" "LINE_AFTER 0 = MSGLine",
  "'        "Please move these lines into the Remarks section."'"
```

CHGDATA - Modify a data file

This exec is used to modify a data file. It reads a data file into core (an array), modifies it (with hard-coded instructions), and then writes it back out. This is an exec that is tailored for use each time it is used.

```
/* ChgData - Change a File - REXX Exec     */
/* Written by David Grund                   */
/* This exec will read a data file, and modify it to contain     */
/* conditions for testing: invalid data, etc        */

/*------------- Main Body of Program ----------------------------*/
ARG IPDSN OPDSN
IPCtr = 0                           /* Input record counter       */
OPCtr = 0                           /* Output record counter      */

Call Pgm_Init

Do Forever
    Call ReadRec                    /* Read rec into stack; count */
    If IPEOF = "YES" then Leave
    Pull IPRec                      /* Get it from the stack      */
    Call ProcessRecord              /* Process it                 */
end

Call ProcEOJ                        /* EOJ Processing             */
Exit
/*-----------------------------------------------------------------*/

/*------------*/
/* Program Initialization */
/*------------*/
Pgm_Init:
"DelStack"
If IPDSN = "" then do
    Say "Command Type:

Syntax: ChgData IpDSN OpDSN"
    Exit
end

If OPDSN = "" then do
    OpDSN = IPDSN||.Modified
    Say "OPDSN not specified;" OPDSN "assumed."
end

"Alloc DDN(InFile) DSN("IPDSN") SHR"
If RC <> 0 then do
    Say "I could not allocate "IPDSN". Sorry."
    Exit
end

Dummy = ListDSI(IPDSN)
OPLRECL = SYSLRECL
OPBLKSize = SYSBlkSize
```

```
"Delete " OPDSN
"Free FI(OutFile)"
"Alloc DD(OutFile) DA("OPDSN") New space(15 15) tracks ",
      "Lrecl("OPLRECL") Block("OPBlkSize") Recfm(F B)"
If RC <> 0 then do
    Say "I could not allocate "OPDSN". Sorry."
    Exit
end
Return

/*------------*/
ReadRec:
/*------------*/
    "EXECIO 1 DiskR Infile"         /* Add the I/P rec to the stack */
    If RC <> 0 then do
        IPEOF = "YES"
        "EXECIO 0 DiskR Infile (Finis"  /* Close the input file */
    end
    Else IpCtr = IpCtr + 1          /* Count the records            */
    Return ""

/*------------*/
/* Process the Record */
/*------------*/
ProcessRecord:
    OpRec = IpRec

    If IpCtr = 11 then              /* Make the class invalid */
        OPRec = Substr(IPRec,1,9)||"0XRJC"||Substr(IpREc,15,307)

    If IpCtr = 16 then              /* Make the class invalid */
        OPRec = Substr(IPRec,1,9)||"123456789"||Substr(IpREc,19,303)

    If IpCtr = 22 then              /* Nom-transfer pack      */
        OPRec = Substr(IPRec,1,24)||"XYZ"||Substr(IpREc,28,294)

    If IpCtr = 33 then              /* Nom-minimum            */
        OPRec = Substr(IPRec,1,27)||"ABC"||Substr(IpREc,31,291)

    If IpCtr = 44 then              /* Store Number           */
        OPRec = Substr(IPRec,1,33)||"DE"||Substr(IpREc,36,286)

    If IpCtr = 55 then              /* Store quantity         */
        OPRec = Substr(IPRec,1,36)||"GHIJ"||Substr(IpREc,41,281)

    If IpCtr = 57 then              /* Warehouse Number         */
        OPRec = Substr(IPRec,1,313)||"89"||Substr(IpREc,316,6)

    If IpCtr = 32 then              /* Warehouse quantity     */
        OPRec = Substr(IPRec,1,315)||"DAVEG "

    Push OpRec
    "EXECIO" 1 "DiskW OutFile"
    OpCtr = OpCtr + 1               /* Count the records            */
Return
```

```
/*-------------*/
/* End-of-job Processing */
/*-------------*/
ProcEOJ:
    "DelStack"
    "EXECIO" 0 "DiskW OutFile (Finis"   /* Close the file        */
    "Free DDNAME(InFile OutFile)"
    Say "*** End of Job Totals ***"
    Say IpCtr "records read"
    Say OpCtr "records written"
Return
```

CHGSTEP - Change steps in JCL

When you have religiously numbered the steps in a job stream, and find that you have to insert a few, especially toward the *beginning*, your neatly-sequenced step names are compromised.

This Rexx exec will "quickly" renumber the steps so they are back in sequence and incremented by 10.

Here, you are creating the list of TSO commands that you will use to ultimately make the changes. Creating a TSO command set is less stressful than making the changes one-by-one. This way, you don't have to remember where you left off, and you can use ISPF's editor to mass-produce the change statements.

```
/* CHGSTEP - RENUMBER STEPS IN A JOB - REXX EXEC */

ADDRESS "ISREDIT" "MACRO PROCESS"
ADDRESS "ISREDIT" "C STEP190 STEP330 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP185 STEP320 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP180 STEP310 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP170 STEP300 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP160 STEP290 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP150 STEP280 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP140 STEP270 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP130 STEP260 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP120 STEP250 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP110 STEP240 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP100 STEP230 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP096 STEP220 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP095 STEP210 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP090 STEP200 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP080 STEP190 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP070 STEP180 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP060 STEP170 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP050 STEP160 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP040 STEP150 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP038 STEP140 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP037 STEP130 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP036 STEP120 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP035 STEP110 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP034 STEP100 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP033 STEP090 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP032 STEP080 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP031 STEP070 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP025 STEP060 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP024 STEP050 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP023 STEP040 WORD ALL 10"
ADDRESS "ISREDIT" "C STEP022 STEP030 WORD ALL 10"
```

COFFEE – the Coffee Game

```
/* Coffee - Coffee Game                                          REXX    */
/* This is a Rexx learning exercise                                      */
/* Two people take turns trying NOT to guess the number picked by the */
/* computer. The person who gets stuck with the number must buy.      */

P1Name = ""               /* Player 1 Name   */
P2Name = ""               /* Player 2 Name   */
Turn = 1

Say "Welcome to the Coffee Game. I will pick a random number. Two "
Say "people will take turns trying NOT to guess it. Whomever does,"
Say "LOSES!"
Say " "

Say "Player 1, please tell me your name!"
Pull P1Name
Upper P1name

Say "Player 2, please tell me your name!"
Pull P2Name
Upper P2name

Redo:
Guesses = 0               /* Number of guesses     */
CNo = Random(1,999)       /* Computer number */
/* Say "The computer picked number " CNo   */
InProgress = Y
P1Number = 0              /* Player 1 number */
P2Number = 0              /* Player 2 number */
Lower = 0
Upper = 1000

Do While InProgress = Y
  If Turn = 1 then do
    Turn = 2
    If (Upper - Lower) = 2 then do
      Say P1name "LOSES (by default). The number was " CNo
      Call Recap
      Leave
    End
    Say P1name", pick a number between " Lower " and " Upper "."
    Reask1 = N
    Pull P1Number
    If (P1Number <= Lower) | (P1Number >= Upper) then do
      Say "Dummy! I said between " Lower " and "Upper "! Try again!"
      ReAsk1 = Y
    End
    If ReAsk1 = Y then
      Turn = 1
    else do
      Guesses = Guesses + 1
      If CNo = P1Number then do
        InProgress = N
        Say P1Name "LOSES. The number was " P1Number
        Call Recap
        Leave
      End
      Else do
        If P1Number < CNo then Lower = P1Number
        If P1Number > CNo then Upper = P1Number
      End
```

```
    End
  End
  If Turn = 2 then do
    Turn = 1
    If (Upper - Lower) = 2 then do
      Say P2Name "LOSES (by default). The number was " CNo
      Call Recap
      Leave
    End
    Say P2Name", pick a number between " Lower " and " Upper "."
    Reask2 = N
    Pull P2Number
    If (P2Number <= Lower) | (P2Number >= Upper) then do
      Say "Dummy! I said between " Lower " and "Upper "! Try again!"
      ReAsk2 = Y
    End
    If ReAsk2 = Y then
      Turn = 2
    else do
      Guesses = Guesses + 1
      If CNo = P2Number then do
        InProgress = N
        Say P2Name "LOSES. The number was " P2Number
        Call Recap
        Leave
      End
      Else do
        If P2Number < CNo then Lower = P2Number
        If P2Number > CNo then Upper = P2Number
      End
    End
  End
End

Say "Again?"
Pull Ans
Upper Ans
If Ans = Y then signal ReDo
exit

Recap:
  Adjective = "only"
  If (Upper - Lower) > 25 then Adjective = "a Whopping"
  Say "The spread was " Adjective (Upper - Lower)
  Say "This game took " guesses "guesses."
Return
```

COMPCO - Compare Two Files of Order Numbers

```
/* CompCO - Compare Two Files Of Order Numbers - Rexx */
/* Written by David Grund                              */

IPDSN1 = "'DGrund.STEP120.SYSUT2'"
IPDSN2 = "'DGrund.STEP140.SYSUT2'"

Call Proc01  /* Program Initialization                */
Call Proc02  /* List First File to an Array           */
Call Proc03  /* List Second File to an Array          */
Call Proc04  /* Compare files now                     */
Call Proc99  /* Finalization                          */
Exit

/*-------------------------------------------------------*/
/*   Called Procedures                                   */
/*-------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say "CompCO - Compare Two Files of Order Numbers"
  Say "Proceeding..."
Return

/*------------*/
/* Read first file into core */
/*------------*/
Proc02:
  "Free fi(sysut1)"
  "Allocate Fi(SYSUT1) DA("IPDSN1") shr"
  "ExecIO  *  DiskR SYSUT1 (STEM File1Lines. FINIS"
  "Free FI(SYSUT1)"
  Say File1Lines.0 "records read from FILE2"
Return

/*------------*/
/* Read second file into core */
/*------------*/
Proc03:
  "Free fi(sysut2)"
  "Allocate Fi(SYSUT2) DA("IPDSN2") shr"
  "ExecIO  *  DiskR SYSUT2 (STEM File2Lines. FINIS"
  "Free FI(SYSUT2)"
  Say File2Lines.0 "records read from FILE1"
Return

/*------------*/
/* Compare the arrays now */
/*------------*/
Proc04:
  File1Rec = 1;     File2Rec = 1;
  Call ReadFile1        /* Read first record from File 1  */
  Call ReadFile2        /* Read first record from File 2  */
```

```
  InFile1Only = 0; Infile2Only = 0; InBoth = 0;

  Do Forever
      /* Say "Comparing " File1Line "to" File2Line */
      If File1Line = File2Line then do
        If File1Line = "99999" then Leave
        /* Say File1Line" in both files" */
        InBoth = InBoth + 1
        Call ReadFile1  /* Read next record from File 1   */
        Call ReadFile2  /* Read next record from File 2   */
      End
      Else If File1Line < File2Line then do
          InFile1Only = InFile1Only + 1
          Say File1Line" in FILE2 but not in FILE1"
          Call ReadFile1 /* Read next record from File 1  */
      End
      Else do
          Infile2Only = Infile2Only + 1
          Say File2Line" in FILE1 but not in FILE2"
          Call ReadFile2 /* Read next record from File 2  */
      End
  End
Return

/*------------*/
/* Read a record from File 1 */
/*------------*/
ReadFile1:
  If File1Rec > File1Lines.0 then
      File1Line = "99999"           /* "end of file" */
  Else DO
      File1Line = left(File1Lines.File1Rec,5)
      File1Rec = File1Rec + 1;
  End
Return

/*------------*/
/* Read a record from File 2 */
/*------------*/
ReadFile2:
  If File2Rec > File2Lines.0 then
      File2Line = "99999"           /* "end of file" */
  Else Do
      File2Line = left(File2Lines.File2Rec,5)
      File2Rec = File2Rec + 1;
  End
Return

/*------------*/
/* Finalization */
/*------------*/
Proc99:
    Say "In FILE2, not in FILE1:" ForMat(InFile1Only,5)
    Say "In FILE1, not in FILE2:" Format(InFile2Only,5)
    Say "In Both               :" ForMat(InBoth,5)
Return
```

COMPARE - Compare two sequential datasets

This exec will call IEBCOMPR to compare two datasets. You don't get a comprehensive and detailed listing of differences. Instead, you get notification as to whether the two datasets contain exactly the same data- a check that is required in a parallel test.

```
Arg IPDSN1 IPDSN2
If Arg() == 0 then do
    Say "Compare - Compare two TSO datasets"
    Say "        Type:

Syntax: Compare IPDSN1 IPDSN2"
    Say "        Please reenter this command"
    Exit
End

If SYSDSN(IPDSN1) = "OK" then nop
Else do
    Say "I cannot find "IPDSN1
    Exit
End

If Arg(2) == '' then nop
Else do
    Say "Please enter the name of the second dataset"
    Pull IPDSN2
End

If SYSDSN(IPDSN2) = "OK" then nop
Else do
    Say "I cannot find "IPDSN2
    Exit
End

"Free fi(sysut1,sysut2,sysin,sysprint)"
"Allocate Fi(SYSUT1) DA("IPDSN1") shr"
"Allocate Fi(SYSUT2) DA("IPDSN2") shr"
"Allocate Fi(SYSIN) DUMMY"
"Allocate Fi(SYSPRINT) DA(*)"
"Call 'SYS1.Linklib(IEBCOMPR)'"
```

COMPDSE – Compare Two Sequential Datasets - Enhanced

This exec will compare two sequential datasets, line-for-line, and report differences. Neither file is assumed to be in any kind of sequence. This differs from COMPDS in that it does not call IEBCOMPR; it does the compares internally.

The best thing about this tool is that it can be copied and modified for specialized file compare needs.

```
/* COMPDSE - Compare Two Datasets - Enhanced                    REXX */
/* Written by David Grund                                            */

/* This exec will compare two sequential datasets, line-for-line.
   We do not regard the input file's sequence.                    */
  ARG IPDS1 IPDS2

Call Proc01  /* Program Initialization               */
Call Proc02  /* Copy both datasets to an array       */
Call Proc04  /* Compare files now                    */
Call Proc99  /* Finalization                         */
Exit

/*--------------------------------------------------------------------*/
/*   Called Procedures                                                */
/*--------------------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say; Say; Say
  Say "CompDSE - Compare Two Datasets - Enhanced"
  If IPDS1 = "" ³ IPDS2 = "" then do
      Say "Command Syntax: CompDSE IPDS1 IPDS2"
      Exit
  End
  Say "Comparing "IPDS1 "to" IPDS2
  Say "Proceeding..."
Return

/*------------*/
/* Copy both datasets to arrays */
/*------------*/
Proc02:
  X = OutTrap("ON"); "Free Fi(IpFile) DA("IPDS1")"; X=OutTrap("OFF")
  "Alloc FI(IPFile) DA("IPDS1") SHR"
  If RC > 0 then exit
  "ExecIO * DiskR IPFile (Stem DS1Lines. Finis "
  "Free  FI(IPFile)"
  X = OutTrap("ON"); "Free Fi(IpFile)"; X=OutTrap("OFF")
  Say DS1Lines.0 "lines were found in" IPDS1

  X = OutTrap("ON"); "Free Fi(IpFile) DA("IPDS2")"; X=OutTrap("OFF")
  "Alloc FI(IPFile) DA("IPDS2") SHR"
  If RC > 0 then exit
```

```
  "ExecIO * DiskR IPFile (Stem DS2Lines. Finis "
  "Free  FI(IPFile)"
  X = OutTrap("ON"); "Free Fi(IpFile)"; X=OutTrap("OFF")
  Say DS2Lines.0 "lines were found in" IPDS2
Return

/*------------*/
/* Compare the files now */
/*------------*/
Proc04:
  CtrEquals = 0; CtrNEquals = 0;
  Do I = 1 to DS1Lines.0
    If DS1Lines.I = DS2Lines.I then
      CtrEquals = CtrEquals + 1
    Else do
      CtrNEquals = CtrNEquals + 1
      Say "Records #"I" differ:"
      Say "IPDS1: "DS1Lines.I
      Say "IPDS2: "DS2Lines.I
      Say
    End
  End
Return

/*------------*/
/* Finalization */
/*------------*/
Proc99:
    Say CtrEquals "records were identical"
    Say CtrNEquals "records were different"
Return
```

COMPPDS - Compare two PDS's

This command will compare two partitioned datasets. One is considered a "test" PDS; the other is considered a "production" PDS.

```
/* COMPPDS - Compare PDS's - REXX Exec     */
  ARG TestPDS ProdPDS

/* This command will compare a "Test" PDS against a "Production" PDS.*/

Call Proc01  /* Program Initialization              */
Call Proc02  /* List First PDS Members to an Array  */
Call Proc03  /* List Second PDS Members to an Array */
Call Proc04  /* Compare files now                   */
Call Proc99  /* Finalization                        */
Exit

/*-------------------------------------------------------*/
/*   Called Procedures                                   */
/*-------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say "CompPDS - Compare PDS's"
  If TestPDS = "" | PRODPDS = "" then do
      Say "Command Type:

Syntax: CompPDS TestPDS ProdPDS"
      Exit
  End
  Say "Comparing "TestPDS "to" ProdPDS
  Say "Proceeding..."
Return

/*------------*/
/* List Members of TestPDS  */
/*------------*/
Proc02:
  /* Say "Reading "TESTPDS"..." */
  Dummy = OutTrap("TestMems.","*")
  "LISTD " TestPDS "M"
  NumLines = TestMems.0 - 6
  Say NumLines "Member names were found in" TestPDS
  Dummy = OutTrap("OFF")

  NumTestRecs = TestMems.0 + 1
  /* Clean up the array */
  Do I = 1 to 6 ; TestMems.I = "" ; End
  Do I = 7 to TestMems.0
      TestMems.I = strip(TestMems.I)
  End
Return

/*------------*/
/* List Members of ProdPDS  */
```

```
/*------------*/
Proc03:
  /*  Say "Reading "ProdPDS"..." */
  Dummy = OutTrap("ProdMems.","*")
  "LISTD " ProdPDS "M"
  NumLines = ProdMems.0 - 6
  Say NumLines "Member names were found in" ProdPDS
  Dummy = OutTrap("OFF")

  NumProdRecs = ProdMems.0 + 1
  /* Clean up the array */
  Do I = 1 to 6 ; ProdMems.I = "" ; End
  Do I = 7 to ProdMems.0
      ProdMems.I = strip(ProdMems.I)
  End
Return

/*------------*/
/* Compare the member names now */
/*------------*/
Proc04:
  TestCurrRec = 6;  ProdCurrRec = 6;
  Call ReadTest                    /* Read first record from TestPDS */
  Call ReadProd                    /* Read first record from ProdPDS */
  InTestOnly = 0; InProdOnly = 0; InBoth = 0;

  Do Forever
      /* Say "Comparing " TestMem "to" ProdMem */
      If TestMem = ProdMem then do
        If TestMem = "99999999" then Leave
        If TestMem = " " then nop
        Else do
          InBoth = InBoth + 1
          Call CompMembers   /* Compare the members           */
        End
        Call ReadTest        /* Read next record from TestPDS */
        Call ReadProd        /* Read next record from ProdPDS */
      End
      Else If Testmem < ProdMem then do
          InTestOnly = InTestOnly + 1
          Say TestMem" in " TestPDS "but not in "ProdPds
          Call ReadTest            /* Read next record from TestPDS */
      End
      Else do
          InProdOnly = InProdOnly + 1
          Call ReadProd            /* Read next record from ProdPDS */
      End
  End
Return

/*------------*/
/* Compare the members, line for line */
/*------------*/
CompMembers:
    /* First, normalize the datasetnames */
    If Left(TestPDS,1) = "'" then do
        TestIPDSN = strip(TestPDS)
```

```
        TestIPDSN = DelStr(TestIPDSN,1,1)
        IDLen = length(TestIPDSN)
        TestIPDSN = DelStr(TestIPDSN,IDLen,1)
    end
    else
        TestIPDSN = TestDSN
    TestIPDSN = "'"||TestIPDSN||"("||TestMem||")'"
    If Left(ProdPDS,1) = "'" then do
        ProdIPDSN = strip(ProdPDS)
        ProdIPDSN = DelStr(ProdIPDSN,1,1)
        IDLen = length(ProdIPDSN)
        ProdIPDSN = DelStr(ProdIPDSN,IDLen,1)
    end
    else
        ProdIPDSN = ProdDSN
    ProdIPDSN = "'"||ProdIPDSN||"("||ProdMem||")'"
    Address TSO
    "Free fi(SYSUT1 SYSUT2 SYSPrint SYSIN)"
    "Alloc Fi(SYSUT1) Da("||TestIPDSN") SHR"
    "Alloc Fi(SYSUT2) Da("||ProdIPDSN") SHR"
    "Alloc FI(SYSPrint) DUMMY"
    "Alloc FI(SYSIN) DUMMY"
    "Call 'SYS1.LinkLib(IEBCOMPR)'"
    RtrnCD = RC
    If RtrnCD = 0 then
        Say TestMem ||": The modules are identical"
    Else
        Say TestMem ||":  The modules differ!"
Return

/*------------*/
/* Read a record from TestPDS */
/*------------*/
ReadTest:
  TestCurrRec = TestCurrRec + 1;
  If TestCurrRec > NumTestRecs then
      TestMem = "99999999"          /* "end of file" */
  Else DO
      TestMem = TestMems.TestCurrRec
      /* Say "I just read from TEST: " TestMem */
  End
Return

/*------------*/
/* Read a record from ProdPDS */
/*------------*/
ReadProd:
  ProdCurrRec = ProdCurrRec + 1;
  If ProdCurrRec > NumProdRecs then
      ProdMem = "99999999"          /* "end of file" */
  Else Do
      ProdMem = ProdMems.ProdCurrRec
      /* Say "I just read from PROD: " ProdMem */
  End
Return

/*------------*/
```

```
/* Finalization */
/*-------------*/
Proc99:
    Say "In Test, not in prod:" InTestOnly
    Say "In Prod, not in test:" InProdOnly
    Say "In Both              :" InBoth
Return
```

ConcatL - Concatenate Libraries

This command will concatenate a library to a current DDName's allocation.

If you wanted to add your Rexx Exec library to an existing SYSEXEC allocation, you could do it two ways:

1) You could free SYSEXEC, and then reallocate all necessary libraries, including your own. But that would make you dependent upon someone in Systems to tell you when the normal allocation (all necessary libraries) changes.
2) You could simply add your library to the current concatenation, using this example. This way, if the "necessary library" sequence changes, you will not be affected. Your library will always be concatenated to that set.

To execute this example:

```
Exec 'userid.REXX.EXEC(ConCatL)' 'SYSEXEC userid.REXX.EXEC'
```

```
/* ConCatL - Allocate a library to an exiting concatenation  REXX */
Arg SearchDD LibToAdd
LibToAdd = "'"LibToAdd"'"    /* Add some quotes */

Found = "NO"
Concat = ""   /* Set to null in case DDName not allocated */
Dummy = OutTrap("Sysoutline.","*")          /* Start capture */
"ListALC Status"
Dummy = OutTrap("OFF")                      /* Stop Capture  */

Do I = 1 to Sysoutline.0
  /* Say "looking at " Sysoutline.I */
  If SubStr(Sysoutline.I,3,8) = SearchDD then do
      Found = "YES"
      I2 = I - 1
      DSN = SubStr(SysoutLine.I2,1,44)
      DSN = strip(DSN)
      Concat = "'" || DSN || "'" /* add apostrophe */
      Leave I
  End
End I

If Found = "YES" then do
  Do I3 = I + 1 to SysoutLine.0 - 1 by 2
    I4 = I3 + 1
    If SubStr(Sysoutline.I4,3,8) <> "        ",
      then Leave
    If SubStr(SysoutLine.I3,1,2) <> "  " then do
      DSN = Substr(Sysoutline.I3,1,45)
      DSN = strip(DSN)
      Concat = Concat || " '" || DSN || "'"
    End
  End
End

"Allocate DDName("SearchDD") SHR Reuse ",
    "DSName("Concat LibToAdd")"
```

```
Say "ConCatL added " LibToAdd "to" SearchDD"."
```

CPDSIX – Compare Two PDS Indexes

This exec will simply compare the directories (or indexes) of two PDS's, and report the differences. This tool can be a very helpful quality control tool.

```
/* CPDSIX - Compare PDS Indexes - REXX Exec      */
ARG IPPDS1 IPPDS2

Call Proc01  /* Program Initialization             */
Call Proc02  /* List First PDS Members to an Array  */
Call Proc03  /* List Second PDS Members to an Array */
Call Proc04  /* Compare files now                   */
Call Proc99  /* Finalization                        */
Exit

/*--------------------------------------------------------------------*/
/*   Called Procedures                                                */
/*--------------------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say; Say; Say;
  Say "CPDSIX, Comparing..."
  Say "PDS1: "IPPDS1
  Say "PDS2: "IPPDS2
Return

/*------------*/
/* List Members of IPPDS1   */
/*------------*/
Proc02:
  /* Say "Reading "IPPDS1"..." */
  Dummy = OutTrap("PDS1Mems.","*")
  "LISTD " IPPDS1 "M"
  NumLines = PDS1Mems.0 - 6
  Say NumLines "Member names were found in" IPPDS1
  Dummy = OutTrap("OFF")

  NumPDS1Recs = PDS1Mems.0 + 1
  /* Clean up the array */
  Do I = 1 to 6 ; PDS1Mems.I = "" ; End
  Do I = 7 to PDS1Mems.0
      PDS1Mems.I = strip(PDS1Mems.I)
  End
Return

/*------------*/
/* List Members of IPPDS2   */
/*------------*/
Proc03:
  /*  Say "Reading "IPPDS2"..." */
  Dummy = OutTrap("PDS2Mems.","*")
  "LISTD " IPPDS2 "M"
  NumLines = PDS2Mems.0 - 6
```

```
  Say NumLines "Member names were found in" IPPDS2
  Dummy = OutTrap("OFF")

  NumPDS2Recs = PDS2Mems.0 + 1
  /* Clean up the array */
  Do I = 1 to 6 ; PDS2Mems.I = "" ; End
  Do I = 7 to PDS2Mems.0
      PDS2Mems.I = strip(PDS2Mems.I)
  End
Return

/*------------*/
/* Compare the member names now */
/*------------*/
Proc04:
  PDS1CurrRec = 6;  PDS2CurrRec = 6;
  Call ReadPDS1                    /* Read first record from IPPDS1 */
  Call ReadPDS2                    /* Read first record from IPPDS2 */
  InPDS1Only = 0; InPDS2Only = 0; InBoth = 0;

  Do Forever
      /* Say "Comparing " PDS1Mem "to" PDS2Mem */
      If PDS1Mem = PDS2Mem then do
        If PDS1Mem = "99999999" then Leave
        If PDS1Mem = " " then nop
        Else do
          InBoth = InBoth + 1
        End
        Call ReadPDS1        /* Read next record from IPPDS1 */
        Call ReadPDS2        /* Read next record from IPPDS2 */
      End
      Else If PDS1mem < PDS2Mem then do
          InPDS1Only = InPDS1Only + 1
          Say PDS1Mem" in PDS1 but not in PDS2"
          Call ReadPDS1            /* Read next record from IPPDS1 */
      End
      Else do
          InPDS2Only = InPDS2Only + 1
          Say PDS2Mem" in PDS2 but not in PDS1"
          Call ReadPDS2            /* Read next record from IPPDS2 */
      End
  End
Return

/*------------*/
/* Read a record from IPPDS1 */
/*------------*/
ReadPDS1:
  PDS1CurrRec = PDS1CurrRec + 1;
  If PDS1CurrRec = NumPDS1Recs then
      PDS1Mem = "99999999"          /* "end of file" */
  Else DO
      PDS1Mem = PDS1Mems.PDS1CurrRec
      /* Say "I just read from PDS1: " PDS1Mem */
  End
Return
```

```
/*------------*/
/* Read a record from IPPDS2 */
/*------------*/
ReadPDS2:
  PDS2CurrRec = PDS2CurrRec + 1;
  If PDS2CurrRec = NumPDS2Recs then
      PDS2Mem = "99999999"          /* "end of file" */
  Else Do
      PDS2Mem = PDS2Mems.PDS2CurrRec
      /* Say "I just read from PDS2: " PDS2Mem */
  End
Return

/*------------*/
/* Finalization */
/*------------*/
Proc99:
    Say "In PDS1, not in PDS2:" InPDS1Only
    Say "In PDS2, not in PDS1:" InPDS2Only
    Say "In Both             :" InBoth
Return
```

DD - Add a DD Statement

This command will add the JCL for an output disk DD statement. It is designed for JES2, and will also generate a delete step.

```
/* DD - ISPF Edit Macro (REXX EXEC)        */
ADDRESS "ISREDIT" "MACRO PROCESS"
address "ISREDIT" "(XDSN)=DATASET"
address "ISREDIT" "(XMEM)=MEMBER"

/* First get the user ID from a list */
UserID = sysvar(SYSUID)
UserName = "an unknown TSO user"
If UserID = "GRUND" then UserName = "David Grund"
   Say "UserID =" UserID "; Name =" UserName
OurDSN = UserID||".whatever"

/* Now create the JCL statements */
J01 = "//*"
J021 = "//*-----------------------------------"
J022 = "--------------------------------*"
J03 = "//* STEPNN1  - IEFBR14  - DELETE OUTPUT DATASETS"
J04 = "//STEPNN1  EXEC PGM=IEFBR14"
J05 = "//DELDS    DD  DSN="||OurDSN||","
J06 = "//         DISP=(MOD,DELETE),UNIT=SYSDA,SPACE=(TRK,(0))"
J07 = "//*"
J08 = "//* Output file description"
J09 = "//filenam  DD  DSN="||OurDSN||","
J10 = "//         DISP=(NEW,CATLG,DELETE),"
J11 = "//         UNIT=SYSDA,SPACE=(080,(123,123),RLSE),AVGREC=U,"
J13 = "//         DCB=(DSORG=PS,RECFM=FB,LRECL=080,BLKSIZE=0)"

/* Now insert them into the currently-edited member */
address "ISREDIT" "LINE_AFTER  0 =" "'"J01"'"
address "ISREDIT" "LINE_AFTER  1 =" "'"J021""J022"'"
address "ISREDIT" "LINE_AFTER  2 =" "'"J03"'"
address "ISREDIT" "LINE_AFTER  3 =" "'"J021""J022"'"
address "ISREDIT" "LINE_AFTER  4 =" "'"J04"'"
address "ISREDIT" "LINE_AFTER  5 =" "'"J05"'"
address "ISREDIT" "LINE_AFTER  6 =" "'"J06"'"
address "ISREDIT" "LINE_AFTER  7 =" "'"J07"'"
address "ISREDIT" "LINE_AFTER  8 =" "'"J08"'"
address "ISREDIT" "LINE_AFTER  9 =" "'"J09"'"
address "ISREDIT" "LINE_AFTER 10 =" "'"J10"'"
address "ISREDIT" "LINE_AFTER 11 =" "'"J11"'"
address "ISREDIT" "LINE_AFTER 12 =" "'"J13"'"

/* NOW PUT ASTERISKS IN COL 71 IN LINES 4 THRU 8 */
ADDRESS "ISREDIT" "LABEL 2 = .LSTART "
ADDRESS "ISREDIT" "LABEL 3 = .LEND   "
ADDRESS "ISREDIT" "CHANGE ' ' '*' 71 .LSTART .LEND ALL"
ADDRESS "ISREDIT" "RESET"

ADDRESS "ISREDIT" "Cursor = 1 0"
```

```
address "ISREDIT" "LINE_AFTER 0 = NoteLine",
  "'--- This is the delete step ---------------------------'"
address "ISREDIT" "LINE_AFTER 7 = NoteLine",
  "'--- The output DD specification follows ---------------'"
address "ISREDIT" "LINE_AFTER 13 = NoteLine",
  "'-----------------------------------------------------'"

address "ISREDIT" "LINE_AFTER 13 = NoteLine",
  "'--- Constructed especially for " || UserName "'"
```

DELDUPS - Delete Duplicate Records

```
/* DelDups - Delete Duplicate Lines     REXX Exec  */
ADDRESS ISREDIT
'MACRO (begcol endcol)'
If Begcol = '?' then do
    zedsmsg = 'DelDups begcol,endcol'
    zedlmsg = 'Command syntax: DelDup beginning col, ending col'
    signal quitme
end
numcheck = DATATYPE(begcol,N)         /* Determine if any parms have */
If NumCheck /= 1 then BegCol = 1      /* been passed.                */
numcheck = DATATYPE(endcol,N)
If NumCheck /= 1 then 'ISREDIT (endcol) = LRECL'

'ISREDIT (currline) = LINENUM .ZFIRST'  /* save starting record # */
'ISREDIT (lastline) = LINENUM .ZLAST'   /* save ending record #   */
'ISREDIT (cl,cc)    = CURSOR'           /* save cursor position   */
DupCnt = 0
Do currline = 1 to lastline - 1
    If CurrLine > (LastLine - 1) then leave
    'ISREDIT (line1) = LINE' currline
    line1 = substr(line1,begcol,(endcol - begcol) + 1)
    nextline = currline + 1
    'ISREDIT (line2) = LINE' nextline   /* get next record */
    line2 = substr(line2,begcol,(endcol - begcol) + 1)
    If line1 == line2 then do
      DupCnt = DupCnt + 1
      "ISREDIT LABEL " currline " = .A"
      "ISREDIT LABEL " nextline " = .B"
      "ISREDIT Delete " nextline
      currline = currline - 1 ; lastline = lastline - 1
    end
end
zedsmsg = DupCnt 'DUPS Deleted'
zedlmsg = DupCnt 'duplicate lines were deleted'
Quitme:
ADDRESS ISPEXEC
'SETMSG MSG(ISRZ000)'
EXIT 0
```

DURATION - Time an EXEC

This Rexx exec can be modified to time the processing of a command. It's a good idea to do this is some of the longer-running execs, or to brag about how fast your exec can accomplish something!

```
/* Duration - Rexx EXEC                    */
/* This command will test the code to perform a calculation
   of command duration                     */
STime = Time(E)             /* Start time  */
Say "I am waiting for you to hit enter!"
Pull Answer
ETime = Time(E)            /* End Time */
Duration = ETime - STime
Say "This command took" Duration "seconds!"
```

FindMem - Find a Member in a Concatenation

This Rexx exec will search a concatenated set of libraries for a specific member name. This is useful for when you want to know exactly which library an ISPF panel or a Rexx Exec is being executed from.

This command can also be executed in batch to look for copybooks or load modules in a concatenation.

```
/* FindMem - Find a Member in a Concatenation       REXX */
ARG OurDD OurMem
Call Proc01            /* Initialization    */
Call Proc02            /* ListA to an array */
Call Proc03            /* Adjust the array  */
Call Proc04            /* Remove 'KEEP' lines */
/* Call Proc05 */      /* Write the array to a dataset and view it */
Call Proc06            /* Isolate the DD     */
Call Proc10            /* Now search each PDS */
Exit

/*-----------*/
/* Proc01 - Initialization  */
/*-----------*/
Proc01:
  If OurDD = "" | OurMem = "" then do
    Say "Command syntax: FindMem DDName MemName"
    Exit(16)
  End
Return

/*-----------*/
/* Proc02 - ListA to an array */
/*-----------*/
Proc02:
  Dummy = OutTrap("output_line.","*")
  "LISTA SY ST"
  NumLines = OutPut_Line.0
  /* Say NumLines "lines were created" */
  Dummy = OutTrap("OFF")
Return

/*-----------*/
/* Proc03 - Adjust the array */
/*-----------*/
Proc03:
  /* Move the line with the DDNAME above the first datasetname
     that it is concatenated to. It is currently below.          */
  Do I = 1 to NumLines
      Col1_2 = SubStr(OutPut_Line.I,1,2)
      Col3 = SubStr(OutPut_Line.I,3,1)
      Col12_15 = SubStr(OutPut_Line.I,12,4)
      If Col1_2 = '  ' & ,
         Col3 /= ' ' & ,
         Col12_15 = 'KEEP' then do
          J = I - 1
          SaveLine      = OutPut_Line.I
          Output_Line.I = OutPut_Line.J
```

```
          Output_Line.J = SaveLine
      end
  end
Return

/*-----------*/
/* Proc04 - Remove all lines that say only "KEEP" */
/*-----------*/
Proc04:
  J = 0         /* Output array counter */
  Do I = 1 to NumLines
    ThisLine = strip(Output_Line.I)
    If (left(ThisLine,4) = 'KEEP') | ,
       (left(ThisLine,8) = 'TERMFILE') then nop=nop
    Else do
      J = J + 1; NewArray.J = OutPut_Line.I
    End
  End
  NewArray.0 = J
Return

/*-----------*/
/* Proc05 - Write the Array to a dataset and view it */
/*-----------*/
Proc05:
  "Delete FindMem.list"
  "Allocate DD(FMList) DA(FindMem.List) new space(1 1) tracks",
            "LRECL(80)  Block(5600)  recfm(f b) RETPD(0)"

  "ExecIO" NewArray.0 "DiskW FMList (STEM NewArray. FINIS"
  "Free DDNAME(FMList) DA(FindMem.List)"

  ADDRESS "ISPEXEC" "View Dataset(FindMem.List)"
Return

/*-----------*/
/* Proc06 - Isolate the DD */
/*-----------*/
Proc06:
  J = 0    /* DSNArray counter */
  DDName = ''
  Do I = 1 to NewArray.0
    If left(NewArray.I,2) = '  ' then
      DDName = left(strip(NewArray.I),8)
    else do
      ThisRec = DDName||strip(NewArray.I)
      J = J + 1; DSNArray.J = ThisRec
    end
  End
  DSNArray.0 = J

  /* Do I = 1 to DSNArray.0
    Say DSNArray.I
  End  */
Return

/*-----------*/
```

```
/* Proc10 - Search each DSN for our member name */
/*-----------*/
Proc10:
  DDFound = 0 ; MemFnd = 0
  Do I = 1 to DSNArray.0
    If left(DSNArray.I,8) = OurDD then do
      DDFound = DDFound + 1
      DSN = strip(substr(DSNArray.i,9,63))
      /* Say "Looking through DSN" DSN */
      Call Proc101   /* Check this DSN */
    End
  End
  Say "All together, I found "DDFound" DSN's allocated to DDName "OurDD
  TWord = 'times'; If MemFnd = 1 then TWord = 'time'
  Say "I found member "OurMem MemFnd TWord"."
Return

/*-----------*/
/* Proc101 - Search this DSN for our member name */
/*-----------*/
Proc101:
  /* First make sure this dataset is a PDS */
  RC = ListDSI("'"DSN"'" Directory)
  If RC > 0 then do
    Say 'Error processing 'DSN
    Say SYSMSGLVL1; Say SYSMSGLVL2 ; Say
    Return
  End
  If SYSDSORG = "PO" then do
    Dummy = OutTrap("PDSLines.","*")
    "LISTD '"DSN"' M"
    NumLines = PDSLines.0
    Dummy = OutTrap("OFF")
    Do K = 6 to PDSLines.0
      /* Say "The line is: "PDSLines.K */
      If Pos(OurMem,PDSLines.K) > 0 then do
        Say "I found member "OurMem" in "DSN
        MemFnd = MemFnd + 1
      End
    End
  End
  Else
    Say "Dataset "DSN" is not a PDS."
Return
```

FixJCL - Fix Job Control

FixJCL is a Rexx exec that will read a set of Mainframe JCL, and make certain format changes.

Granted that these format changes are to personal style and specifications: it puts the datasetname on the first line, the disposition parameters on the second line (unless they are short), the space parameters together on the next line, the DCB parameters on the next, and anything else on the last.

The beauty of this exec is that it parses the JCL, and isolates just about every "common" JCL field, so if you didn't want to create finished, or "fixed" JCL, you could do whatever processing you wanted. Additionally, the code is all there, so you could make any desired enhancements.

The exec first reads the JCL into an array, parses and identifies it, and then creates a file of fields. The code to catalogue this particular file has been commented out, but for testing or development, you would want to open this code back up.

That array is then read, and the final JCL file is created.

Please note that the objective of this exec is twofold: to present a usable tool, and to provide the code to enhance the tool. There is a lot of room for improvement in this particular tool. It isn't meant to be a finished and shiny product. It is meant to accomplish something very useful, and allow the user to make any desired improvements or enhancements to something that has a good, solid base.

The code, in its entirety, follows:

```
/* FixJCL - Create a Fixed File of JCL                - REXX Exec */

ARG IPDSN
Call Proc01                         /* Program initialization       */
Call Proc10                         /* Parse the JCL                */
Call Proc30                         /* Write the control card array */
Call Proc40                         /* Write the fixed JCL          */

Call ProcEOJ                        /* EOJ Processing               */
Exit

/*------------
Output record layout: */
Cols 1-3: Record Type
Cols 4-72: text
1-- JOBCARD
101 Jobname
102 Accounting Info
103 Routing Info
104 MSGLEVEL
105 MSGCLASS
106 CLASS
```

```
107 NOTIFY
199 Other info
4-- STEP/EXEC
401 Stepname
402 PGM= or procname
403 PARM
404 COND
405 REGION
5-- DD Statement
501 DDName
502 SYSOUT
503 'DUMMY'
504 DSN
505 DISP
506 UNIT
507 SPACE
508 AVGREC
509 DCB first positional
510 DCB DSORG
511 DCB RECFM
512 DCB LRECL
513 DCB BLKSIZE
514 LABEL
515 COPIES
516 DEST
517 HOLD
518 TRTCH
519 OUTPUT
520 VOL=SER
521 FREE
601 Data
701 COMMENT
801 OUTPUT
901 Unknown
--------------*/

/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  "DelStack"
  If IPDSN = "" then do
      Say "Command syntax: FixJCL DSN"
      Exit
  end

  Say "FixJCL Working on " IPDSN ": ",
       sysvar(SYSUID) Date(U) Time() "..."

  Call Proc011                    /* Read the JCL into an array    */
  ExpectingContinuation = "N"

  OpCtr = 0
  OData.0 = OpCtr                 /* Create the output array  */
  Spaces = "                                                   "
Return
```

```
/*------------*/
/* Read the JCL into an array */
/*------------*/
Proc011:
  "Alloc DDN(InFile) DSN("IPDSN") SHR"
  If RC <> 0 then do
      Say "I could not allocate "IPDSN". Sorry."
      Exit
  end
  "ExecIO * DiskR InFile (Stem JCL. Finis"
  "Free FI(InFile)"
  Say "I read "JCL.0" lines of JCL into the array."
Return

/*------------*/
/* Parse the JCL  */
/*------------*/
/* This routine will parse the JCL, and create an array of
   control cards representing the JCL values                    */
Proc10:
  Do I = 1 to JCL.0
    Record = strip(left(JCL.I,72))     /* Look at only cols 1-72  */
    Call Proc20                        /* Parse/identify the stmt */
    If RecID = "J" then Call Proc230   /* Job card         */
    If RecID = "E" then Call Proc240   /* Exec card        */
    If RecID = "D" then Call Proc250   /* DD card          */
    If RecID = "A" then Call Proc260   /* data card        */
    If RecID = "C" then Call Proc270   /* Comment card     */
    If RecID = "O" then Call Proc280   /* Output Card      */
    If RecID = "U" then Call Proc290   /* Unknown card     */
  End
Return

/*------------*/
/* Parse and identify the JCL Statement */
/*------------*/
Proc20:
  Parse Var Record Piece1 Piece2 Piece3
  If ExpectingContinuation = "Y" then Return
  RecID = "U"                                  /* Unknown */
  If left(Record,1) /= "/"  then RecID = "A"   /* Data    */
  If left(Record,3) = "//*" then RecID = "C"   /* Comment */
  Else
    If left(Record,2) = "//" then do
      If strip(Piece2) = "JOB"  then RecID = "J" /* Job card       */
      If strip(Piece2) = "EXEC" then RecID = "E" /* Execute card   */
      If strip(Piece2) = "DD"   then RecID = "D" /* DD card        */
      If strip(Piece2) = "OUTPUT" then RecID = "O" /* Output card    */
    End
  /* Say "The following record:"
  Say Record
  Say "has been identified as "RecID  */
Return

/*------------*/
/* Process Job Card  */
/*------------*/
```

```
Proc230:
  /* If this is the first card of a set, then the variable
     ExpectingContinuation will be "N". For all other cards, it
     will be 'Y'.                                              */
  If ExpectingContinuation = "N" then JobCard = Record
  Else                               Jobcard = Jobcard||Piece2

  If right(Record,1) = ',' then ExpectingContinuation = "Y"
  Else do                      /* We have read the final job card */
    ExpectingContinuation = "N"
    Call Proc2301                  /* Parse the job statement */
    Call Proc2302                  /* Write them to the array */
  End
Return

/*------------*/
/* Parse the Job Statement */
/*------------*/
Proc2301:
  Parse Var Jobcard Piece1 Piece2 Piece3
  V101 = DelStr(Piece1,1,2)           /* Job name               */

  Parse var Piece3 V102 "," Piece3    /* Job accounting info */

  If left(Piece3,1) = "'" then Piece3 = DelStr(Piece3,1,1)
  Parse var Piece3 V103 "'" Piece3    /* Routing info        */

  V104 = " " ; V105 = "" ; V106 = ""; V107 = "" ; V199 = ""
  Parse var Piece3 PJ1 "," PJ2 "," PJ3 "," PJ4 "," ,
                   PJ5 "," PJ6 "," PJ7 "," PJ8
  Do J = 1 to 8
    ThisArg = Value(PJ||J)
    If left(ThisArg,9) = "MSGLEVEL=" then do
      V104 = right(ThisArg,1)
      ThisPos = Index(Piece3,ThisArg)                /* Del */
      Piece3 = DelStr(Piece3,ThisPos,length(ThisArg)+1)
    End
    If left(ThisArg,9) = "MSGCLASS=" then do
      V105 = right(ThisArg,1)
      ThisPos = Index(Piece3,ThisArg)                /* Del */
      Piece3 = DelStr(Piece3,ThisPos,length(ThisArg)+1)
    End
    If left(ThisArg,6) = "CLASS=" then do
      V106 = right(ThisArg,1)
      ThisPos = Index(Piece3,ThisArg)                /* Del */
      Piece3 = DelStr(Piece3,ThisPos,length(ThisArg)+1)
    End
    If left(ThisArg,7) = "NOTIFY=" then do
    V107 = DelStr(ThisArg,1,7)
      ThisPos = Index(Piece3,ThisArg)                /* Del */
      Piece3 = DelStr(Piece3,ThisPos,length(ThisArg)+1)
    End
  End
  V199 = V199||Piece3             /* Whatever is left */
  If left(V199,1) = "," then V199 = DelStr(V199,1,1)
  If right(V199,1) = "," then V199 = DelStr(V199,length(V199),1)
Return
```

```
/*------------*/
/* Write the job information to the array */
/*------------*/
Proc2302:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "101"||V101
    OpCtr = OpCtr + 1 ; OData.OpCtr = "102"||V102
    OpCtr = OpCtr + 1 ; OData.OpCtr = "103"||V103
    OpCtr = OpCtr + 1 ; OData.OpCtr = "104"||V104
    OpCtr = OpCtr + 1 ; OData.OpCtr = "105"||V105
    OpCtr = OpCtr + 1 ; OData.OpCtr = "106"||V106
    OpCtr = OpCtr + 1 ; OData.OpCtr = "107"||V107
    OpCtr = OpCtr + 1 ; OData.OpCtr = "199"||V199
Return

/*------------*/
/* Execute Card */
/*------------*/
Proc240:
  /* If this is the first card of a set, then the variable
     ExpectingContinuation will be "N". For all other cards, it
     will be 'Y'.                                              */
  If ExpectingContinuation = "N" then ExecCard = Record
  Else                                ExecCard = ExecCard||Piece2

  If right(Record,1) = ',' then ExpectingContinuation = "Y"
  Else do                     /* We have read the final card */
    ExpectingContinuation = "N"
    /* Say "The entire execute statement follows" */
    /* Say ExecCard                               */
    Call Proc2401                 /* Parse the exec statement */
    Call Proc2402                 /* Write them to the array  */
  End
Return

/*------------*/
/* Parse the Exec Statement */
/*------------*/
Proc2401:
  /* With the job statement, we parse the whole thing at once.
  We cannot do that with the Exec, because of operands that begin in
  a left parenthesis, like the COND. Therefore, we have to
  "break off" a piece at a time.                                */

  V402 = " " ; V403 = ""; V404 = "" ; V405 = "";
  V499 = ""                                          /* Init vars */

  Parse Var ExecCard Piece1 Piece2 Piece3
  V401 = DelStr(Piece1,1,2)           /* Step name              */
  Piece3 = strip(Piece3)

  Do 10                 /* There shouldn't be more than this */
    If left(Piece3,1) = "," then Piece3 = DelStr(Piece3,1,1)
    If left(Piece3,4) = "PGM=" then do
      Parse Var Piece3 V402 "," Piece3
      V402 = right(V402,length(V402)-4)
    End
```

```
    If left(Piece3,6) = "PARM='" then do
      Piece3 = DelStr(Piece3,1,6)
      Parse Var Piece3 V403 "'" Piece3
    End
    If left(Piece3,6) = "PARM=(" then do
      Piece3 = DelStr(Piece3,1,6)
      Parse Var Piece3 V403 ")" Piece3
    End
    If left(Piece3,5) = "PARM=" then do
      Parse Var Piece3 V403 "," Piece3
      V403 = right(V403,length(V403)-5)
    End
    If left(Piece3,6) = "COND=(" then do
      Piece3 = delstr(Piece3,1,6)
      Parse Var Piece3  V404 ")" Piece3
    End
    If left(Piece3,7) = "REGION=" then do
      Parse Var Piece3 V405 "," Piece3
      V405 = right(V405,length(V405)-7)
    End
  End
  V499 = V499||Piece3                  /* Whatever is left */
Return

/*------------*/
/* Write the job information to the array */
/*------------*/
Proc2402:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "401"||V401
    OpCtr = OpCtr + 1 ; OData.OpCtr = "402"||V402
    OpCtr = OpCtr + 1 ; OData.OpCtr = "403"||V403
    OpCtr = OpCtr + 1 ; OData.OpCtr = "404"||V404
    OpCtr = OpCtr + 1 ; OData.OpCtr = "405"||V405
    OpCtr = OpCtr + 1 ; OData.OpCtr = "499"||V499
Return

/*------------*/
/* DD Card    */
/*------------*/
Proc250:
  /* If this is the first card of a set, then the variable
     ExpectingContinuation will be "N". For all other cards, it
     will be 'Y'.                                              */
  If ExpectingContinuation = "N" then DDCard = Record
  Else                                DDCard = DDCard||Piece2

  If right(Record,1) = ',' then ExpectingContinuation = "Y"
  Else do                     /* We have read the final card */
    ExpectingContinuation = "N"
    /* Say "The entire DD statement follows" */
    /* Say DDCard                            */
    Call Proc2501                 /* Parse the exec statement */
    Call Proc2502                 /* Write them to the array  */
  End
Return

/*------------*/
```

```
/* Parse the DD Statement */
/*------------*/
Proc2501:
  /* With the job statement, we parse the whole thing at once.
  We cannot do that with the DD, because of operands that begin in
  a left parenthesis. Therefore, we have to "break off" a piece
  at a time.                                    */

  V501= ""; V502= ""; V503= ""; V504= ""; V505= "";
  V506= ""; V507= ""; V508= ""; V509= ""; V510= "";
  V511= ""; V512= ""; V513= ""; V514= ""; V515= "";
  V516= ""; V517= ""; V518= ""; V519= ""; V520= "";
  V521= "";
  DCBStmt= "";
  V599= ""                                          /* Init vars */

  Parse Var DDCard Piece1 Piece2 Piece3
  V501 = DelStr(Piece1,1,2)           /* DD Name                */
  Piece3 = strip(Piece3)

  Do 20                 /* There shouldn't be more than this */
    If left(Piece3,1) = "," then Piece3 = DelStr(Piece3,1,1)

    If left(Piece3,8) = "SYSOUT=(" then do
      Piece3 = DelStr(Piece3,1,8)
      Parse Var Piece3 V502 ")" Piece3
    End
    If left(Piece3,7) = "SYSOUT=" then do
      Parse Var Piece3 V502 "," Piece3
      V502 = right(V502,length(V502)-7)
    End
    If left(Piece3,5) = "DUMMY" then do
      Parse Var Piece3 V503 "," Piece3
    End
    If left(Piece3,4) = "DSN=" then do
      Parse Var Piece3 V504 "," Piece3
      V504 = right(V504,length(V504)-4)
    End
    If left(Piece3,7) = "DSNAME=" then do
      Parse Var Piece3 V504 "," Piece3
      V504 = right(V504,length(V504)-7)
    End
    If left(Piece3,6) = "DISP=(" then do
      Piece3 = DelStr(Piece3,1,6)
      Parse Var Piece3 V505 ")" Piece3
    End
    If left(Piece3,5) = "DISP=" then do
      Parse Var Piece3 V505 "," Piece3
      V505 = right(V505,length(V505)-5)
    End
    If left(Piece3,5) = "UNIT=" then do
      Parse Var Piece3 V506 "," Piece3
      V506 = right(V506,length(V506)-5)
    End
    If left(Piece3,6) = "SPACE=" then do
      Piece3 = DelStr(Piece3,1,6)    /* Delete the string  */
      Call Proc810;  V507 = Result   /* Call nest isolator */
```

```
    End
    If left(Piece3,5) = "DCB=(" then do
      Piece3 = DelStr(Piece3,1,5)
      Parse Var Piece3 DCBStmt ")" Piece3
      Call Proc2509                  /* Parse the DCB statement */
    End
    If left(Piece3,4) = "DCB=" then do
      Parse Var Piece3 DCBStmt "," Piece3
      DCBStmt = right(DCBStmt,length(DCBStmt)-4)
      Call Proc2509                  /* Parse the DCB statement */
    End
    If left(Piece3,6) = "LABEL=" then do
      Parse Var Piece3 V514 "," Piece3
      V514 = right(V514,length(V514)-6)
    End
    If left(Piece3,7) = "COPIES=" then do
      Piece3 = DelStr(Piece3,1,7)    /* Delete the string  */
      Call Proc810;  V515 = Result   /* Call nest isolator */
    End
    If left(Piece3,5) = "DEST=" then do
      Parse Var Piece3 V516 "," Piece3
      V516 = right(V516,length(V516)-5)
    End
    If left(Piece3,5) = "HOLD=" then do
      Parse Var Piece3 V517 "," Piece3
      V517 = right(V517,length(V517)-5)
    End
    If left(Piece3,6) = "TRTCH=" then do
      Parse Var Piece3 V518 "," Piece3
      V518 = right(V518,length(V518)-6)
    End
    If left(Piece3,8) = "OUTPUT=(" then do
      Piece3 = DelStr(Piece3,1,8)
      Parse Var Piece3 V519 ")" Piece3
    End
    If left(Piece3,7) = "OUTPUT=" then do
      Parse Var Piece3 V519 "," Piece3
      V519 = right(V519,length(V519)-7)
    End
    If left(Piece3,8) = "VOL=SER=" then do
      Parse Var Piece3 V520 "," Piece3
      V520 = right(V520,length(V520)-8)
    End
    If left(Piece3,5) = "FREE=" then do
      Parse Var Piece3 V521 "," Piece3
      V521 = right(V521,length(V521)-5)
    End
  End
  V599 = V599||Piece3                 /* Whatever is left */

  /* Impose my personal styles upon the values here */
  If (left(V505,6) = ",CATLG") | ,
     (left(V505,5) = ",PASS" ) then V505 = "NEW"||V505
Return

/*------------*/
/* Write the job information to the array */
```

```
/*------------*/
Proc2502:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "501"||V501
    OpCtr = OpCtr + 1 ; OData.OpCtr = "502"||V502
    OpCtr = OpCtr + 1 ; OData.OpCtr = "503"||V503
    OpCtr = OpCtr + 1 ; OData.OpCtr = "504"||V504
    OpCtr = OpCtr + 1 ; OData.OpCtr = "505"||V505
    OpCtr = OpCtr + 1 ; OData.OpCtr = "506"||V506
    OpCtr = OpCtr + 1 ; OData.OpCtr = "507"||V507
    OpCtr = OpCtr + 1 ; OData.OpCtr = "508"||V508
    OpCtr = OpCtr + 1 ; OData.OpCtr = "509"||V509
    OpCtr = OpCtr + 1 ; OData.OpCtr = "510"||V510
    OpCtr = OpCtr + 1 ; OData.OpCtr = "511"||V511
    OpCtr = OpCtr + 1 ; OData.OpCtr = "512"||V512
    OpCtr = OpCtr + 1 ; OData.OpCtr = "513"||V513
    OpCtr = OpCtr + 1 ; OData.OpCtr = "514"||V514
    OpCtr = OpCtr + 1 ; OData.OpCtr = "515"||V515
    OpCtr = OpCtr + 1 ; OData.OpCtr = "516"||V516
    OpCtr = OpCtr + 1 ; OData.OpCtr = "517"||V517
    OpCtr = OpCtr + 1 ; OData.OpCtr = "518"||V518
    OpCtr = OpCtr + 1 ; OData.OpCtr = "519"||V519
    OpCtr = OpCtr + 1 ; OData.OpCtr = "520"||V520
    OpCtr = OpCtr + 1 ; OData.OpCtr = "521"||V521
    OpCtr = OpCtr + 1 ; OData.OpCtr = "599"||V599
Return

/*------------*/
/* Parse the DCB Statement */
/*------------*/
Proc2509:
  Parse Var DCBStmt DCBTemp "," DCBStmt
  If Pos('=',DCBTemp) = 0 then V509 = DCBTemp       /* Model DSCB */
  Else DCBStmt = DCBTemp||","||DCBStmt

  Do 20                       /* This should be more than enough */
    If left(DCBStmt,6) = "DSORG=" then do
      Parse Var DCBStmt V510 "," DCBStmt
      V510 = right(V510,length(V510)-6)
    End
    If left(DCBStmt,6) = "RECFM=" then do
      Parse Var DCBStmt V511 "," DCBStmt
      V511 = right(V511,length(V511)-6)
    End
    If left(DCBStmt,6) = "LRECL=" then do
      Parse Var DCBStmt V512 "," DCBStmt
      V512 = right(V512,length(V512)-6)
    End
    If left(DCBStmt,8) = "BLKSIZE=" then do
      Parse Var DCBStmt V513 "," DCBStmt
      V513 = right(V513,length(V513)-8)
    End
  End
Return

/*------------*/
/* Data card  */
/*------------*/
```

```
Proc260:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "601"||JCL.I
Return

/*------------*/
/* Comment Card */
/*------------*/
Proc270:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "701"||Record
Return

/*------------*/
/* Output Card */
/*------------*/
Proc280:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "801"||Record
Return

/*------------*/
/* Unknown card */
/*------------*/
Proc290:
    OpCtr = OpCtr + 1 ; OData.OpCtr = "901"||Record
Return

/*------------*/
/* Write the control card file */
/*------------*/
Proc30:
  OData.0 = OpCtr

  ViewData = false
  If ViewData = true then do
    OPDSN = FixJCL.Data
    "Delete "OPDSN
    "Allocate DD(OutFile) DA("OPDSN") new space(1 1) tracks",
              "LRECL(80)  Block(6160)  recfm(f b) RETPD(0)"

    "ExecIO" OData.0 "DiskW OutFile (STEM OData. FINIS"
    "Free DDNAME(OutFile)"
    Say OpCtr "Records written to "OPDSN
    ADDRESS "ISPEXEC" "View   Dataset("OPDSN")"
  end
Return

/*------------*/
/* Write the fixed JCL */
/*------------*/
Proc40:
  Call Proc401                /* Create the Fixed JCL array */
  Call Proc402                /* Write the array to disk    */
Return

/*------------*/
/* Create the Fixed JCL Array */
/*------------*/
Proc401:
```

```
  OJCLCtr = 0
  Do I = 1 to OData.0
    /* Say "Proc401; i/p=" OData.I */
    RecClass = left(OData.I,1)
    RecID    = left(OData.I,3)
    Text     = DelStr(OData.I,1,3)
    If (RecID = 401) | (RecID = 501) | (RecID = 601) | ,
       (RecID = 701) | (RecID = 801) | (RecID = 901) then do
      /* Write the previous recordset */
      If LastClass = "1" then Call Proc40121 /* Job card */
      If LastClass = "4" then Call Proc40124 /* Step/Exec */
      If LastClass = "5" then Call Proc40125 /* DD Statement */
    End
    LastClass = RecClass
    If Text /= "" then do
      /* Set values */
      If RecClass = "1" then Call Proc40111  /* Job card */
      If RecClass = "4" then Call Proc40114  /* Step/Exec */
      If RecClass = "5" then Call Proc40115  /* DD Statement */
    End
    If RecClass = "6" | ,      /* Data: write ALL records */
       RecClass = "7" | ,      /* Comment: write ALL records */
       RecClass = "8" | ,      /* Output: write ALL records */
       RecClass = "9" then do  /* Unknown: write ALL records */
      OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = Text
      Iterate
    End
  End

  /* Write the final class; it's sitting in core */
  LastClass = left(OData.OJCLCTR,1)
  If LastClass = "1" then Call Proc40121 /* Job card */
  If LastClass = "4" then Call Proc40124 /* Step/Exec */
  If LastClass = "5" then Call Proc40125 /* DD Statement */
  If LastClass = "7" then Call Proc40127 /* Comment      */
Return

/*------------*/
/* Clear Values */
/*------------*/
ClearValues:
  V101=" "; V102=" "; V103=" "; V104=" "; V105=" "
  V106=" "; V107=" "; V199=" ";

  V401=" "; V402=" "; V403=" "; V404=" "; V405=" "
  V499=" ";

  V501=""; V502=""; V503=""; V504=""; V505=""
  V506=""; V507=""; V508=""; V509=""; V510=""
  V511=""; V512=""; V513=""; V514=""; V515=""
  V516=""; V517=""; V518=""; V519=""; V520=""
  V521="";
  V599="";
  SOLine = ""; USLine = ""; DCBTemp = "" ; DDLine4 = ""
Return

/*------------*/
```

```
/* Process Job card */
/*------------*/
Proc40111:
  If RecID = "101" then V101 = Text
  If RecID = "102" then V102 = Text
  If RecID = "103" then V103 = Text
  If RecID = "104" then V104 = "MSGLEVEL="||Text
  If RecID = "105" then V105 = "MSGCLASS="||Text
  If RecID = "106" then V106 = "CLASS="||Text
  If RecID = "107" then V107 = "NOTIFY="||Text
  If RecID = "199" then V199 = Text
Return

/*------------*/
/* Process Step/Exec card */
/*------------*/
Proc40114:
  If RecID = "401" then V401 = Text
  If RecID = "402" then V402 = "PGM="||Text
  If RecID = "403" then V403 = "PARM='"Text"'"
  If RecID = "404" then V404 = "COND=("Text")"
  If RecID = "405" then V405 = "REGION="||Text
  If RecID = "499" then V499 = Text
Return

/*------------*/
/* Process DD Card */
/*------------*/
Proc40115:
  If RecID = "501" then V501 = Text

  /* Construct the SYSOUT line */
  If RecID = "502" then do
    If Text = "," then Text = "(,)"
    V502 = "SYSOUT="Text
    SOLine = SOLine||V502
  End
  If RecID = "519" then do
    If Pos(",",Text) > 0 then Text = "("Text")"
    SOLine = SOLine",OUTPUT="Text
  End
  If RecID = "521" then do
    If Pos(",",Text) > 0 then Text = "("Text")"
    SOLine = SOLine",FREE="Text
  End

  If RecID = "503" then V503 = Text
  If RecID = "504" then V504 = "DSN="Text

  If RecID = "505" then do
    If Pos(',',Text) = 0 then V505 = "DISP="Text
    else                      V505 = "DISP=("Text")"
  End

  /* Construct the UNIT and SPACE line */
  If RecID = "506" then USLine = USLine"UNIT="Text
  If RecID = "507" then USLine = USLine",SPACE="Text
```

```
  If RecID = "508" then USLine = USLine",AVGREC="Text
  If RecID = "520" then USLine = USLine",VOL=SER="Text
  If left(USLine,1) = "," then USLine = DelStr(USLine,1,1)

  If RecID = "509" then DCBTemp = DCBTemp||Text
  If RecID = "510" then DCBTemp = DCBTemp",DSORG="Text
  If RecID = "511" then DCBTemp = DCBTemp",RECFM="Text
  If RecID = "512" then DCBTemp = DCBTemp",LRECL="Text
  If RecID = "513" then DCBTemp = DCBTemp",BLKSIZE="Text
  If left(DCBTemp,1) = "," then DCBTemp = DelStr(DCBTemp,1,1)

  /* Construct the "DD Line 4"         */
  If RecID = "514" then DDLine4 = DDLine4"LABEL="Text
  If RecID = "515" then DDLine4 = DDLine4",COPIES="Text
  If RecID = "516" then DDLine4 = DDLine4",DEST="Text
  If RecID = "517" then DDLine4 = DDLine4",HOLD="Text
  If RecID = "518" then DDLine4 = DDLine4",TRTCH="Text
  If left(DDLine4,1) = "," then DDLine4 = DelStr(DDLine4,1,1)

  If RecID = "599" then V599 = Text
Return

/*------------*/
/* Write the Job card */
/*------------*/
Proc40121:
  V101 = left(V101||spaces,8)
  JC1 = "//"V101" JOB "||V102",'"V103"',"
  JC2 = "//             "V104","V105","V106","V107
  OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC1
  If V199 = "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC2
  End
  Else do
    JC2 = JC2||","
    JC3 = "//             "V199
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC2
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC3
  End
  Call ClearValues;
Return

/*------------*/
/* Write the Step/Exec card */
/*------------*/
Proc40124:
  V401 = left(V401||spaces,8)
  JC1 = "//"V401" EXEC "||V402
  If V404 /= "" then JC1 = JC1","V404
  If V405 /= "" then JC1 = JC1","V405
  If V403 = "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC1
  end
  else do            /* There IS a parm field  */
    if (length(JC1) + 1 + length(V403)) < 72 then do  /* same line */
      JC1 = JC1","V403
      OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC1
```

```
    end
    else do
      JC1 = JC1||","
      JC2 = "//             "V403
      OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC1
      OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = JC2
    End
  End
  Call ClearValues;
Return

/*------------*/
/* Write the DD Card */
/*------------*/
Proc40125:
  C1=""; C2=""; C3=""; C4=""; C5=""; C6=""
  V501 = left(V501||spaces,8)           /* DDName */
  C1 = strip("//"V501" DD  "SOLine||V503||V504)

  /* There is almost NEVER a good reason to say DISP=(OLD,DELETE).
     At UMB, OLD,DELETE is used too frequently. Therefore,
     impose my personal preferences and replace those.          */
  If left(V505,9) = "DISP=(OLD" then
    V505 = "DISP=SHR"||right(V505,length(V505)-9)
  If left(V505,8) = "DISP=OLD" then
    V505 = "DISP=SHR"||right(V505,length(V505)-8)
  If left(V505,15) = "DISP=SHR,DELETE" then
    V505 = "DISP=SHR"

  If Pos(',',DCBTemp) > 0 then DCBTemp = "DCB=("DCBTemp")"

  If ((length(C1) + 1 + length(V505)) < 72) & ,
     (V505 = "DISP=SHR") then do
    C1 = C1","V505
    C2 = strip("//         "USLine)
    C3 = strip("//         "DCBTemp)
    C4 = strip("//         "DDLine4)
    C5 = strip("//         "V599)
  end
  else do
    C2 = strip("//         "V505)
    C3 = strip("//         "USLine)
    C4 = strip("//         "DCBTemp)
    C5 = strip("//         "DDLine4)
    C6 = strip("//         "V599)
  end

  Do 4
    If length(C2) < 3 then do   /* The 2nd card is completely blank */
      C2 = C3; C3 = C4; C4 = C5; C5 = C6; C6 = ""
    end
  end

  /* See if we can (should) combine any JCL lines */
  If length(strip(C1)) < 16 then do
    C2 = DelStr(C2,1,11)
    C1 = C1"  "C2
```

```
    C2 = C3; C3 = C4; C4 = C5; C5 = C6; C6 = ""
  End

  /* See which lines need continuation commas  */
  If length(C2) > 2 then C1 = C1||","
  else                   C2 = ""
  If length(C3) > 2 then C2 = C2||","
  else                   C3 = ""
  If length(C4) > 2 then C3 = C3||","
  else                   C4 = ""
  If length(C5) > 2 then C4 = C4||","
  else                   C5 = ""
  If length(C6) > 2 then C5 = C5||","
  else                   C6 = ""

  OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C1
  If C2 /= "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C2
  End
  If C3 /= "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C3
  End
  If C4 /= "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C4
  End
  If C5 /= "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C5
  End
  If C6 /= "" then do
    OJCLCtr = OJCLCtr + 1; OJCL.OJCLCtr = C6
  End
  Call ClearValues;
Return

/*------------*/
/* Write the Fixed JCL Array to Disk */
/*------------*/
Proc402:
  OJCL.0 = OJCLCtr
  OPDSN = FixJCL.JCL
  If OJCLCtr = 0 then do
    Say "There are no records to write to" OPDSN"!"
    Return
    Exit
  End
  "Delete" OPDSN
  "Allocate DD(OutFile) DA("OPDSN") new space(1 1) tracks",
            "LRECL(80)  Block(6160)  recfm(f b) RETPD(0)"

  "ExecIO" OJCL.0 "DiskW OutFile (STEM OJCL. FINIS"
  "Free DDNAME(OutFile)"
  Say OJCLCtr "Records written to "OPDSN
  ADDRESS "ISPEXEC" "View Dataset("OPDSN")"
Return

/*------------*/
/* Nested operand isolator */
```

```
/*------------*/
/* This routine will isolate operands that are nested within
   parenthesis. It is used mainly for COPIES= and SPACE=.
   Example: Piece3=(1,(1,1,1,1)),DEST=U98,HOLD=NO,
   This routine will split Piece3 into:
   (1,(1,1,1,1)) and DEST=U98,HOLD=NO,                */
Proc810:
  If left(Piece3,1) = "(" then do       /* May be nested */
    ReturnStr = "(" ; Level = 1; Index = 2
    Do Until Level = 0
      If substr(Piece3,Index,1) = "(" then Level = Level + 1
      If substr(Piece3,Index,1) = ")" then Level = Level - 1
      ReturnStr = ReturnStr||substr(Piece3,Index,1)
      Index = Index + 1
    End
    Piece3 = DelStr(Piece3,1,Index)
  End
  Else Parse var Piece3 ReturnStr "," Piece3 /* No nesting */
Return ReturnStr

/*------------*/
/* End-of-job Processing */
/*------------*/
ProcEOJ:
Return
```

FX - File name cross-reference

This exec will convert JCL into a list of stepnames and datasetnames, that can be used as somewhat of a cross-reference.

```
/* FX - File Cross-Reference - REXX Exec  */
/* This exec will read a set of job control, parse it, and       */
/* create a file, one record per datasetname, as follows:        */
/*  1- 8  8  Jobname                              */
/*  9-16  8  Stepname                             */
/* 17-24  8  DDName                               */
/* 25-78 54  Datasetname (allowing room for PDS member name)     */
/* 79-81  3  Disposition (NEW, OLD, MOD)         */

/*------------- Main Body of Program ----------------------------*/
ARG IPDSN
Call Pgm_Init

Do Forever
    Call ReadRec                    /* Read rec into stack; count */
    If IPEOF = "YES" then Leave
    Pull Record                     /* Get it from the stack      */
    Call IdentifyRecord             /* See what kind it is        */
    Call ProcessRecord              /* Process it                 */
end

Call ProcEOJ                        /* EOJ Processing             */
/*  ADDRESS "ISPEXEC" "Browse Dataset("OPDSN")"   */
Exit
/*-------------------------------------------------------------------*/

/*------------*/
/* Program Initialization */
/*------------*/
Pgm_Init:
"DelStack"
If IPDSN = "" then do
    Say "Command Type:

Syntax: FX DSN"
    Exit
end

"Alloc DDN(InFile) DSN("IPDSN") SHR"
If RC <> 0 then do
    Say "I could not allocate "IPDSN". Sorry."
    Exit
end

Say "FX Working on " IPDSN ": " sysvar(SYSUID) Date(U) Time() "..."

OPDSN = "FX.DATAFILE"
"Free FI(OutFile)"
"Alloc DD(OutFile) DA("OPDSN") MOD space(15 15) tracks ",
      "Lrecl(81) Block(6156) Recfm(F B)"
```

```
If RC <> 0 then do
    Say "I could not allocate "OPDSN". Sorry."
    Exit
end

"NewStack"
IPEOF = "NO"                          /* Input EOF Switch           */
RecType = "   "                       /* Record Type                */
Spaces = "                                      "
Spaces = Spaces || Spaces             /* Now it's 72 spaces         */
JobName = "(Unk)"                     /* Job Name                   */
StepName= "(Unk)"                     /* Step Name                  */
DDName  = "(Unk)"                     /* DDName                     */
RecCount = 0                          /* Total Records              */
Type1Ctr = 0                          /* First JCL card of a set    */
Type11Ctr = 0                         /* Job Cards                  */
Type12Ctr = 0                         /* DD Cards                   */
Type13Ctr = 0                         /* EXEC cards                 */
Type14Ctr = 0                         /* JES (output, message) cards*/
Type15Ctr = 0                         /* Other JCL cards: first card*/
Type2Ctr = 0                          /* JCL continuations          */
Type3Ctr = 0                          /* Comment card counter       */
Type41Ctr = 0                         /* Data card counter          */
Type42Ctr = 0                         /* end of Data card counter   */
Type5Ctr = 0                          /* end of job card            */
TypeUCtr = 0                          /* Unknown                    */
OpRecCtr = 0                          /* Output Records             */
DSNFound = 0                          /* DSN Found                  */
DispFound = 0                         /* Disp Found                 */

/*------------*/
ReadRec:
/*------------*/
    "EXECIO 1 DiskR Infile"           /* Add the I/P rec to the stack */
    If RC <> 0 then do
        IPEOF = "YES"
        "EXECIO 0 DiskR Infile (Finis"  /* Close the input file */
    end
    Else RecCount = RecCount + 1      /* Count the records            */
    Return ""

/*------------*/
IdentifyRecord:
/*------------*/
    Part1 = Substr(record,1,2)
    Part2 = Substr(record,3,1)
    Part3 = Substr(record,3,71)
    Spaces2 = Substr(Spaces,3,71)
    If Substr(Record,1,3) = "//*"       then Call Proc_Type3
    Else If Part1 = '//' & Part2 /= ' ' then Call Proc_Type1
    Else if Substr(Record,1,3) = "// "  then Call Proc_Type2
    Else If Substr(Record,1,3) = "/* "  then Call Proc_Type42
    Else If Part1 = '//' & Part3 = Spaces2 then
                                        Call Proc_Type5
    Else if Substr(Record,1,1) /= "/"   then Call Proc_Type41
    Else if Substr(Record,1,9) = "/*MESSAGE" then ,
        Type14Ctr = Type14Ctr + 1
```

```
    Else if Substr(Record,1,3) = "/*$" then ,
        Type14Ctr = Type14Ctr + 1
    Else if Substr(Record,1,7) = "/*ROUTE" then ,
        Type14Ctr = Type14Ctr + 1
    Else if Substr(Record,1,8) = "/*NOTIFY" then ,
        Type14Ctr = Type14Ctr + 1
    Else Call Proc_Type_Unk
    Return

Proc_Type1:    /*  - First JCL cards read         */
    RecType = "1  "
    Type1Ctr = Type1Ctr + 1
    FirstBlk = Pos(' ',Record)
    TempRecord = Delstr(Record,1,FirstBlk)
    TempRecord = Strip(TempRecord,L)
    FirstBlk = Pos(' ',TempRecord)
    JCLType = SubStr(TempRecord,1,FirstBlk-1)
    If JCLType = "JOB"         then do
        RecType = "11 "
        Type11Ctr = Type11Ctr + 1
        FirstBlk = Pos(' ',Record)
        JobName  = SubStr(Record,3,FirstBlk-1)
    end
    else If JCLType = "DD"     then do
        RecType = "12 "
        Type12Ctr = Type12Ctr + 1
        FirstBlk = Pos(' ',Record)
        DDName = SubStr(Record,3,FirstBlk-1)
        Call FindDSN; Call FindDisp
    end
    else If JCLType = "EXEC"   then do
        RecType = "13 "
        Type13Ctr = Type13Ctr + 1
        FirstBlk = Pos(' ',Record)
        StepName  = SubStr(Record,3,FirstBlk-1)
    end
    else If JCLType = "OUTPUT" then do
        RecType = "14 "
        Type14Ctr = Type14Ctr + 1
    end
    else do
        RecType = "15 "
        Type15Ctr = Type15Ctr + 1
    end
Return

Proc_Type2:    /*- JCL continuation cards read    */
    RecType = "2  "
    Type2Ctr = Type2Ctr + 1
    Call FindDSN; Call FindDisp
Return

Proc_Type3:    /*- Comment cards read             */
    RecType = "3  "
    Type3Ctr = Type3Ctr + 1
Return
```

```
Proc_Type41:   /*- Data cards read                */
    RecType = "41 "
    Type41Ctr = Type41Ctr + 1
Return

Proc_Type42:   /*- End of Data cards read         */
    RecType = "42 "
    Type42Ctr = Type42Ctr + 1
Return

Proc_Type5:    /*- end of job cards read          */
    RecType = "5  "
    Type5Ctr = Type5Ctr + 1
Return

Proc_Type_Unk: /* Unknown type                    */
    RecType = "?  "
    TypeUCtr = TypeUCtr + 1
    Say "Unknown; number " RecCount " was read; Type " RecType,
        " record follows:"
    Say Record
    Say "------------------------------------------------------------"
Return

/* Find the datasetname */
FindDSN:
    DSNLoc = Index(Record,"DSN=")
    If DSNLOC > 0 then do
        TempRec = Delstr(Record,1,DSNLOC+3) /* Delete past dsn= */
        FirstBlk = Pos(' ',TempRec)
        FirstCom = Pos(',',TempRec)
        If FirstCom = 0 then FirstCom = 80  /* In case no comma */
        If FirstBlk < FirstCom then EndPos = FirstBlk
        Else                        EndPos = FirstCom
        If EndPos = 0 then do
            Say "FindDSN error: " Record
        end
        DSN = SubStr(TempRec,1,EndPos-1)
        DSN = substr(DSN||Spaces,1,54)
        DSNFound = DSNFound + 1
        OPRecPending = "YES"
    end
Return

/* Find the dataset disposition */
FindDisp:
    DispLoc = Index(Record,"DISP=")
    If DispLOC > 0 then do
        TempRec = Delstr(Record,1,DispLOC+4) /* Delete past Disp= */
        FirstBlk = Pos(' ',TempRec)
        EndPos = FirstBlk
        Disp = SubStr(TempRec,1,EndPos-1)
        DispFound = DispFound + 1
        OPRecPending = "YES"
        If substr(Disp,1,2) = "(," then Disp = "NEW"
        else If substr(Disp,1,5) = "SHARE" then disp = "SHR"
        else If substr(Disp,1,4) = "(OLD"  then disp = "OLD"
```

```
        else If substr(Disp,1,4) = "(MOD"  then disp = "MOD"
        else If substr(Disp,1,4) = "(NEW"  then disp = "NEW"
    end
Return

/*------------*/
/* Process the Record */
/*------------*/
ProcessRecord:
If OPRecPending = "YES" then do
    If Substr(RecType,1,1) /= '2' then do
        Jobname= substr(Jobname||Spaces,1,8)
        Stepname= substr(Stepname||Spaces,1,8)
        DDName  = substr(DDName||Spaces,1,8)
        OPRec = Jobname||StepName||DDName||DSN||Disp
        OpRecPending = "NO"
        OpRecCtr = OpRecCtr + 1
        Push OpRec
        "EXECIO" 1 "DiskW OutFile"
    end
end
Return

/*------------*/
/* End-of-job Processing */
/*------------*/
ProcEOJ:
    "DelStack"
    "Free DDNAME(InFile)"
    "EXECIO" 0 "DiskW OutFile (Finis"   /* Close the file        */
    Queue "*** End of Job Totals for " IPDSN "***"
    Queue RecCount "records read"
    Queue "  "Type1Ctr   "First JCL cards read"
    Queue "     "Type11Ctr  "- Job cards"
    Queue "     "Type12Ctr  "- DD cards"
    Queue "     "Type13Ctr  "- EXEC cards"
    Queue "     "Type14Ctr  "- JES (OUTPUT, MESSAGE) cards"
    If Type15Ctr > 0 then Queue "     "Type15Ctr  "- other JCL cards"
    Queue "   "Type2Ctr   "JCL continuation cards read"
    Queue "   "Type3Ctr   "Comment cards read"
    Queue "   "Type41Ctr  "Data cards read"
    Queue "   "Type42Ctr  "End of Data cards read"
    Queue "   "Type5Ctr   "end of job cards read"
    If TypeUCtr > 0 then Queue "   "TypeUCtr   "Unknown cards read"
    If TypeUCtr > 0 then Say,
        "Warning: " TypeUCtr   "Unknown cards read"
    Queue OpRecCtr "records written"
    OPDSN = FX.LOGFILE
    "Free FI(LogFile)"
    "Alloc DD(LogFile) DA("OPDSN") MOD space(15 1) tracks ",
          "Lrecl(73) Block(6205) Recfm(F B)"
    If RC <> 0 then do
        Say "I could not allocate "OPDSN". Sorry."
        Exit
    end
    Quantity = queued()
    "EXECIO " Quantity " DiskW LogFile (Finis"
```

```
Return
```

Guess – Guess the Number

This is an example of a game where the computer picks a number, and you have to guess it in the fewest tries possible. You are rewarded with feedback after the game.

```
/* Guess - Guess the Number                                    REXX    */
/* This is a Rexx learning exercise                                    */
/* Guess the computer-generated number in the fewest number of turns.*/

PName = ""                /* Player Name     */
Turn = 1

Say "Welcome to the Guess the Number. I have chose a number between"
Say "000 and 1000, exclusive. See how long it takes you to guess what"
Say "number I have chosen."
Say " "

Say "Player, please tell me your name!"
Pull PName
Upper PName

Redo:
Guesses = 0               /* Number of guesses     */
CNo = Random(1,999)       /* Computer number */
/* Say "The computer picked number " CNo   */
InProgress = Y
PNumber = 0               /* Player number */
Lower = 0
Upper = 1000

Do While InProgress = Y
  Say PName", pick a number between " Lower " and " Upper "."
  ReAsk = N
  Pull PNumber
  If (PNumber <= Lower) | (PNumber >= Upper) then do
    Say "Dummy! I said between " Lower " and "Upper "! Try again!"
    ReAsk = Y
  End
  If ReAsk = Y then
    Turn = 1
  else do
    Guesses = Guesses + 1
    If CNo = PNumber then do
      InProgress = N
      Call Recap
      Leave
    End
    Else do
      If PNumber < CNo then Lower = PNumber
      If PNumber > CNo then Upper = PNumber
    End
  End
End

Say "Again?"
Pull Ans
```

```
Upper Ans
If Ans = Y then signal ReDo
exit

Recap:
  Say "You guessed it," PName"!"
  Select
    When Guesses = 1 then
      Say "One guess! Buy a Powerball ticket, quick!"
    When Guesses = 2 then
      Say "Two guesses! Buy a lottery ticket, quick!"
    When Guesses = 3 then
      Say "Three guesses! Are you sitting on a horseshoe?"
    When Guesses = 4 then
      Say "Four guesses! You got lucky!"
    When Guesses = 5 then
      Say "Five guesses! That's phenomenal!"
    When Guesses = 6 then
      Say "Six guesses! That's excellent!"
    When Guesses = 7 then
      Say "Seven guesses! Very good!"
    When Guesses = 8 then
      Say "Eight guesses! That's pretty good!"
    When Guesses = 9 then
      Say "Nine guesses. That's about average."
    When Guesses = 10 then
      Say "Ten guesses. That's a little under average."
    When Guesses = 11 then
      Say "Eleven guesses? Don't go to the racetrack!"
    When Guesses = 12 then
      Say "Twelve guesses? That's pretty poor!"
    Otherwise
      Say Guesses" guesses! Have you ever heard of a binary search?"
  End

  Adjective = "only"
  Spread = (Upper - Lower)
  If Spread > 10 then Adjective = ""
  If Spread > 25 then Adjective = "a Whopping"
  Say "The spread was" Adjective Spread

  Numeric Digits 4
  Points = Spread * 100/Guesses
  Say "For this game, you get "Points" points."
Return
```

HD - Hex Dump

This command will hex dump a sequential file.

```
/* REXX PROGRAM */
/* HD - HEX DUMP A SEQUENTIAL FILE IN HEX                */
ARG IPDsn NUMRECS OPDsn

/* CHECK COMMAND LINE ARGUMENTS                       */
IF IPDsn = '' THEN DO
    SAY 'COMMAND TYPE:

SYNTAX: HD IPDsn NUMRECS OPDSN'
    EXIT
END

/* Some users have turned off their Profile Prefix.   */
/* If that is the case with this user, then prefix the OP DSN with */
/* his userid                                 */
If SYSVAR(SYSPREF) = "" then
    DSNPref = USERID()||"."
Else
    DSNPref = ""

IF OPDsn = '' THEN DO
    OPDsn = DSNPREF || "HD.OUTLIST"
END

IF NUMRECS = '' THEN
            NUMRECS = 999999

/* SET OUR CONSTANTS */
DFL = 100                                    /* FRAGMENT LENGTH OF ONE LINE
*/
TESTING = N                                  /* TEST CODE WILL BE EXECUTED
*/
SCALE1 = '         1         2         3         4         5         6'
scale1 = scale1 || '         7         8         9        10'
SCALE2 = COPIES('....5....0',10)

SAY 'WORKING...'

DUMMY = LISTDSI(IPDsn)
INFLRECL = SYSLRECL
IF INFLRECL > DFL THEN DO
    RECSEGS = TRUNC(INFLRECL/DFL,0)  /* NO. OF RECORD SEGMENTS */
    IF INFLRECL/DFL > TRUNC(INFLRECL/DFL,0) THEN
        RECSEGS = RECSEGS + 1
    RECSEGL = DFL                    /* SEGMENT LENGTH         */
    RECSEGLAST = INFLRECL // RECSEGL /* LAST SEGMENT LENGTH */
END
ELSE DO
    RECSEGS = 1                      /* NO. OF RECORD SEGMENTS */
    RECSEGL = INFLRECL               /* SEGMENT LENGTH         */
    RECSEGLAST = INFLRECL            /* SEGMENT LENGTH         */
END
```

```
SAY '*** HD - HEXDUMP, VERS 1.0 ***'
SAY 'IPDsn:  ' IPDsn '; LRECL = ' INFLRECL
SAY 'OPDsn:  ' OPDsn
IF TESTING = Y THEN DO
  SAY 'NO. OF SEGMENTS TO DISPLAY FOR EACH RECORD: ' RECSEGS
  SAY 'SEGMENT LENGTH:      ' RECSEGL
  SAY 'LAST SEGMENT LENGTH: ' RECSEGLAST
END

"ALLOCATE DDNAME(INFILE) DSN(" IPDsn ") SHR "
"DELETE " OPDsn
"ALLOCATE DDNAME(OUTFILE) DSN(" OPDsn ") NEW SPACE(20,20)" ,
    "BLOCK(6171) UNIT(SYSDA) LRECL(121) RECFM(F B)"
"NEWSTACK"
"EXECIO * DISKR INFILE (STEM INFILE. FINIS"
SAY 'INPUT FILE SIZE:' INFILE.0 'RECORDS.'
QUEUE ' DUMP OF DSN:' IPDsn
IF NUMRECS > INFILE.0 THEN
    NUMRECS = INFILE.0
SAY 'DUMPING ' NUMRECS 'RECORDS'
DO I = 1 TO NUMRECS
    ISTR = FORMAT(I,3,0)
    DO J = 1 TO RECSEGS
        SSTR = FORMAT(J,1,0)
        RC = ((J-1)*DFL)+1
        SC = RC // 100
        IF J = RECSEGS THEN
            THISRSL = RECSEGLAST
        ELSE
            THISRSL = RECSEGL
        QUEUE '           'SUBSTR(SCALE1,SC,THISRSL)
        QUEUE '           'SUBSTR(SCALE2,SC,THISRSL)
        THISPORTION = SUBSTR(INFILE.I,RC,THISRSL)
        QUEUE ISTR'.'SSTR 'CHAR' THISPORTION

        /* WORK ON THE ZONE PORTION */
        WORKPORTION = C2X(THISPORTION)
        THISPORTIONZONE = ' '
        DO K = 1 TO (THISRSL*2) BY 2
            THISPORTIONZONE = THISPORTIONZONE SUBSTR(WORKPORTION,K,)
            THISPORTIONZONE = SPACE(THISPORTIONZONE,0)
        END
        QUEUE ISTR'.'SSTR' ZONE' THISPORTIONZONE

        /* WORK ON THE NUMERIC PORTION */
        WORKPORTION = C2X(THISPORTION)
        THISPORTIONNUMR = ' '
        DO K = 2 TO (THISRSL*2) BY 2
            THISPORTIONNUMR = THISPORTIONNUMR SUBSTR(WORKPORTION,K,)
            THISPORTIONNUMR = SPACE(THISPORTIONNUMR,0)
        END
        QUEUE ISTR'.'SSTR' NUMR' THISPORTIONNUMR
    END
    QUEUE ' '
    HOW_MANY = QUEUED()
    "EXECIO" HOW_MANY "DISKW OUTFILE"
```

```
END
"EXECIO" 0 "DISKW OUTFILE (FINIS"          /* CLOSE THE FILE */
"FREE DDNAME(INFILE OUTFILE)"
SAY 'DUMP COMPLETE. CHECK ' OPDsn

"ISPEXEC BROWSE DATASET(" OPDsn
```

INIT - Establish my TSO environment

I use this Rexx exec to establish my TSO environment: allocate my Rexx libraries, tellme what the temperature is, etc.

```
/* Init - TSO Session Initialization  - REXX EXEC  */

Address TSO
"Free  Fi(SYSEXEC)"
"Alloc Fi(SYSEXEC) DA('GRUND.TSTREXX.EXEC' 'GRUND.REXX.EXEC') SHR "

Say;Say;Say   /* Start at the top of the screen */
Say "Hello, and welcome to TSO, courtesy of David Grund's INIT EXEC."

Say "Today is" Date(W) Date(U) "; julian is " substr(Date(J),3,3)

MoNum = substr(Date(U),1,2)
If Monum =  1 then Do; Low =  0; High = 55; end
If Monum =  2 then Do; Low =  0; High = 60; end
If Monum =  3 then Do; Low = 15; High = 65; end
If Monum =  4 then Do; Low = 35; High = 80; end
If Monum =  5 then Do; Low = 45; High = 85; end
If Monum =  6 then Do; Low = 50; High = 90; end
If Monum =  7 then Do; Low = 55; High = 95; end
If Monum =  8 then Do; Low = 55; High = 95; end
If Monum =  9 then Do; Low = 50; High = 90; end
If Monum = 10 then Do; Low = 30; High = 85; end
If Monum = 11 then Do; Low = 10; High = 75; end
If Monum = 12 then Do; Low =  0; High = 60; end

Temp = Random(Low,High)
Say "The temperature right now is " Temp

TSOMSG = "I executed your INIT exec on " || Date(W) Date(U) "at" Time(C)
TSOMSG = TSOMSG || ", Dave"
"Send '"TSOMSG || "' U(GRUND) LOGON NoWait"

InitSPF
```

INITSPF - Establish my ISPF environment

I use this command to establish my ISPF environment, which is mainly to allocate my test ISPF libraries in front of the production ones.

```
/* InitSPF -         REXX EXEC */
/* Initialize personal ISPF environment     */

UserID = SYSVAR(SYSUID)
Say "Initializing personal ISPF environment..."
Address TSO
/* Allocate panel libraries */
"Free  Fi(ISPPLIB)"
"Alloc Fi(ISPPLIB) DA('GRUND.ISPF.PANELS'  "  ,
                    " 'ISR.IBM.ISRPLIB'      "  ,
                    " 'ISP.IBM.ISPPLIB'     ,"  ,
                    " 'ISR.PRODUCT.ISRPLIB'  ) SHR "

/* Allocate message libraries */
"Free  Fi(ISPMLIB)"
"Alloc Fi(ISPMLIB) DA('GRUND.ISPF.MESSAGES'"  ,
                    " 'ISR.UP.ISRMLIB'       "  ,
                    " 'ISR.IBM.ISRMLIB'      "  ,
                    " 'ISP.IBM.ISPMLIB'     ,"  ,
                    " 'ISR.PRODUCT.ISRMLIB'  ) SHR "

/* Allocate input table libraries */
"Free  Fi(ISPTLIB)"
"Alloc Fi(ISPTLIB) DA('GRUND.ISPF.TABLES'  "  ,
                    " 'ISR.IBM.ISRTLIB'      "  ,
                    " 'ISP.IBM.ISPTLIB'      ) SHR "

/* Allocate output table libraries */
"Free  Fi(ISPTABL)"
"Alloc Fi(ISPTABL) DA('GRUND.ISPF.TABLES') SHR"

/* Allocate skeleton libraries */
"Free  Fi(ISPSLIB)"
"Alloc Fi(ISPSLIB) DA('GRUND.ISPF.SKELETON'"  ,
                    " 'ISR.IBM.ISRSLIB'      "  ,
                    " 'ISP.IBM.ISPSLIB'      "  ,
                    " 'ISR.PRODUCT.ISRSLIB'  ) SHR "
Say "...Done"
```

JOBCARD - Create a jobcard

I use this exec to add a standard job card to my JCL

```
/* JOBCARD - ISPF Edit Macro (REXX EXEC)  */
ADDRESS "ISREDIT" "MACRO PROCESS"
address "ISREDIT" "(XDSN)=DATASET"
address "ISREDIT" "(XMEM)=MEMBER"
J01 =  "//"sysvar(sysuid)"A  JOB (accounting info),@DAVID GRUND@, "
J021 = "//        MSGLEVEL=1,MSGCLASS=C,CLASS=C,PASSWORD=,TIME=1,       "
J022 = "//        USER=" || Sysvar(sysuid) || ",NOTIFY=" || Sysvar(Sysuid)
J031 = "//*-----------------------------------"
J032 = "-----------------------------------*"
J03 = J031 || J032
J04 = "//* CREATED BY JOBCARD MACRO" date(U) time()
J05 = "//* THIS JOB SUBMITTED FROM &XDSN(&XMEM)"
J06 = "//*   ** JOB STEPS **"
J07 = "//* STEP010  - IEHGOD00 - DO ANYTHING YOU WISH"
J08 = "//* "
J09 = "//       JCLLIB ORDER=(GRUND.INCLUDE.JCL)"
J10 = "//STEP010 EXEC PGM=IEHGOD00,REGION=640K"
address "ISREDIT" "LINE_AFTER 00 = " "'"J01"'"
address "ISREDIT" "LINE_AFTER 01 ="  "'"J021"'"
address "ISREDIT" "LINE_AFTER 02 ="  "'"J022"'"
address "ISREDIT" "LINE_AFTER 03 ="  "'"J03"'"
address "ISREDIT" "LINE_AFTER 04 ="  "'"J04"'"
address "ISREDIT" "LINE_AFTER 05 ="  "'"J05"'"
address "ISREDIT" "LINE_AFTER 06 ="  "'"J03"'"
address "ISREDIT" "LINE_AFTER 07 ="  "'"J06"'"
address "ISREDIT" "LINE_AFTER 08 ="  "'"J07"'"
address "ISREDIT" "LINE_AFTER 09 ="  "'"J03"'"
address "ISREDIT" "LINE_AFTER 10 ="  "'"J09"'"
address "ISREDIT" "LINE_AFTER 11 ="  "'"J08"'"
address "ISREDIT" "LINE_AFTER 12 ="  "'"J03"'"
address "ISREDIT" "LINE_AFTER 13 ="  "'"J07"'"
address "ISREDIT" "LINE_AFTER 14 ="  "'"J03"'"
address "ISREDIT" "LINE_AFTER 15 ="  "'"J10"'"

/* NOW PUT ASTERISKS IN COL 71 IN LINES 4 THRU 8 */
ADDRESS "ISREDIT" "LABEL 4 = .LSTART "
ADDRESS "ISREDIT" "LABEL 8 = .LEND   "
ADDRESS "ISREDIT" "CHANGE ' ' '*' 71 .LSTART .LEND ALL"
ADDRESS "ISREDIT" "RESET"

/* I can't get apostrophes around the name to begin with */
/* because of syntax restrictions. So do it now.         */
ADDRESS "ISREDIT" "LABEL 1 = .LONLY "
ADDRESS "ISREDIT" "CHANGE '@' ''' .LONLY .LONLY ALL"

ADDRESS "ISREDIT" "Cursor = 1 0"

address "ISREDIT" "LINE_AFTER 0 = NoteLine",
  "'"Jobcard generated."'"
address "ISREDIT" "LINE_AFTER 15 = NoteLine",
  "'-------------------------------------------------------------'"
```

LA - List TSO allocations

This Rexx exec will list the TSO allocations and write them to a dataset. It will then edit that dataset using ISPF macro LAE (included below).

```
/* LA - Create a List of TSO Allocations - Rexx Exec */

Dummy = OutTrap("output_line.","*")
"LISTA SY ST"
NumLines = OutPut_Line.0
Say NumLines "lines were created"
Dummy = OutTrap("OFF")

/* Move the line with the DDNAME above the first datasetname
   that it is concatenated to. It is currently below.          */
Do I = 1 to NumLines
    Piece1 = SubStr(OutPut_Line.I,1,2)
    Piece2 = SubStr(OutPut_Line.I,3,1)
    Piece3 = SubStr(OutPut_Line.I,12,4)
    If Piece1 = '  ' & ,
       Piece2 ¬= ' ' & ,
       Piece3 = 'KEEP' then do
        J = I - 1
        SaveLine      = OutPut_Line.I
        Output_Line.I = OutPut_Line.J
        Output_Line.J = SaveLine
    end
end

/* Many users have the TSO profile set to NoPrefix */
/* Account for that here.                          */
If SYSVAR(SYSPREF) = '' then do
    "profile prefix(" userid() ")"
    TurnPrefixBackOff = 1
end
Else
    TurnPrefixBackOff = 0

"Delete la.list"
"Allocate DD(LAList) DA(LA.List) new space(1 1) tracks",
          "LRECL(80)  Block(5600)  recfm(f b) RETPD(0)"

"ExecIO" OutPut_line.0 "DiskW LAList (STEM OutPut_Line. FINIS"
"Free DDNAME(LaList) DA(La.List)"

ADDRESS "ISPEXEC" "EDIT Dataset(La.List)  Macro(LAE)"
ADDRESS "TSO"

If TurnPrefixBackOff = 1 then
    "Profile Noprefix"
```

LAE - ISPF Edit macro for LA

```
/* LAE - Edit macro for LA           - Rexx Exec */

ADDRESS "ISREDIT" "MACRO PROCESS"
ADDRESS "ISREDIT" "EXCLUDE ALL --DDNAME 1"
ADDRESS "ISREDIT" "EXCLUDE ALL '           keep' 1 "
ADDRESS "ISREDIT" "Delete ALL X"
ADDRESS "ISREDIT" "C 'KEEP' '--------------' word all 12"
```

LOTTERY - Pick Lottery Numbers

```
/* Lottery - Pick a Lottery Number  - Rexx Exec                 */
/* This program will pick a lottery number for you              */

Arg Game

Call Init                                    /* Init Program          */
Call Main                                    /* Mainline              */

Exit

/*------------*/
/* Program Initialization */
/*------------*/
Init:
    If Game = "" then do
      Say "Which game do you want numbers for?"
      Say "The choices are: 1)Pick3 2)PowerBall 3)Show Me Five"
      Pull Game
    End

    If (Game = 1) | (Game = 2) | (Game = 3) then Return
    Say Game "is an invalid selection!"
    Exit

Return

/*------------*/
/* Mainline   */
/*------------*/
Main:
    /* Pick 3 */
    If Game = 1 then do
        Number1 = Random(0,9)
        Number2 = Random(0,9)
        Number3 = Random(0,9)
        Say "The Pick3 numbers I have selected are:",
            Number1 Number2 Number3
    End

    /* PowerBall */
    If Game = 2 then do
        Number1 = Random(1,49)

        Number2 = Number1
        Do While Number2 = Number1
          Number2 = Random(1,49)
        End

        Number3 = Number1
        Do While (Number3 = Number1) | (Number3 = Number2)
          Number3 = Random(1,49)
        End

        Number4 = Number1
```

```
        Do While (Number4 = Number1) | (Number4 = Number2) | ,
                 (Number4 = Number3)
          Number4 = Random(1,49)
        End

        Number5 = Number1
        Do While (Number5 = Number1) | (Number5 = Number2) | ,
                 (Number5 = Number3) | (Number5 = Number4)
          Number5 = Random(1,49)
        End

        Number6 = Random(1,42)

        Say "The Powerball numbers I have selected are:",
            Number1 Number2 Number3 Number4 Number5 "PB:"Number6
    End

    /* Show Me Five */
    If Game = 3 then do
        Number1 = Random(1,30)

        Number2 = Number1
        Do While Number2 = Number1
          Number2 = Random(1,30)
        End

        Number3 = Number1
        Do While (Number3 = Number1) | (Number3 = Number2)
          Number3 = Random(1,30)
        End

        Number4 = Number1
        Do While (Number4 = Number1) | (Number4 = Number2) | ,
                 (Number4 = Number3)
          Number4 = Random(1,30)
        End

        Number5 = Number1
        Do While (Number5 = Number1) | (Number5 = Number2) | ,
                 (Number5 = Number3) | (Number5 = Number4)
          Number5 = Random(1,30)
        End

        Say "The Show Me Five numbers I have selected are:",
            Number1 Number2 Number3 Number4 Number5
    End

Return
```

ListDSI - List Dataset Information

```
/* ListDSI - List Dataset information    REXX */
Arg Datasetname
RC = listdsi(datasetname)
If RC = 0 then do
    Say "Allocation was successful."
    Say "SYSADirBlk="SYSADirBlk
    Say "SYSALLOC="SYSALLOC
    Say "SYSBLKSIZE="SYSBLKSIZE
    Say "SYSCreate="SYSCreate
    Say "SYSDSorg="SYSDSOrg
    Say "SYSDSName="SYSDSName
    Say "SYSExtents="SYSExtents
    Say "SYSExDate="SySExDate
    Say "SYSKEYLEN="SYSKEYLEN
    Say "SYSLRECL="SYSLRECL
    Say "SYSMembers="SYSMembers
    Say "SYSPassword="SYSPassword
    Say "SYSPrimary="SYSPrimary
    Say "SYSRefDate="SYSRefDate
    Say "SYSRACFA="SYSRACFA
    Say "SYSRECFM="SYSRECFM
    Say "SYSSeconds="SYSSeconds
    Say "SYSTrksCyl="SYSTrksCyl
    Say "SYSUnit="SYSUnit
    Say "SYSUnits="SYSUnits
    Say "SYSUpdated="SYSUpdated
    Say "SYSUSED="SYSUSED
    Say "SYSVolume="SYSVolume
End
Else do
    Say "Return code = " RC
    Say "SYSReason="SYSReason
    Say "SYSMSGLVL1="SYSMsgLvl1
    Say "SYSMSGLVL2="SYSMsgLvl2
End
```

LPDSIX - List a PDS Index to a Sequential File

This command will list the members of a PDS out to a sequential dataset for subsequent editing.

```
/* LPDSIX - List a PDS Index to a Sequential File  */
Arg PDSName
Call Proc01            /* Program Initialization   */
Call Proc02            /* List Members to an array */
Call Proc03            /* Create the sequential file array    */
Call Proc99            /* Finalization             */
Exit

/*--------------------------------------------------------------------*/
/*   Called Procedures                                                */
/*--------------------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say "LPDSIX - List a PDS Index to Sequential File"
  Say "Proceeding..."

  If PDSName = '' then do
      Say "PDSName not specified"
      Exit(16)
  End

  Prefix = sysvar(SYSUID)

  /* Say "The datasetname is " PDSName */
Return

/*------------*/
/* List Members to an array */
/*------------*/
Proc02:
  TmStart = Time(S)
  Say "Listing "PDSName" Members..."
  Dummy = OutTrap("Members.","*")
  "LISTD "PDSName" M "
  Dummy = OutTrap("OFF")
  NumMembers = Members.0
  If NumMembers < 2 then do
      Say "No members found: problem?"
      Exit(16)
  End
  AdjMembers = NumMembers - 6   /* Don't count the first six blanks */
  Say AdjMembers PDSName "names were found"
  TmEnd = Time(S)
  TmDur = TmEnd - TmStart
  Say "That took " TmDur "seconds!"
Return

/*------------*/
/* Create the sequential file array */
```

```
/*------------*/
Proc03:
  Do I = 1 to NumMembers
      Members.I = strip(Members.I)
      OrigMemname = left(Members.I,8)
  End

  OPDSN = "LPDSIX.Work"
  Dummy = OutTrap("Junk.","*")
  /* Allocate the sequential output file */
  Address TSO
  "Delete " OPDSN
  "Free FI(SeqFil)"
  Dummy = OutTrap("OFF")
  "ALLOC F(SeqFil) DA("OPDSN") NEW UNIT(SYSDA) DSORG(PS)",
         "SPACE(45 45) Tracks LRECL(88) BLKSIZE(6160) RECFM(F,B)"

  'EXECIO' NumMembers 'DISKW SeqFil ( STEM Members. FINIS'
  "Free FI(SeqFil)"
Return

/*------------*/
/* Finalization */
/*------------*/
Proc99:
    Say OPDSN "created. LPDSIX complete :)"
Return
```

Prime – Calculate Prime Numbers

This is an exec that will calculate prime numbers. If you want more or less, you can easily change the upper and lower limits.

```
/* Prime - Calculate Prime Numbers                              REXX  */
/* Written by David Grund, January, 2005                              */
Lowest  = 000005           /* This MUST be an odd number! */
Highest = 030000
Nth     = 200              /* Displaying every nth one    */

STime = time(E)
Quantity = 0
Say "Calculating prime numbers from "Lowest" to "Highest
Say " and displaying every "Nth"th one."
Do I = Lowest to Highest by 2
  Do J = 3 to I - 1 by 2
    /* Say "A: I="I"; J="J"; I mod J=" I // J */
    Quotient = I // J
    If Quotient = 0 then Leave
  End
  If Quotient <> 0 then do
    Quantity = Quantity + 1
    If Quantity // Nth = 0 then Say I "is a prime number; #"Quantity
  End
End
Say "I found "Quantity" prime numbers altogether."
ETime = time(E)
Say "This took "trunc((ETime - STime),2)" seconds."
```

PROCSYMS - Perform Symbolic Substitution

```
/* ProcSyms - ISPF Edit Macro                                  REXX EXEC */

/* This macro is used to perform symbolic substitution on a set of   */
/* JCL that calls a proc.                                            */

/* 1) Put all symbolics from the PROC statement into an array        */
/* 2) For testing, list the array                                    */
/* 3) Copy the array to a change command array                       */
/* 4) Execute the change command array                               */
Address "ISREDIT" "MACRO PROCESS"
Address "ISREDIT"
Call Proc01                          /* Put Symbs and Vals => arrays*/
Call Proc02                          /* List the arrays             */
Call Proc03                          /* Create the change arrays    */
/* l Proc04 */                       /* List the change arrays      */
Call Proc05                          /* Execute the changes         */
Exit

/*-----------*/
/* Proc01 - Put all the symbolics and values from the PROC statement */
/*          into arrays.                                             */
/*-----------*/
Proc01:
  Address "ISREDIT"
  "Exclude All '//*' 1"
  "Find ' PROC ' All NX"
  "ISREdit (NumFnd,Junk) = Find_Counts"

  If NumFnd = 0 then do
    zedsmsg = "'Not a PROC"
    zedlmsg = "I did not find a PROC statement in this member"
    Address ISPExec
    "SETMSG MSG(ISRZ000)"
    Exit
  End

  If NumFnd > 1 then do
    zedsmsg = "Too many"
    zedlmsg = "I found "NumFnd" PROC statements.",
              "I don't know how to process more than one."
    Address ISPExec
    "SETMSG MSG(ISRZ000)"
    Exit
  End

  /* At this point, we are looking at a line with the word 'PROC'    */
  ProcLine = 'Y'                      /* This is the PROC line      */
  "(CurrLine) = LINE .ZCSR"   /* Read the line that the cursor is on */
  CurrLine = left(CurrLIne,72)        /* Drop off the sequence number*/
  'ISREDIT (CLineNo,x) = CURSOR'           /* save cursor position   */
  Say "The input line is "CurrLine

  SymArray.0 = 0 ; ValArray.0 = 0     /* Init Sym and Value arrays   */
  NextEnt = 0                         /* Next array entry number     */
```

```
  StillIn = 'Y'                       /* Set continue processing sw  */

  Do while StillIn = 'Y'
    /* Parse the line into operands */
    Parse var CurrLine Operand1 Operand2 Operand3 Operand4
    Say "   Operand 1="Operand1
    Say "   Operand 2="Operand2
    Say "   Operand 3="Operand3
    Say "   Operand 4="Operand4

    If ProcLine = 'Y' then do         /* If this is the 'PROC' line, */
      Params = Operand3               /* Params are operand 3        */
      ProcLine = 'N'
    End
    Else
      Params = Operand2               /* Params are operand 2        */
    Params = strip(Params)

    If right(Params,1) = ',' then do  /* end in comma?               */
      LastLine = 'N'                  /* Off ind: this is not last   */
      Params = left(Params,length(Params)-1) /* Remove the comma     */
    End
    Else
      LastLine = 'Y'                   /* Set indicator              */

    Do while length(Params) > 0
        Call Proc011                   /* Get the next Parameter     */
    End
    If LastLine = 'Y' then             /* If this is the last line,  */
      StillIn = 'N'                    /* we are done                */
    Else do                            /* otherwise                  */
      CLineNo = CLineNo + 1            /* Bump line number           */
      "(CurrLine) = LINE "ClineNo      /* Read the next line down    */
      CurrLine = left(CurrLIne,72)     /* Drop off the seq number    */
    End
  End
Return

/*-----------*/
/* Get the Next Symbolic Parameter and Value */
/*-----------*/
Proc011:
  /* First handle the Symbolic */
  Pos = Index(Params,'=')              /* Point to the equals sign   */
  If Pos = 0 then do                   /* No more params on this line*/
    Params = ""                        /* Reduce the line to nothing */
    Return
  End
  ThisSym = left(Params,Pos-1)
  /* Say "Trace: ThisSym="ThisSym */
  NextEnt = NextEnt + 1
  SymArray.NextEnt = ThisSym
  SymArray.0 = NextEnt
  Params = DelStr(Params,1,length(ThisSym)+1)
  /* Say "The remainder of the line is" Params */

  /* Now handle the value      */
```

```
  Params = Params||" "                  /* Add a space, just in case */
  If left(Params,1) = "'" then do       /* Delimiter is an apostrophe*/
    Params = Delstr(Params,1,1)         /* Delete the first one      */
    EndPos = Index(Params,"'")
    If EndPos = 0 then do
      Say "Problem! No second apostrophe found; line=" Params
      Exit
    End
    Params = Delstr(Params,EndPos,1) /* Delete the second one        */
  End
  Else do
    EndPos = Index(Params,",")
    If EndPos = 0 then EndPos = Index(Params," ")
    If EndPos = 0 then do
      Say "Problem! Data line is corrupted; line="Params
      Say "         Length of Params="length(Params)
      Exit
    End
  End
  ThisVal = substr(Params,1,EndPos-1)
  /* Say "ThisVal="ThisVal */
  Params = DelStr(Params,1,length(ThisVal)+1)
  ValArray.NextEnt = ThisVal
  ValArray.0 = NextEnt
  Params = strip(Params)
Return

/*-----------*/
/* Proc02 - List the arrays */
/*-----------*/
Proc02:
  Say "Symbolic|Value"
  Do I = 1 to SymArray.0
      ThisStr = left(SymArray.I||"        ",8)
      ThisStr = ThisStr||" "
      ThisStr = ThisStr||ValArray.I
      Say ThisStr
  End
Return

/*-----------*/
/* Proc03 - Create the change arrays */
/*-----------*/
Proc03:
  ChgArray1.0 = 0 ; ChgArray2.0 = 0;
  Do I = 1 to SymArray.0
    /* Symbolics with the '.' */
    ChgArray1.I = "Change '&&&&"SymArray.I".' '"ValArray.I"' all"
    /* Symbolics without the '.' */
    ChgArray2.I = "Change '&&&&"SymArray.I"' '"ValArray.I"' all word"
  End
  ChgArray1.0 = SymArray.0 ; ChgArray2.0 = SymArray.0
Return

/*-----------*/
/* Proc04 - List the Change Arrays */
/*-----------*/
```

```
Proc04:
  Do I = 1 to ChgArray1.0
      Say ChgArray1.I
  End
  Do I = 1 to ChgArray2.0
      Say ChgArray2.I
  End
Return

/*-----------*/
/* Proc05 - Execute the Change Arrays */
/*-----------*/
Proc05:
  Address "ISREDIT"
  Do I = 1 to ChgArray1.0
      ChgArray1.I
      ChgArray2.I
  End
  address "ISREDIT" "LINE_AFTER 0 = NoteLine",
    "'-------------------------------------------------------------'"
  address "ISREDIT" "LINE_AFTER 0 = NoteLine",
    "'" Symbolic substitution performed ISPF macro ProcSyms."'"
  address "ISREDIT" "LINE_AFTER 0 = NoteLine",
    "'-------------------------------------------------------------'"
  "Up Max"
Return
```

PTS - PDS-to-Sequential; member name is prefix

This exec will "flatten out" a PDS, adding the member name to the front of each line. The result is written to a dataset for subsequent modification.

```
/* PTS - Copy a PDS to a sequential file, adding the     */
/*         member name to the first 8 positions          */
Arg PDSName
Call Proc01            /* Program Initialization         */
Call Proc02            /* List Members to an array       */
Call Proc03            /* Create the sequential file array    */
Call Proc04            /* Write the array to a dataset   */
Call Proc99            /* Finalization                   */
Exit

/*--------------------------------------------------------------------*/
/*   Called Procedures                                                */
/*--------------------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say "PTS - Copy PDS to Sequential"
  Say "Proceeding..."

  If PDSName = '' then do
      Say "PDSName not specified"
      Exit(16)
  End

  Prefix = sysvar(SYSPREF)
  If Prefix = "" then
      Prefix = sysvar(SYSUID)

  /* Follow TSO conventions. If the PDSName has quotes remove them.
     If not, add the userid to the front                          */
  If Left(PDSName,1) = "'" then do
      OurLen = length(PDSName) - 2
      PDSName = substr(PDSName,2,OurLen)
  End
  Else
      PDSName = Prefix||"."||PDSName
  /* Say "The datasetname is " PDSName */
Return

/*------------*/
/* List Members to an array */
/*------------*/
Proc02:
  Say "Listing "PDSName" Members..."
  Dummy = OutTrap("Members.","*")
  "LISTD '"PDSName"' M "
  Dummy = OutTrap("OFF")
  NumMembers = Members.0
  If NumMembers < 2 then do
```

```
      Say "No members found: problem?"
      Exit(16)
  End
  AdjMembers = NumMembers - 6   /* Don't count the first six blanks */
  Say AdjMembers PDSName "names were found"
Return

/*------------*/
/* Create the sequential file array */
/*------------*/
Proc03:
  SeqFileNumLines = 0
  Do I = 7 to NumMembers
      Members.I = strip(Members.I)
      OrigMemname = left(Members.I,8)
      Memname = strip(OrigMemName)
      InputDSN = "'"PDSName"("Memname")'"
      /* Say "InputDSN=" InputDSN */
      Address TSO
      "ALLOC DA("InputDSN") F(INDD) SHR REUSE"
      'EXECIO * DISKR INDD ( STEM REC. FINIS'
      'FREE F(INDD)'
      ThisMemNumLines = REC.0
      /*  Say "Member contains" ThisMemNumLines" lines" */

      Do J = 1 to ThisMemNumLines
          ThisLine = OrigMemName || Rec.J
          SeqFileNumLines = SeqFileNumLines + 1
          SeqArray.SeqFileNumLines = ThisLine
      End
  End
  /* Say 'The sequential file array contains' SeqFileNumLines' lines'*/
Return

/*------------*/
/* Write the array to a dataset */
/*------------*/
Proc04:
  OPDSN = "'"Prefix||"."||PTS.Work"'"
  Dummy = OutTrap("Junk.","*")
  /* Allocate the sequential output file */
  Address TSO
  "Delete " OPDSN
  "Free FI(SeqFil)"
  Dummy = OutTrap("OFF")
  "ALLOC F(SeqFil) DA("OPDSN") NEW UNIT(SYSDA) DSORG(PS)",
         "SPACE(45 45) Tracks LRECL(88) BLKSIZE(6160) RECFM(F,B)"

  /* Now write the array to the sequential output file */
  'EXECIO' SeqFileNumLines 'DISKW SeqFil ( STEM SeqArray. FINIS'
  "Free FI(SeqFil)"
Return

/*------------*/
/* Finalization */
/*------------*/
Proc99:
```

```
    Say OPDSN "created. PTS complete :)"
Return
```

PTS2 - PDS-to-Sequential; member name is inserted

This exec will "flatten out" a PDS, inserting a line with the member name between each member. The result is written to a dataset for subsequent modification.

```
/* PTS2 - Copy a PDS to a sequential file, adding the */
/*         member name between members                */
Arg PDSName
Call Proc01            /* Program Initialization      */
Call Proc02            /* List Members to an array    */
Call Proc03            /* Create the sequential file array    */
Call Proc04            /* Write the array to a dataset */
Call Proc99            /* Finalization                 */
Exit

/*--------------------------------------------------------------------*/
/*   Called Procedures                                                */
/*--------------------------------------------------------------------*/
/*------------*/
/* Program Initialization */
/*------------*/
Proc01:
  Say "PTS2 - Copy PDS to Sequential"
  Say "Proceeding..."

  If PDSName = '' then do
      Say "PDSName not specified"
      Exit(16)
  End

  Prefix = sysvar(SYSPREF)
  If Prefix = "" then
      Prefix = sysvar(SYSUID)

  /* Follow TSO conventions. If the PDSName has quotes remove them.
     If not, add the userid to the front                          */
  If Left(PDSName,1) = "'" then do
      OurLen = length(PDSName) - 2
      PDSName = substr(PDSName,2,OurLen)
  End
  Else
      PDSName = Prefix||"."||PDSName
  /* Say "The datasetname is " PDSName */
Return

/*------------*/
/* List Members to an array */
/*------------*/
Proc02:
  Say "Listing "PDSName" Members..."
  Dummy = OutTrap("Members.","*")
  "LISTD '"PDSName"' M "
  Dummy = OutTrap("OFF")
  NumMembers = Members.0
  If NumMembers < 2 then do
```

```
      Say "No members found: problem?"
      Exit(16)
  End
  AdjMembers = NumMembers - 6   /* Don't count the first six blanks */
  Say AdjMembers PDSName "names were found"
Return

/*------------*/
/* Create the sequential file array */
/*------------*/
Proc03:
  SeqFileNumLines = 0
  Do I = 7 to NumMembers
      Members.I = strip(Members.I)
      OrigMemname = left(Members.I,8)
      Memname = strip(OrigMemName)
      InputDSN = "'"PDSName"("Memname")'"
      /* Say "InputDSN=" InputDSN */
      Address TSO
      "ALLOC DA("InputDSN") F(INDD) SHR REUSE"
      'EXECIO * DISKR INDD ( STEM REC. FINIS'
      'FREE F(INDD)'
      ThisMemNumLines = REC.0
      /*  Say "Member contains" ThisMemNumLines" lines" */

      /* First write a record containing the member name   */
      SeqFileNumLines = SeqFileNumLines + 1
      SeqArray.SeqFileNumLines = "== " || OrigMemName || " =="

      Do J = 1 to ThisMemNumLines
          ThisLine = Rec.J
          SeqFileNumLines = SeqFileNumLines + 1
          SeqArray.SeqFileNumLines = ThisLine
      End
  End
  /* Say 'The sequential file array contains' SeqFileNumLines' lines'*/
Return

/*------------*/
/* Write the array to a dataset */
/*------------*/
Proc04:
  OPDSN = "'"Prefix||"."||PTS2.Work"'"
  Dummy = OutTrap("Junk.","*")
  /* Allocate the sequential output file */
  Address TSO
  "Delete " OPDSN
  "Free FI(SeqFil)"
  Dummy = OutTrap("OFF")
  "ALLOC F(SeqFil) DA("OPDSN") NEW UNIT(SYSDA) DSORG(PS)",
         "SPACE(45 45) Tracks LRECL(80) BLKSIZE(6160) RECFM(F,B)"

  /* Now write the array to the sequential output file */
  'EXECIO' SeqFileNumLines 'DISKW SeqFil ( STEM SeqArray. FINIS'
  "Free FI(SeqFil)"
Return
```

```
/*-------------*/
/* Finalization */
/*-------------*/
Proc99:
    Say OPDSN "created. PTS2 complete :)"
Return
```

RexxModl - Rexx Exec Model

Every toolbox should have a model from which to create a new program, be it bare-bones, or chock-full of routines to weed through. Here is the former.

```
/* PgmID - Program Function         - Rexx Exec */
/* Written by . . .                             */
/* This program will...                         */

Arg Spec

Call Init                                  /* Init Program      */

Exit

/*-------------*/
/* Program Initialization */
/*-------------*/
Init:

Return
```

Scale - Display a Scale

This is a code snippet that is handy for lining things up, when necessary.

```
Say '         1         2         3         4         5         6'
Say '....5....0....5....0....5....0....5....0....5....0....5....0'
```

ScanLibs – Scan Library Concatenations

```
/* ScanLibs - Scan a Concat of Libraries for text              REXX */
/* Written by David Grund, Feb 21, 2005.                          */
Arg OurDD OurText
Call Proc01             /* Initialization         */
Call Proc02             /* LisaA to an array      */
Call Proc03             /* Adjust the array       */
Call Proc04             /* Remove 'KEEP' lines    */
/* Call Proc05 */       /* Write the array to a dataset and view it */
Call Proc06             /* Isolate the DD         */
Call Proc07             /* List all DSN's in the concatenation */
Call Proc08             /* Scan each PDS          */

/*------------*/
/* Proc01 - Initialization */
/*------------*/
Proc01:
  If OurDD = "" | OurText = "" then do
    Say "Command syntax: ScanLibs DDName OurText"
    Exit(16)
  End
Return

/*------------*/
/* Proc02 - ListA to an Array */
/*------------*/
Proc02:
  Dummy = OutTrap("output_line.","*")
  "LISTA SY ST"
  NumLines = Output_Line.0
  /* Say Numlines "lines were created */
  Dummy = OutTrap("OFF")
Return

/*------------*/
/* Proc03 - Adjust the Array */
/*------------*/
Proc03:
  /* Move the line with the DDName above the first datasetname that */
  /* it is concatenated to. It is currently below.                  */
  Do I = 1 to NumLines
    Col1_2 = Substr(Output_Line.I,1,2)
    Col3   = Substr(Output_Line.I,3,1)
    Col12_15 = Substr(Output_Line.I,12,4)
    If Col1_2 = '  ' & Col 3 /= "  " & Col12_15 = 'KEEP' then do
      J = I - 1
      SaveLine      =    Output_Line.I
      Output_Line.I = Output_Line.J
      Output_Line.J = SaveLine
    End
  End
Return

/*------------*/
/* Proc04 - Remove all lines that say only 'KEEP'  */
/*------------*/
```

```
Proc04:
  J = 0           /* Output array counter */
  Do I = 1 to NumLines
    ThisLine = strip(Output_Line.I)
    If (Left(ThisLine,4) = 'KEEP') | ,
       (Left(ThisLine,8) = 'TERMFILE') then nop=nop
    Else do
      J = J + 1
      NewArray.J = Output_Line.I
    End
  End
  NewArray.0 = J
Return

/*------------*/
/* Proc05 - Write the array to a dataset and view it */
/*------------*/
Proc05:
  "Delete ScanLibs.List Purge"
  "Allocate DD(FMList) DA(ScanLibs.List) new space(1 1) tracks",
            "Lrecl(80) Block(5600)  recfm(f b) retpd(1)"
  "ExecIO" NewArray.0 "DiskW FMList (STem NewArray. FINIS"
  "Free DDName(FMList) DA(ScanLibs.List)"
  Address "ISPEXEC" "View Dataset (Scanlibs.List)"
  exit(0)
Return

/*------------*/
/* Proc06 - Isolate the DD */
/*------------*/
Proc06:
  J = 0    /* DSNArray counter */
  DDName = ''
  Do I = 1 to NewArray.0
    If left(NewArray.I,2) = '  ' then
      DDName = left(strip(NewArray.I),8)
    else do
      ThisRec = DDName||strip(NewArray.I)
      J = J + 1
      DSNArray.J = ThisRec
    End
  End
  DSNArray.0 = J
  /* Do I = 1 to DSNArray.0; Say DSNArray.I; End  */
Return

/*------------*/
/* Proc07 - List all DSN's in the concatenation */
/*------------*/
Proc07:
  DDFound = 0
  Do I = 1 to DSNArray.0       /* Skip the first six lines */
    If left(DSNArray.I,8) = OurDD then do
      DDFound = DDFound + 1
      DSN = strip(substr(DSNArray.i,9,63))
      /* Say "I found" DSN */
      DSNArray2.DDFound = DSN
```

```
    End
  End
  DSNArray2.0 = DDFound
  Say "I found" DDFound "datasets concatenated to "OurDD
Return

/*------------*/
/* Proc08 - Iteratively scan each PDS */
/*------------*/
Proc08:
MemSearched = 0
TextFound = 0
Do I = 1 to DSNArray2.0
  DSN = DSNArray2.I
     Say "Processing DSN="DSN
  /* First let's make sure this dataset is a PDS */
  RC = ListDSI("'"DSN"'" Directory)
  If RC > 0 then do
    Say "Error processing "DSN
    Say SYSMSGLVL1
    Say SYSMSGLVL2; Say
    Return
  End
  If SYSDSORG = "PO" then do
    Dummy = OutTrap("PDSMems.","*") /* List the member names */
    "ListD '"DSN"' M"
    NumLines = PDSMems.0
    Dummy = OutTrap("OFF")
    Do J = 7 to PDSMems.0
      /* Say "  Proc08: This PDS member is: "PDSMems.J   */
      Call Proc081  /* Scan THIS PDS */
    End
  End
  Else
    Say "Dataset "DSN" is not a PDS."
End
Say "Members searched: "MemSearched
Say "LInes found containing the text: "TextFound
Return

/*------------*/
/* Proc081 - Scan THIS PDS Member */
/*------------*/
Proc081:
  MemSearched = MemSearched + 1
  Member = strip(PDSMems.J)
  If Pos('-',Member) = 0 then do /* Member Names with a '-' in them? */
    OurDS  = DSN"("Member")"
    If Pos('ALIAS',OurDS) = 0 then do       /* Bypass aliases */
      /* Say "    Proc081: Processing "OurDS   */
      "Allocate DD(ThisMem) DA('"OurDS"') shr"
      If RC > 0 then exit
      "ExecIO * DiskR ThisMem (Stem DS1Lines. Finis "
      "Free DDName(ThisMem)"

      Do K = 1 to DS1Lines.0
        If Pos(OurText,DS1Lines.K) > 0 then do
```

```
          Say "I found "OurText" in "OurDS
          Say DS1Lines.K
          TextFound = TextFound + 1
        End
      End
    End
  End
Return
```

SDN - Sorted Directory w/Notes (Directory Annotator)

This is a handy ISPF macro that I wrote to keep track of what I have in my PDS's. This command will create and maintain a member called "@LIST", which contains a one-liner about each member in the PDS. Hopefully, this member will always be the first in a PDS.

Unfortunately, this command can be invoked only while you are editing a member of the PDS that you wish to annotate.

```
/* SDN - REXX EXEC                          */
/* Sorted Directory w/Notes - Edit Macro  */
/* Written by David Grund                  */
/* Changed 7/27/95- restore the TSO Profile prefix before ISPF       */
/* edit is invoked, instead of after the command is complete         */

ADDRESS "ISREDIT" "MACRO PROCESS"

/*-------------------------------------------------------------------*/
/*  Initialization                                                   */
/*-------------------------------------------------------------------*/
/* It's almost impossible to effectively handle datasetnames while   */
/* the TSO Profile Prefix is set to on.                              */
PREFIX = SYSVAR(SYSPREF)              /* Get the Prefix              */
If PREFIX = "" then DO                /* prefix is not set           */
    PrefixOn = 0                      /* Set a switch for later      */
end
else Do
    PrefixOn = 1                      /* Set a switch for later      */
    ADDRESS TSO
    "Profile NoPrefix"                /* Turn the prefix off         */
end

/*-------------------------------------------------------------------*/
/*  1) Read @LIST from current pds                                   */
/*-------------------------------------------------------------------*/
Address "ISREDIT" "(XDSN)=DATASET"
Dummy = ListDsi(XDSN)
If SYSDSORG ¬= "PO" then do
    Say "This dataset is not a PDS. No action performed."
    Exit
end
IPDSN = "'"XDSN"(@LIST)'"
If SYSdsn(IPDSN) = "OK" then
    nop = nop
    /* Say "The dsn is "IPDSN  */
else do                            /* Create @List with one member */
    "NewStack"
    "Allocate DD(FileA) DA("IPDSN") shr"
    ARec = "@LIST      This member"
    Push ARec
    "ExecIO 1 DiskW FileA "
    "ExecIO 0 DiskW FileA (Finis"       /* Close the output file */
    "Free DDNAME(FileA)"
end
```

```
/* "Free     FI(OldFile)"  */
"Allocate FI(OldFile) DA("IPDSN") shr"

"ExecIO * DiskR OldFile (STEM FileARec. FINIS"
"Free     FI(OldFile)"
/* Say FileARec.0 "Records read into the FileARec array"  */

/*--------------------------------------------------------------------*/
/*  2) Get member list of current PDS                                 */
/*--------------------------------------------------------------------*/
Dummy = OutTrap("FileBRec.","*")
"LISTD " IPDSN " M"
Dummy = OutTrap("OFF")
/* Say FileBRec.0 "Records read into the FileBRec array"  */

/* ListD has a problem when run from within a REXX EXEC.          */
/* It spits out two or three lines that it doesn't write when     */
/* running from outside of an EXEC. These lines start with the    */
/* string "--MEMBER--". Find out where our list really starts,    */
/* and save that record number for use later.                     */
FileBPos = 0                    /* Initialize this value         */
Do I = 1 to 15
    If SubStr(FileBRec.I,1,11) = "--MEMBERS--" then do
        FileBPos = I + 1
        Signal Done2
    end
    /* Say I FileBRec.I */
end
Done2: Nop=nop
If FileBPos = 0 then do
    Say "Problem with SDN EXEC at POINT 1"
    Exit(0)
end

/*--------------------------------------------------------------------*/
/*  3) Compare, and create the new @List                              */
/*--------------------------------------------------------------------*/
OPDSN = "'"XDSN"(@LIST)'"
"NewStack"
"Allocate DD(FileC) DA("OPDSN") shr"
FileAPos = 1
/* FileBPos is set in section 2 above  */
FileCPos = 1

GetBoth:
/* Get a record from File A */
If FileAPos > FileARec.0 then
    FileAKey = '99999999'
Else Do
    FileAKey = SubStr(FileARec.FileAPos,1,8)
    ARec     = SubStr(FileARec.FileAPos,1,72)
    FileAPos = FileAPos + 1
end
/* Say "The first record from FileA is: " ARec  */

/* Get a record from File B */
If FileBPos > FileBRec.0 then
```

```
    FileBKey = '99999999'
Else Do
    FileBKey = SubStr(FileBRec.FileBPos,3,10)
    BRec     = SubStr(FileBRec.FileBPos,3,72)
    FileBPos = FileBPos + 1
END
/* Say "The first record from FileB is: " BRec  */

Compare:
If FileAKey < FileBKey then signal ALow
If FileBKey < FileAKey then signal BLow
/* Say "The record being compared is " FileAKey FileBKey  */

/* Member names are the same */
If FileAKey = "99999999" then       /* Both files are at end-of-file */
    signal EOF
CRec = SubStr(ARec,1,9)" "Substr(ARec,11,70)
/* Say "The record going out is " Crec     */
Push CRec
"ExecIO 1 DiskW FileC "
Signal GetBoth

ALow:
CRec = SubStr(ARec,1,9)"-"Substr(ARec,11,70)
Push CRec
"ExecIO 1 DiskW FileC "
/* Get a record from File A */
If FileAPos > FileARec.0 then
    FileAKey = '99999999'
Else Do
    FileAKey = SubStr(FileARec.FileAPos,1,8)
    ARec     = SubStr(FileARec.FileAPos,1,72)
    FileAPos = FileAPos + 1
end
Signal Compare

BLow:
CRec = SubStr(BRec,1,9)"+"Substr(BRec,11,70)
Push CRec
"ExecIO 1 DiskW FileC "
/* Get a record from File B */
If FileBPos > FileBRec.0 then
    FileBKey = '99999999'
Else Do
    FileBKey = SubStr(FileBRec.FileBPos,3,10)
    BRec     = SubStr(FileBRec.FileBPos,3,72)
    FileBPos = FileBPos + 1
END
Signal Compare

EOF:
"ExecIO 0 DiskW FileC (Finis"       /* Close the output file */
"Free DDNAME(FileC)"

/* If the TSO Profile Prefix was set to on when we came in, restore  */
/* it.                                                               */
If PrefixOn = 1 then do              /* We came in with the setting */
```

```
    ADDRESS TSO
    "Profile Prefix("PREFIX")"        /* Restore it                    */
end

ADDRESS "ISPEXEC" "EDIT Dataset("OPDSN") "
```

SHOWDUPS - Show Duplicates

This exec is an ISPF macro that will show all duplicated lines in a dataset.

```
/* ShowDups - Show Duplicate Lines - REXX Exec  */
ADDRESS ISREDIT
'MACRO (begcol endcol)'
If Begcol = '?' then do
    zedsmsg = 'ShowDup begcol,endcol'
    zedlmsg = 'Command syntax: ShowDup beginning col, ending
col'
    signal quitme
end
numcheck = DATATYPE(begcol,N)         /* Determine if any parms
have */
If NumCheck /= 1 then BegCol = 1      /* been passed.
*/
numcheck = DATATYPE(endcol,N)
If NumCheck /= 1 then 'ISREDIT (endcol) = LRECL'

'ISREDIT (currline) = LINENUM .ZFIRST'  /* save starting record
# */
'ISREDIT (lastline) = LINENUM .ZLAST'   /* save ending record #
*/
'ISREDIT (cl,cc)    = CURSOR'           /* save cursor position
*/
DupCnt = 0
'ISREDIT EXCLUDE ALL'
Do currline = 1 to lastline - 1
    'ISREDIT (line1) = LINE' currline
    line1 = substr(line1,begcol,(endcol - begcol) + 1)
    nextline = currline + 1
    'ISREDIT (line2) = LINE' nextline   /* get next record */
    line2 = substr(line2,begcol,(endcol - begcol) + 1)
    If line1 == line2 then do
      DupCnt = DupCnt + 1
      "ISREDIT LABEL " currline " = .A"
      "ISREDIT LABEL " nextline " = .B"
      "ISREDIT RESET EXCLUDED .A .B"
    end
end
zedsmsg = DupCnt 'DUPS FOUND'
zedlmsg = DupCnt 'duplicate lines were detected'
Quitme:
ADDRESS ISPEXEC
'SETMSG MSG(ISRZ000)'
EXIT 0
```

Stack - Start another ISPF session

This is a handy Rexx exec that, while you are in an ISPF session, will start another one. The action is totally recursive.

```
/* Stack - Start Another ISPF Session     - Rexx Exec */
/* This program will start another ISPF session so you don't
have to back out of everything you have when you want another
window.     */

Address ISPExec
"Select Panel(ISR@Prim)"
```

TimeFmts - Show all time formats

```
/* TimeFmts - Time Formats - Rexx EXEC     */
/* Written by David Grund                   */

  Say "Date( )" Date( )
  Say "Date(B)" Date(B)
  Say "Date(C)" Date(C)
  Say "Date(D)" Date(D)
  Say "Date(E)" Date(E)
  Say "Date(J)" Date(J)
  Say "Date(M)" Date(M)
  Say "Date(O)" Date(O)
  Say "Date(S)" Date(S)
  Say "Date(U)" Date(U)
  Say "Date(W)" Date(W)

  Say "Time( )" Time( )
  Say "Time(C)" Time(C)
  Say "Time(H)" Time(H)
  Say "Time(L)" Time(L)
  Say "Time(M)" Time(M)
  Say "Time(N)" Time(N)
  Say "Time(R)" Time(R)
  Say "Time(S)" Time(S)
```

TimeToGo - Display time until an event

This exec can be used to display how much time remains until a certin event. This can be pretty informative and useful on a Friday afternoon at about 2:00.

```
/* TimeToGo                - Rexx EXEC          */
/* Written by David Grund                        */
/* This is a Rexx learning exercise. Its purpose is to      */
/* calculate how much time remains to a specific event      */

TargetHH = 16                /* Set these to the  */
TargetMM = 00                /* event             */
TargetSS = 00                /* time              */
TargetSeconds = (TargetHH * 60 * 60) + (TargetMM * 60) +
TargetSS

TimeNow   = Time(N)
TimeNowHH = left(TimeNow,2)
TimeNowMM = substr(TimeNow,4,2)
TimeNowSS = right(TimeNow,2)
SecondsNow = (TimeNowHH * 60 * 60) + (TimeNowMM * 60) +
TimeNowSS

SecondsLeft = TargetSeconds - SecondsNow
/* Say "SecondsLeft = " SecondsLeft */

TimeToGoHH = trunc(SecondsLeft / 3600)
SecondsLeft = SecondsLeft - (TimeToGoHH * 3600)

TimeToGoMM = trunc(SecondsLeft / 60)
SecondsLeft = SecondsLeft - (TimeToGoMM * 60)
TimeToGoSS = SecondsLeft

/* Now format the time so we don't get something like 7:7:4 */
If TimeToGoSS < 10 then
    TimeToGoSS = '0' || TimeToGoSS
If TimeToGoMM < 10 then
    TimeToGoMM = '0' || TimeToGoMM

If TimeToGoHH > 0 then
    Say "Time to Go: "TimeToGoHH":"TimeToGoMM":"TimeToGoSS
Else
    Say "Time to Go: "TimeToGoMM":"TimeToGoSS
```

Section IV - The Rexx Environment

This section of the manual describes the following Rexx features:

1. Establishing your Rexx environment

2. Using Rexx with ISPF

3. Using Rexx in the background (batch jobs)

4. Debugging your Rexx program

5. Trapping Errors

6. Examples

Establishing Your Rexx Environment

Establishing your Rexx environment is simply a matter of allowing the system to quickly and easily find your commands, so you don't have to type in lengthy strings to execute your commands. Like so many things, there are several ways to do this.

Regardless of the method you choose, you need to pick an existing library, or create a new one. To create a new Rexx exec library (a library from which all of *your* execs will be called), either use ISPF 3.2, then option M, or you can do this from TSO by issuing the following commands:

```
Address TSO
"Free Fi(NEWDA)"
"delete REXX.EXEC"
"Alloc Fi(NEWDA) DA(REXX.EXEC) new space(15 1) dir(45) track" ,
      "DSNType(Library)" ,
      "dsorg(PO) recfm(V B) lrecl(255) blksize(0)"
"Free Fi(NEWDA)"
```

Now, you have to point the system to your Rexx exec library. There are five options that I can think of. I will discuss the simplest first, to the most involved.

1. Simply allocate DDName SYSEXEC to your library:
   ```
   "Free Fi(SYSEXEC)"
   "Alloc Fi(SYSEXEC) DA(REXX.EXEC) SHR"
   ```
   The problem with this is that this unallocates ALL other concatenated libraries that SYSEXEC was pointing to. For short-term or emergency purposes, this will work. But it could be that the successful processing in your system will depend on those libraries being available.
2. You could research to see what was currently allocated to SYSEXEC (LISTA SY ST). Then, after freeing DDName SYSEXEC, you would allocate your exec library to SYSEXEC, and then reallocate all of the system libraries that were previously allocated to it. The problem with this was that if the system administrators responsible for the concatenation of your procedure libraries changed the list of files allocated to SYSEXEC, you would not have that updated list available to you.
3. Some shops write their logon procedures so you could pass it the name of a library that you wanted to allocate in front of (or in back of) the list of system exec or clist libraries. There was a lot of room for error in this method. If that is not available to you, proceed to the next item.
4. Whenever you log on to TSO, allocate DDName SYSUEXEC to your Rexx Exec library:
   ```
   "Alloc Fi(SYSUEXEC) DA(REXX.EXEC) SHR".
   ```
   Then issue the ALTLIB command.
   ```
   "ALTLIB Activate User(exec)"
   ```
   There is on big problem with this method: you must issue the ACTIVATE portion wherever you will be working.
   If you issue the Activate within TSO, and before you start ISPF, then your ISPF session will not see that allocation, and your commands will not be available. If you enter ISPF, and then split screens, the commands are not available to you on that side until you issue the Activate on that side. I don't know if this is a design feature or a bug, but it is definitely problematic.

5. Concatenate your Rexx Exec library to the top of the current SYSEXEC list, regardless of what is currently allocated to it. This differs from #2 above in that you don't have to explicitly supply the names of the libraries. This option requires the use of a tool that will do this dynamically. I have created that tool, and it is called ConcatL. Check this manual for the source.

Using Rexx with ISPF

You can invoke the ISPF editor or browser from within a Rexx exec. Furthermore, you can run a Rexx exec upon beginning the edit of a dataset. This feature is called an ISPF edit macro.

ISPF Browser

To browse a dataset from within a Rexx exec:

```
ADDRESS "ISPEXEC" "BROWSE Dataset(dsn)"
```

where

dsn is the datasetname of the file you wish to browse

ISPF Editor

To edit a dataset from within a Rexx exec:

```
ADDRESS "ISPEXEC" "EDIT Dataset(dsn) Macro(macname)"
```

where

dsn is the datasetname of the file you wish to edit
macname is the name of the ISPF Rexx exec that will function as the ISPF macro.

ISPF Edit Macros

The purpose of an ISPF edit macro is to perform one or more ISPF edit commands on a dataset immediately after opening it for edit. If you need to do something to a dataset after it is opened for edit, an edit macro may be the way to accomplish this.

A complete dissertation of ISPF edit macros is beyond the scope of this book, but I provide enough to at least let you know how they are used in conjunction with Rexx.

An ISPF edit macro can be used to reformat or restructure data in a dataset prior to the dataset being presented to the user for editing.

The first line in an ISPF macro is one to tell the Rexx exec that it is to function as an ISPF macro:

```
Address "ISREDIT" "Macro Process".
```

Just about any ISPF editor primary command can be used in an ISPF macro. Simply precede the command with `Address "ISREDIT"`.

This is an example of an ISPF macro that is used to edit the output of the TSO command `LISTA SY ST` (see the "LA" exec in the examples):

```
/* REXX - LAE - Edit macro for LA          - Rexx Exec */
/* Written by David Grund, April 7, 1995               */

1 ADDRESS "ISREDIT" "MACRO PROCESS"
2 ADDRESS "ISREDIT" "EXCLUDE ALL --DDNAME 1"
```

```
3 ADDRESS "ISREDIT" "EXCLUDE ALL '           keep' 1 "
4 ADDRESS "ISREDIT" "Delete ALL X"
5 ADDRESS "ISREDIT" "C 'KEEP' '--------------' word all 12"
```

Line 1 tells the Exec that it is an ISPF macro.

Line 2 is an ISPF command that excludes all lines where "`--DDNAME`" appears in column 1.

Line 3 is an ISPF command that does the same thing with a different character string.

Line 4 tells ISPF to delete all excluded lines (those that were excluded by the previous two lines)

Line 5 tells ISPF to change the all occurrences of the string "`KEEP`" that start in column 12 to 14 dashes.

Using Rexx in the background (batch jobs)

As long as your Rexx exec is not interactive, you should have no problem running it in the background, that is, via a job you submit from your terminal.

A good candidate for a Rexx exec that should run in the background is one that will take a lot of CPU time, or produce a lot of output. By running it in the background, you can free up your terminal to do other things.

Instead of allocating files from within your Rexx exec, you would allocate them via the JCL. You could *keep* the allocations buried within your Rexx exec, but then you will be hiding the datasetname from your user. Unless this is what you *specifically* want to do, put the DD statement for that file in the JCL, and remove the allocate step from your Rexx exec.

An example of JCL for running a Rexx exec in the background is shown:

```
1  //STEP010 EXEC PGM=IKJEFT01
2  //SYSTSPRT DD  SYSOUT=*
3  //SYSTSIN  DD   *
4    EXEC 'GRUNDDAV.REXX.EXEC(TEST1)'
5  /*
```

Note that this JCL can be used for executing *any* TSO command, not just Rexx execs.
Line 1 executes program IKJEFT01, which is the background TSO command processor.
Line 2 allocates the TSO SYSOUT dataset.
Line 3 allocates the TSO SYSIN dataset
Line 4 executes the TSO command. In this case, it's an exec from my exec PDS.
(Line 5 is simply the JES end-of-data statement.)

Debugging your Rexx program

If your program operates in a manner that doesn't seem quite right, and the cause is not immediately evident, it is probably time to go into debugging mode. Debugging is the process of putting code into your program to make your program tell you where it is, what it is about to do, or what it has done.

Typically, you would not leave any "active" debugging code in your production program. Instead of deleting it, you could comment it out, but if there is too much, it could detract from the readability of the program.

There are several ways to debug a Rexx exec.

One way is to put "Say" statements in strategic locations. This will tell you what paths the program is taking. Along this same line is commenting out instructions that you suspect to be causing the problems.

Another way is to use the Rexx Trace facilities.

I have always used the first method, because it is simpler, easier to "unplug", and gave me the same end result. The second method can hammer you with output that can serve more to confuse you than to help you. And to top it off, I think the Rexx Trace facilities are a little complicated. But it still warrants a short discussion, so here it is.

To interrupt your Rexx program from running, press the ATTN, or PA1 key. The program will break out of its current processing, and if there is code left to execute, the following will be displayed:

```
ENTER HI TO END, A NULL LINE TO CONTINUE, OR AN IMMEDIATE COMMAND+ -
```

You have several options for a response:

1) Enter key- The program will continue running
2) **HI** (Halt Interpret)- The program will end.
3) **HT** (Halt Typing)- The program will stop displaying output.
4) **RT** (Resume Typing)- The program will resume displaying output
5) **TS** (Trace Start)- Rexx will enter Interactive Trace Mode
6) **TE** (Trace End)- Rexx will exit Interactive Trace Mode

Interactive Trace Mode

Interactive Trace Mode is where Rexx will display each of the lines as it executes them, prefixed by the line numbers. When it pauses for input, you can change the value of a variable, or hit Enter to continue processing.

Trapping Errors

Trapping Errors is the process of detecting certain program conditions, and then acting based on those conditions.

This facility may be used in debugging, but can also be used in a production program (but carefully).

Error-trapping instructions:
Signal On *condition*
Signal Off *condition*
Call On *condition* Name *subroutinename*

Signal On *condition*

This instruction will effect a transfer of control to a designated location in the program whenever a certain condition is detected by the program. After the condition is handled, the program terminates.

Signal Off *condition*

This instruction will cancel the effects of a Signal On for this particular condition only.

Call On condition Name subroutinename

This instruction will cause the program to perform a call to subroutine every time the program detects a certain condition. After the condition is handled, the subroutine returns control to the next sequential instruction iun the program. The subroutine cannot return any values.

Naming a subroutine is optional.

Condition

The condition cited in the above instructions can be one of the following:

1. Syntax- Rexx encountered a syntax error in an instruction.
2. Error - A TSO or ISPF command returned a non-zero return code
3. Failure- A command that was passed to the environment has failed
4. NoValue- A variable was never given a value. Typically, this is not an error, because Rexx, by default, treats an unassigned variable as a literal.
5. Halt- The PA1/Attn key was hit.

Examples

The following Rexx exec will be used in each of the examples. For each example, the "Main processing" section of the program will be different.

```
/* Rexx program to demonstrate error-trapping */
Signal On Syntax
Call   On Error  Name Error_Handler
Call   On Failure
Signal On NoValue
Signal On Halt

(Main processing section)

Exit

Syntax:
  Say "I am in the Syntax condition-handling routine now."
  Say "I am going to terminate the program because of this"
Exit

Error_Handler:
  Say "I am in the Error  condition-handling routine now."
  Say "I am going to continue processing"
Return

Failure:
  Say "I am in the Failure condition-handling routine now."
  Say "I am going to continue processing"
Return

NoValue:
  Say "I am in the NoValue condition-handling routine now."
   Say "I am going to terminate the program because of this"
 Exit

 Halt:
   Say "I am in the Halt   condition-handling routine now."
   Say "I think you hit the attention key!"
   Say "I am going to terminate the program because of this"
 Exit
```

The following illustrates the output from running the above Rexx exec, causing different conditions to occur. We do this by replacing the "main processing section" above with each of the examples.

Example 1

Main processing section:

```
Say "1) This statement is perfect, and will generate no errors."
Say "2) The next statement will generate a Syntax condition"
PI = 3.1416
Circumference = PI *
```

Displays:

```
1) This statement is perfect, and will generate no errors.
2) The next statement will generate a Syntax condition
I am in the Syntax condition-handling routine now.
I am going to terminate the program because of this
```

Example 2

Main processing section:

```
Say "3) The next statement will generate a Error condition"
"Delete junk.data.set"
```

Displays:

```
3) The next statement will generate a Error condition
ERROR QUALIFYING XCON620.JUNK.DATA.SET
** DEFAULT SERVICE ROUTINE ERROR CODE 20, LOCATE ERROR CODE 8
LASTCC=8
I am in the Error  condition-handling routine now.
I am going to continue processing
```

(The dataset did not exist)

Example 3

Main processing section:

```
Say "4) The next statement will generate a Failure condition"
"This is not a good command"
```

Displays:

```
4) The next statement will generate a Failure condition
COMMAND THIS NOT FOUND
    10 *-* "This is not a good command"
       +++ RC(-3) +++
I am in the Failure condition-handling routine now.
I am going to continue processing
```

Example 4

Main processing section:

```
Say "5) The next statement will generate a NoValue condition"
Say "My age is " MyAge
```

Displays:

```
5) The next statement will generate a NoValue condition
I am in the NoValue condition-handling routine now.
I am going to terminate the program because of this
```

Appendix

Rexx instructions

Address	If	Options	Return
Arg	Interpret	Parse	Say
Call	Iterate	Procedure	Select
Do	Leave	Pull	Signal
Drop	Nop	Push	Trace
Exit	Numeric	Queue	Upper

Rexx functions

Abbrev	C2X	Fuzz	Reverse	Value
Abs	Datatype	Index	Right	Verify
Address	Date	Insert	Sign	Word
Arg	DBCS	Justify	Sourceline	WordIndex
Bitand	Delstr	LastPos	Space	WordLength
Bitor	Delword	Left	Strip	WordPos
Bitxor	Digits	Length	Substr	Words
B2X	D2C	Linesize	Subword	XRange
Center	D2X	Max	Symbol	X2C
Centre	ErrorText	Min	Time	X2D
Compare	Externals	Overlay	Trace	
Condition	Find	Pos	Translate	
Copies	Form	Queued	Trunc	
C2D	Format	Random	Userid	

TSO External functions

ListDSI	Storage
Msg	SYSDSN
OutTrap	SysVar
Prompt	

TSO Commands

DelStack	HI	QBuf	SubCom
DropBuf	HT	QElem	TE
ExecIO	MakeBuf	QStack	TS
ExecUtil	NewStack	RT	

Other Rexx References

The MVS QuickRef documentation (on TSO) also contains extensive technical documentation on Rexx (available only in some shops). This feature is commonly available via the “QW” command.

Book Manager is available in many shops:
Bookshelf: IKJ2BI01 - TSO/E V2R4 REXX/MVS Reference
Book name: IKJ2A303 TSO/E V2R4 REXX/MVS Reference
Book name: IKJ2C305 TSO/E V2R4 REXX/MVS User's Guide

The End