REstructured eXtended eXecutor (REXX)

Features of REXX

- Extremely Versatile
- Common Programming Structure
- Readability, Ease of Use – most instructions are meaningful English words
- Free Format –
  - Unlike other mainframe languages there is no restrictions to start an instruction in a particular column
  - You can skip spaces in a line or can skip entire lines
  - Multiple instructions in a single line or single instruction in multiple lines
  - No need to pre define variables
  - Instructions can be upper case, lower case or mixed case

Features of REXX

- Can execute commands from different host environments – Main feature of REXX on MVS which makes it a preferred language for developing productivity improvement tools.
- Powerful & Convenient built-in Functions
- Debugging capabilities – Using TRACE instruction. No need of external debugging tools
- Interpreted language – REXX interpreter is a built in component of TSO installation on any mainframe. Compilers are also available but not widely used
- Extensive Parsing Capabilities for character manipulation

History of REXX?

- Developed almost single handedly by **Michael Cowlishaw** of IBM.
- Standardized by American National Standard Institute
- IBM provides Rexx on following platforms:
  - REXX for VM (1983)
  - REXX for TSO/E (1988)
  - REXX Compiler (1989)
  - REXX for AS/400 (1990)
  - REXX for OS/2 (1990)
  - REXX for AIX/6000 (1993)
  - REXX for CICS/ESA (1994)
  - REXX for PC DOS (1995)
  - Object REXX for Windows (1997)

Before we begin

- Rexx program also called as ‘Exec’ can be written as a sequential dataset or partitioned dataset (PDS). But PDS is preferred.

- Some rules:
  - First instruction is a comment line which should contain string ‘REXX’. This is used by interpreter to identify the REXX program
  - Make generous use of blank lines and spaces
  - Do not give too many comments
  - Use mixed case for all variables and instructions
  - All structured programming practices are applicable to REXX also

Let us begin

- A simple REXX program

```
------------------
/* REXX */
Say Hello World
--------------------
```

- The output

```
HELLO WORLD
***
```

- For mixed case output include the string in quotes.

```
------------------
/* REXX */
Say 'Hello World'
--------------------
```

How free is the Free Format?

Try this

```
/************************* REXX ******************************/
      SAY 'This is a REXX literal string.'

      SAY                'This is a REXX literal string.'

        SAY 'This is a REXX literal string.'

      SAY,

      'This',

      'is',

      'a',

      'REXX',

      'literal',

      'string.'
```

Types of REXX instructions

- Keyword instruction – Such as Say, IF, ELSE
- Assignment – Changes the value of a variable. E.g. number = 4
- Label – A symbolic name followed by a colon used as identifier for subroutine
- Null – A comment or blank line. Ignored by interpreter
- Command – Processed by Host Environment

Running REXX EXEC

- Running EXEC explicitly
  - From any ISPF command line
    TSO EXEC ‘USERID.REXX.EXEC(TIMEGAME)’
  - From TSO shell
    EXEC ‘USERID.REXX.EXEC(TIMEGAME)’
- Running EXEC implicitly
  - allocate the PDS that contains it to a system file (SYSPROC or SYSEXEC)
  - The exec can be run as any TSO command

Passing information to an EXEC

- Through terminal interaction
  - Using PULL instruction

- By specifying input when invoking the exec
  - Using ARG instruction
  - TSO EXEC ‘userid.rexx.exec(add)’ '42 21'

- If too few values are specified then remaining variables are set to spaces
- If number of values passed are more than the number of variables the last variable will hold remaining values. To avoid this use ‘dummy variable’, a period (.)
- Use PARSE command to prevent the translation to upper case

Variables in REXX

- Variable names can consist of:

  A...Z uppercase alphabetic

  a...z lowercase alphabetic

  0...9 numbers

  @ # $ [? ! . _ special characters

- Restrictions on the variable name are:

  The first character cannot be 0 through 9 or a period (.)

  The variable name cannot exceed 250 bytes

  The variable name should not be RC, SIGL, or RESULT, which are REXX special variables

Examples of acceptable variable names are:

ANSWER ?98B X Word3 number the_ultimate_value

Arithmetic Operators

Operator	Meaning
+	Add
-	Subtract
*	Multiply
/	Divide
%	Divide and return a whole number without a remainder
//	Divide and return the remainder only
**	Raise a number to a whole number power
-number	Negate the number
+number	Add the number to 0

Arithmetic Operators (cont…)

Order of Evaluation:

- Expressions within parentheses are evaluated first.
- Expressions with operators of higher priority are evaluated before expressions with operators of lower priority.

--- Arithmetic Operator Priority ----------

- +	Prefix operators
**	Power (exponential)
* / % //	Multiplication and division
+ -	Addition and subtraction

Comparison Operators

Operator	Meaning
==	Strictly Equal
=	Equal
\ ==	Not strictly equal
\ =	Not equal
>	Greater than
<	Less than
> <	Greater than or less than (same as not equal)
> =	Greater than or equal to
\ <	Not less than
< =	Less than or equal to
\ >	Not greater than

Logical Operators

Operator	Meaning
&	AND
|	Inclusive OR
&&	Exclusive OR
Prefix \	NOT

Concatenation Operators

To concatenate two strings

Operator	**Meaning**
blank	Inserts a blank between two strings (even if there are multiple blanks)
||	Concatenate without any blanks

E.g.

```
Say true blue        /* results in TRUE BLUE */
Say true||blue       /* results in TRUEBLUE */
Say 'true'||'blue'   /* results in trueblue */
Say ('true')('blue')  /* results in trueblue */
A = (8/2)(3*3)      /* What is the value of A? */
```

Priority of REXX Operators

--- Overall Operator Priority ----------

\ or ^ - +	Prefix operators
**	Power (exponential)
* / % //	Multiply and divide
+ -	Add and subtract
blank abuttal	Concatenation operators
== = >< etc.	Comparison operators
&	Logical AND
&&	Inclusive OR and exclusive OR

TRACE Instruction - debugging

- TRACE R (Trace Results)
  Traces only final results
- TRACE I (Trace Intermediates)
  >V> - Variable value - The data traced is the contents of a variable
  >L> - Literal value - The data traced is a literal (string, uninitialized variable, or constant)
  >O> - Operation result - The data traced is the result of an operation on two terms
- Other Trace options:
  - **TRACE A (Trace ALL), TRACE C (Trace Commands), TRACE E (Trace Error), TRACE F (Trace Failure), TRACE L (Trace Labels), TRACE N (Trace Normal – default setting), TRACE O (Trace Off), TRACE S (Trace Scan)**

Controlling the flow

- Conditional instructions
  - IF expression/THEN/ELSE
  - SELECT/WHEN expression/OTHERWISE/END
- Looping instructions
  - DO expression/END
  - DO FOREVER/END
  - DO WHILE expression=true/END
  - DO UNTIL expression=true/END
- Interrupt instructions
  - EXIT
  - SIGNAL label
  - CALL label/RETURN

Conditional Instructions (IF/THEN/ELSE)

- Syntax:

  IF expression THEN instruction; ELSE instruction

  IF expression THEN NOP; ELSE instruction (NOP means ‘no operation’)

- If there are multiple instructions

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

If the DO END is not specified then only one line will be considered as part of ELSE expression.

Conditional Instructions
(SELECT/WHEN/OTHERWSIE/END)

- Syntax:

```
SELECT
  WHEN  expression  THEN  instruction
  WHEN  expression  THEN  instruction
  WHEN  expression  THEN  instruction
 .
 .
 .
  OTHERWISE
    instruction(s)
END
```

- DO END must be used here also if there are multiple instructions

Looping Instructions (Repetitive loops)

- Syntax:
  - Using a constant in DO/END loop

```
DO 5
  SAY 'Hello!'
END                /* Prints ‘Hello!’ five times on the screen */
```

  - Using a control variable in DO/END loop (by default incremented by 1)

```
DO number = 1 TO 5
  SAY 'Hello!'
END
```

  - Using a control variable in DO/END loop (incremented by 2 using ‘BY’)

```
DO number = 1 TO 10 BY 2
  SAY 'Hello!'
END
```

Looping Instructions (Infinite loops)

- DO FOREVER infinite loop (!!!)

```
DO FOREVER
  Say ‘Hello!’
END
```

- EXIT instruction to terminate the DO loop – Both loop and exec will be terminated
- LEAVE instruction to terminate the DO loop – Control is passed to the instruction following END
- ITERATE instruction – Control is passed to the DO instruction at the beginning of the loop

Looping Instructions (Conditional loops)

- DO WHILE loop – Evaluates the expression before the loop executes the first time and repeat only when the expression is true

```
DO WHILE expression
  instruction(s)
END
```

- DO UNTIL loop - Tests the expression after the loop executes at least once and repeat only when the expression is false

```
DO UNTIL expression
  instruction(s)
END
```

Looping Instructions (Combining types of loops)

- Combining repetitive and conditional loops

```
quantity = 20
DO number = 1 TO 10 WHILE quantity < 50
   quantity = quantity + number
   SAY 'Quantity = 'quantity '  (Loop 'number')'
END
```

- Nested DO loops

```
DO outer = 1 TO 2
      DO inner = 1 TO 2
         SAY 'HIP'
      END
      SAY 'HURRAH'
END
```

Looping Instructions (Infinite loops)

- How to stop a EXEC in infinite loop:
  - Use attention interrupt key (PA1)
  - A message IRX0920I will be displayed
  - Type HI in response and press enter (HI = Halt interpretation)
  - If that doesn’t stop the loop then press PA1 key again and press HE (Halt Execution)

- **Note**: HI will not halt external function or host commands. It is applicable to only the current REXX exec.

Interrupt Instructions

- EXIT instruction causes an exec to unconditionally end and return to where the exec was invoked
- CALL instruction interrupts the flow of an exec by passing control to an internal or external subroutine. RETURN instruction returns control from a subroutine back to the calling exec and optionally returns a value.
- SIGNAL instruction, like CALL, interrupts the normal flow of an exec and causes control to pass to a specified label. Unlike CALL, SIGNAL does not return to a specific instruction to resume execution. (SIGNAL is similar to GOTO in COBOL. Avoid using it)

Using Functions

- Built-in functions -- These functions are built into the language processor.

- User-written functions -- These functions are written by an individual user.

  - Regardless of the kind of function, all functions return a value to the exec that issued the function call. To call a function, type the function name directly followed by one or more arguments within parentheses. There can be no space between the function name and the left parenthesis.

    function(arguments)

    A function call can contain up to 20 arguments separated by commas.

Built-In Functions

- Arithmetic functions
- Comparison functions
- Conversion functions
- Formatting functions
- String manipulating functions
- Miscellaneous functions

Arithmetic Built-In Functions

- ABS - Returns the absolute value of the input number.
- DIGITS - Returns the current setting of NUMERIC DIGITS.
- FORM - Returns the current setting of NUMERIC FORM.
- FUZZ - Returns the current setting of NUMERIC FUZZ.
- MAX - Returns the largest number from the list specified.
- MIN - Returns the smallest number from the list specified.
- RANDOM - Returns a quasi-random, non-negative whole number in the range specified.
- SIGN - Returns a number that indicates the sign of the input number.
- TRUNC - Returns the integer part of the input number, and optionally a specified number of decimal places.

Comparison Built-In Functions

- COMPARE - Returns 0 if the two input strings are identical. Otherwise, returns the position of the first character that does not match.
- DATATYPE - Returns a string indicating the input string is a particular data type, such as a number or character.
- SYMBOL - Returns this state of the symbol (variable, literal, or bad).

Conversion Built-In Functions

- B2X - Returns a string, in character format, that represents the input binary string converted to hexadecimal. (Binary to hexadecimal)
- C2D - Returns the decimal value of the binary representation of the input string. (Character to Decimal)
- C2X - Returns a string, in character format, that represents the input string converted to hexadecimal. (Character to Hexadecimal)
- D2C - Returns a string, in character format, that represents the input decimal number converted to binary. (Decimal to Character)
- D2X - Returns a string, in character format, that represents the input decimal number converted to hexadecimal. (Decimal to Hexadecimal)
- X2B - Returns a string, in character format, that represents the input hexadecimal string converted to binary. (Hexadecimal to binary)

Formatting Built-In Functions

- CENTER/CENTRE - Returns a string of a specified length with the input string centered in it, with pad characters added as necessary to make up the length.
- COPIES - Returns the specified number of concatenated copies of the input string.
- FORMAT - Returns the input number, rounded and formatted.
- JUSTIFY - Returns a specified string formatted by adding pad characters between words to justify to both margins.
- LEFT - Returns a string of the specified length, truncated or padded on the right as needed.
- RIGHT - Returns a string of the specified length, truncated or padded on the left as needed.
- SPACE - Returns the words in the input string with a specified number of pad characters between each word.

String Manipulating Functions

- ABBREV - Returns a string indicating if one string is equal to the specified number of leading characters of another string.
- DELSTR - Returns a string after deleting a specified number of characters, starting at a specified point in the input string.
- DELWORD - Returns a string after deleting a specified number of words, starting at a specified word in the input string.
- FIND - Returns the word number of the first word of a specified phrase found within the input string.
- INDEX - Returns the character position of the first character of a specified string found in the input string.
- INSERT - Returns a character string after inserting one input string into another string after a specified character position.
- LASTPOS - Returns the starting character position of the last occurrence of one string in another.
- LENGTH - Returns the length of the input string.

String Manipulating Functions

- OVERLAY - Returns a string that is the target string overlaid by a second input string.
- POS - Returns the character position of one string in another.
- REVERSE - Returns a character string, the characters of which are in reverse order (swapped end for end).
- STRIP - Returns a character string after removing leading or trailing characters or both from the input string.
- SUBSTR - Returns a portion of the input string beginning at a specified character position.
- SUBWORD - Returns a portion of the input string starting at a specified word number.
- TRANSLATE - Returns a character string with each character of the input string translated to another character or unchanged.
- VERIFY - Returns a number indicating whether an input string is composed only of characters from another input string or returns the character position of the first unmatched character.

String Manipulating Functions

- WORD - Returns a word from an input string as indicated by a specified number.
- WORDINDEX - Returns the character position in an input string of the first character in the specified word.
- WORDLENGTH - Returns the length of a specified word in the input string.
- WORDPOS - Returns the word number of the first word of a specified phrase in the input string.
- WORDS - Returns the number of words in the input string.

Miscellaneous Built-In Functions

- ADDRESS - Returns the name of the environment to which commands are currently being sent.
- ARG - Returns an argument string or information about the argument strings to a program or internal routine.
- BITAND - Returns a string composed of the two input strings logically ANDed together, bit by bit.
- BITOR - Returns a string composed of the two input strings logically ORed together, bit by bit.
- BITXOR - Returns a string composed of the two input strings eXclusive ORed together, bit by bit.
- CONDITION - Returns the condition information, such as name and status, associated with the current trapped condition.
- DATE - Returns the date in the default format (dd mon yyyy) or in one of various optional formats.
- ERRORTEXT - Returns the error message associated with the specified error number.

Miscellaneous Built-In Functions

- EXTERNALS Returns the number of elements in the terminal input buffer. In TSO/E, this function always returns a 0.
- LINESIZE - Returns the current terminal line width minus 1.
- QUEUED - Returns the number of lines remaining in the external data queue at the time when the function is invoked.
- SOURCELINE - Returns either the line number of the last line in the source file or the source line specified by a number.
- TIME - Returns the local time in the default 24-hour clock format (hh:mm:ss) or in one of various optional formats.
- TRACE - Returns the trace actions currently in effect.
- USERID - Returns the TSO/E user ID, if the REXX exec is running in the TSO/E address space.
- VALUE - Returns the value of a specified symbol and optionally assigns it a new value.
- XRANGE - Returns a string of all 1-byte codes (in ascending order) between and including specified starting and ending values.

Subroutines and Functions

- Calling a subroutine

  To call a subroutine, use the CALL instruction followed by the subroutine name (label or exec member name) and optionally followed by up to 20 arguments separated by commas. The subroutine call is an entire instruction.

  CALL subroutine_name argument1, argument2,...

- Calling a function

  To call a function, use the function name (label or exec member name) immediately followed by parentheses that can contain arguments. There can be no space between the function name and the parentheses. The function call is part of an instruction, for example, an assignment instruction.

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

Subroutines and Functions

- Returning a value from a subroutine

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

  RETURN value

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

- Returning a value from a function

  A function must return a value. When the function is a REXX exec, the value is returned with either the RETURN or EXIT instruction.

  RETURN value

  The calling exec receives the value at the function call. The value replaces the function call, so that in the following example, x = value.

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

Subroutines and Functions

The variables used within internal sub routine and functions can be protected using PROCEDURE instruction. When you use the PROCEDURE instruction immediately after the subroutine label, all variables used in the subroutine become local to the subroutine and are shielded from the main part of the exec. You can also use the PROCEDURE EXPOSE instruction to protect all but a few specified variables.

subroutine: PROCEDURE

subroutine: PROCEDURE EXPOSE number1

Compound Variables and stems

Compound variables are a way to create a one-dimensional array or a list of variables in REXX. Subscripts do not necessarily have to be numeric. A compound variable contains at least one period with characters on both sides of it.

E.g. var.5 Array.Row.Col

When working with compound variables, it is often useful to initialize an entire collection of variables to the same value. You can do this easily with a stem. A stem is the first variable name and first period of the compound variable. Thus every compound variable begins with a stem. The following are stems:

Var.

Array.

Employee. = ‘Nobody’ (Entire array is initialized with value ‘Nobody’)

Parsing Data

- PULL instruction
  - PARSE PULL
  - PARSE UPPER PULL
- ARG instruction
  - PARSE ARG
  - PARSE UPPER ARG
- PARSE VAR and PARSE UPPER VAR
  - PARSE VAR quote Word1 Word2 Word3

Parsing Data

- PARSE VALUE … WITH instruction

```
PARSE VALUE 'Knowledge is power.' WITH word1 word2 word3
            /* word1 contains 'Knowledge'  */
            /* word2 contains 'is'         */
            /* word3 contains 'power.'     */
```

Parsing Data using Template

- Blank:

The simplest template is a group of variable names separated by blanks. Each variable name gets one word of data in sequence except for the last, which gets the remainder of the data. The last variable name might then contain several words and possibly leading and trailing blanks.

```
PARSE VALUE ‘Knowledge is power' WITH  first last
    /* first contains ‘Knowledge'        */
    /* last contains ' is power'   */
```

When there are more variables than data, the extra variables are set to null.

A period in a template acts as a place holder. You can use a period as a "dummy variable" within a group of variables or at the end of a template to collect unwanted information.

Parsing Data using Template

- String:

  You can use a string in a template to separate data as long as the data includes the string as well. The string becomes the point of separation and is not included as data.

```
phrase = 'To be, or not to be?'       /* phrase containing comma   */
PARSE VAR phrase part1 ',' part2     /* template containing comma */
                                     /*  as string separator     */
    /* part1 contains 'To be'           */
    /* part2 contains ' or not to be?' */
```

  In this example, notice that the comma is not included with 'To be' because the comma is the string separator.

Parsing Data using Template

- Variable:

  When you do not know in advance what string to specify as separator in a template, you can use a variable enclosed in parentheses. The variable value must be included in the data.

```
separator = ','
phrase = 'To be, or not to be?'
PARSE VAR phrase part1 (separator) part2
    /* part1 contains 'To be'          */
    /* part2 contains ' or not to be?' */
```

Again, in this example, notice that the comma is not included with 'To be' because the comma is the string separator.

Parsing Data using Template

- Number:

  You can use numbers in a template to indicate the column at which to separate data. An unsigned integer indicates an absolute column position and a signed integer indicates a relative column position.

Parsing Data using Template

- Absolute column position: An unsigned integer or an integer prefixed with an equal sign (=) in a template separates the data according to absolute column position. The first segment starts at column 1 and goes up to, but does not include, the information in the column number specified. The subsequent segments start at the column numbers specified.

```
Quote = ‘Ignorance is bliss.’
PARSE VAR quote part1 5 part2
        /* part1 contains 'Igno'            */
        /* part2 contains 'rance is bliss.'  */
```

When each variable in a template has column numbers both before and after it, the two numbers indicate the beginning and the end of the data for the variable.

```
PARSE VAR quote 1 part1 10 11 part2 13 14 part3 19 1 part4 20
        /* part1 contains 'Ignorance'          */
        /* part2 contains 'is'                 */
        /* part3 contains 'bliss'              */
        /* part4 contains 'Ignorance is bliss.' */
```

Parsing Data using Template

- Relative column position: A signed integer in a template separates the data according to relative column position, that is, a starting position relative to the starting position of the preceding part. A signed integer can be either positive (+) or negative (-) causing the part to be parsed to shift either to the right (with a +) or to the left (with a -).

```
PARSE VAR quote part1 +5 part2 +5 part3 +5 part4
        /* part1 contains 'Ignor'        */
        /* part2 contains 'ance '        */
        /* part3 contains 'is bl'        */
        /* part4 contains 'iss.'         */
```

REXX in TSO/E Address space

- TSO/E REXX commands - Commands provided with the TSO/E implementation of the language. These commands do REXX-related tasks in an exec.
- Host commands - The commands recognized by the host environment in which an exec runs. A REXX exec can issue various types of host commands.
- When an exec issues a command, the REXX special variable RC is set to the return code.

REXX in TSO/E Address space

- Commands need to be enclosed in single or double quotes to differentiate them from other types of instruction (double quotes are recommended).

"ALLOC DA(‘USERID.NEW.DATA’) LIKE(‘USERID.OLD.DATA’) NEW"

“ALLOC DA('USERID.MYREXX.EXEC') F(SYSEXEC) SHR REUSE”

Dsname = USERID.MYREXX.EXEC

“ALLOC DA(‘”Dsname”') F(SYSEXEC) SHR REUSE”

Dsname = “’USERID.MYREXX.EXEC’”

“ALLOC DA(”Dsname”) F(SYSEXEC) SHR REUSE”

Changing the host environment

- Using ADDRESS instruction followed by host command environment name. (such as TSO, ISPEXEC, ISREDIT)

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

```
ADDRESS ispexec    /* Change the host command environment to ISPF */
"edit DATASET("dsname")"
```

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

```
ADDRESS ispexec "edit DATASET("dsname")"
```

Host Command Environment

- To find out what host command environment is currently active, use the ADDRESS built-in function.

```
x = ADDRESS()
```

- To check if a host command environment is available before trying to issue commands to that environment, issue the TSO/E REXX SUBCOM command followed by the name of the host command environment, such as ISPEXEC. If the environment is present, the REXX special variable RC returns a 0. If the environment is not present, RC returns a 1.

```
SUBCOM ISPEXEC
```

SIGL and SIGNAL ON ERROR

- The SIGL special variable is used in connection with a transfer of control within an exec because of a function, or a SIGNAL or CALL instruction.

- SIGL and the SIGNAL ON ERROR instruction can help determine what command caused an error and what the error was. When SIGNAL ON ERROR is included in an exec, any host command that returns a nonzero return code causes a transfer of control to a routine named "error". The error routine runs regardless of other actions that would normally take place, such as the display of error messages.

TSO/E External functions - LISTDSI

- LISTDSI - returns in variables the data set attributes of a specified data set. X=LISTDSI('USERID.DATA.SET')
- The variables populated by LISTDSI function:

SYSDSNAME	Data set name
SYSVOLUME	Volume serial ID
SYSUNIT	Device unit on which volume resides
SYSDSORG	Data set organization: PS, PSU, DA, DAU, IS, ISU, PO, POU, VS
SYSRECFM	Record format; three-character combination of the following: U, F, V, T, B, S, A, M
SYSLRECL	Logical record length
SYSBLKSIZE	Block size
SYSKEYLEN	Key length
SYSALLOC	Allocation, in space units
SYSUSED	Allocation used, in space units
SYSUSEDPAGES	Used space of a partitioned data set extended (PDSE)

TSO/E External functions - LISTDSI

- The variables populated by LISTDSI function (contd…):

SYSPRIMARY	Primary allocation in space units
SYSSECONDS	Secondary allocation in space units
SYSUNITS	Space units: CYLINDER, TRACK, BLOCK
SYSEXTENTS	Number of extents used
SYSCREATE	Creation date: Year/day format, for example: 1985/102
SYSREFDATE	Last referenced date: Year/day format, for example: 1985/107
SYSEXDATE	Expiration date: Year/day format, for example: 1985/365
SYSPASSWORD	Password indication: NONE, READ, WRITE
SYSRACFA	RACF indication: NONE, GENERIC, DISCRETE
SYSUPDATED	Change indicator: YES, NO
SYSTRKSCYL	Tracks per cylinder for the unit identified in the SYSUNIT variable
SYSBLKSTRK	Blocks per track for the unit identified in the SYSUNIT variable

TSO/E External functions - LISTDSI

- The variables populated by LISTDSI function (contd…):

SYSADIRBLK	Directory blocks allocated - returned only for partitioned data sets when DIRECTORY is specified
SYSUDIRBLK	Directory blocks used - returned only for partitioned data sets when DIRECTORY is specified
SYSMEMBERS	Number of members - returned only for partitioned data sets when DIRECTORY is specified
SYSREASON	LISTDSI reason code
SYSMSGLVL1	First level message if an error occurred
SYSMSGLVL2	Second level message if an error occurred
SYSDSSMS	Information about the type of a data set provided by DFSMS/MVS.
SYSDATACLASS	SMS data class name
SYSSTORCLASS	SMS storage class name
SYSMGMTCLASS	SMS management class name

TSO/E External functions - OUTTRAP

- OUTTRAP - The OUTTRAP function puts lines of command output into a compound variable. Using this function we can trap the terminal output of any command.

```
X=OUTTRAP('Var.')
Address TSO "LISTDS 'USERID.TEST.DATASET'"
X=OUTTRAP(OFF)
```

- Output of the LISTDS TSO command will be populated in Var. compound variable.

TSO/E External functions - SYSDSN

- The SYSDSN function determines if a specified data set is available for your use. If the data set is available for your use, it returns "OK".

```
x = SYSDSN(“'USERID.TEST.DATASET‘”)
/* x could be set to "OK"  */
```

- When a data set is not correct as specified or when a data set is not available, the SYSDSN function returns one of the following messages:

MEMBER SPECIFIED, BUT DATASET IS NOT PARTITIONED
MEMBER NOT FOUND
DATASET NOT FOUND
ERROR PROCESSING REQUESTED DATASET
PROTECTED DATASET
VOLUME NOT ON SYSTEM
UNAVAILABLE DATASET
INVALID DATASET NAME, data-set-name
MISSING DATASET NAME

Data Stack

- REXX in TSO/E uses an expandable data structure called a data stack to store information. The data stack combines characteristics of a conventional stack and queue.

- Stack – last in first out (LIFO)
- Queue – First in first out (FIFO)

Data Stack

- Adding Elements to the Data Stack: You can store information on the data stack with two instructions, PUSH and QUEUE.
  - PUSH – puts one item of data on the top of the data stack.
  - QUEUE – puts one item of data on the bottom of the data stack.

- Removing Elements from the Stack: To remove information from the data stack, use the PULL and PARSE PULL instructions.

```
PARSE PULL stackitem
SAY stackitem          /* displays first item retrieved from the data stack
   */
```

- Determining the Number of Elements on the Stack: The QUEUED built-in function returns the total number of elements on a data stack.

```
SAY QUEUED()        /* displays a decimal number */
```

Buffer on a Data Stack

- MAKEBUF command creates a buffer, which you can think of as an extension to the stack.
- DROPBUF command deletes the buffer and all elements within it.

- An exec can create multiple buffers before dropping them. Every time MAKEBUF creates a new buffer, the REXX special variable RC is set with the number of the buffer created. Thus if an exec issues three MAKEBUF commands, RC is set to 3 after the third MAKEBUF command.
- Issuing QBUF command populates RC with number of buffers created.
- QELEM – returns the number of items present in recently created buffer in Rexx special variable RC.

Protecting elements in a Data Stack

- NEWSTACK command creates a private data stack that is completely isolated from the original data stack. The elements on the original data stack cannot be accessed by an exec or the routines that it calls until a DELSTACK command is issued.

- The DELSTACK command removes the most recently created data stack. If no stack was previously created with the NEWSTACK command, DELSTACK removes all the elements from the original stack.

- QSTACK returns the total number of stacks.

Dynamic modification of expression

- INTERPRET: Rexx interpreter can be invoked with in a EXEC using INTERPRET command.

- E.g.

```
Expression = 'Say 5+5'          /* Variable Expression contains Rexx
    instruction */
Say Expression                  /* Displays 'Say 5+5' on the terminal */
INTERPRET(Expression)           /* Displays 10 on the terminal */
```

Processing datasets - EXECIO

- EXECIO: An exec uses the EXECIO command to perform the input and output (I/O) of information to and from a data set. The information can be stored in the data stack for serialized processing or in a list of variables for random processing.

- Before using EXECIO, the data set must first be allocated to a ddname using TSO command ALLOC.

  "ALLOC DA(‘userid.input.dataset’) F(myindd) SHR REUSE"

Processing datasets - EXECIO

- Reading all records from the dataset into a compound variable:

  "EXECIO * DISKR myindd (FINIS STEM newvar.“

  - newvar.0 will contain the total number of records populated in the stem.

- Reading all records from the dataset into the data stack:

  "EXECIO * DISKR myindd (FINIS"

  - QUEUED() function will give the total number of records populated in the stem.

Processing datasets - EXECIO

- Reading all records starting from record 100:
  "EXECIO * DISKR myindd 100 (FINIS“
- Reading 6 records starting from record 50:
  "EXECIO 6 DISKR myindd 50 (FINIS“
- Open a dataset without reading:
  "EXECIO 0 DISKR myindd (OPEN"
- Read a record from dataset for updating:
  "EXECIO 1 DISKRU myindd (OPEN"

Processing datasets - EXECIO

- Skip 24 record without reading and place file pointer on next record:

  "EXECIO 24 DISKR myindd (SKIP"

- To read the information in LIFO order onto the stack. In other words, use the PUSH instruction to place the information on the stack :

  "EXECIO * DISKR myindd (LIFO"

- To read the information in FIFO order onto the stack. In other words, use the QUEUE instruction to place the information on the stack :

  "EXECIO * DISKR myindd (FIFO“

  - Note: FIFO is the default if no option is specified.

Processing datasets - EXECIO

- Write records from a data stack into the dataset:

  "EXECIO * DISKW myoutdd (FINIS"

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

    QUEUE ''

- Write all records from a stem into the dataset:

  "EXECIO * DISKW myoutdd (FINIS STEM newvar."

- Write 25 records from a data stack into the dataset:

  "EXECIO 25 DISKW myoutdd (FINIS STEM newvar."

Processing datasets - EXECIO

Return Code	Meaning
0	Normal completion of requested operation.
1	Data was truncated during DISKW operation.
2	End-of-file reached before the specified number of lines were read during a DISKR or DISKRU operation. (This return code does not occur when * is specified for number of lines because the remainder of the file is always read.)
4	An empty data set was found within a concatenation of data sets during a DISKR or DISKRU operation. The file was not successfully opened and no data was returned.
20	Severe error. EXECIO completed unsuccessfully and a message is issued.

Allocating datasets in a REXX EXEC

- Using ALLOC command in TSO we can allocate new dataset with in REXX exec.

- Allocating a PS

"ALLOCATE DA('“userid.new.dataset"') NEW SPACE(1,1) DSORG(PS) RECFM(F,B) LRECL(133) BLKSIZE(26600)"

- Allocating a PDS

"ALLOCATE DA('“userid.new.dataset"') NEW DIR(10) SPACE(1,1) DSORG(PO) RECFM(F,B) LRECL(133) BLKSIZE(26600)"

- Using a model dataset:

"ALLOC DA(‘”userid.new.data”’) LIKE(‘”userid.old.data”’) NEW"

Allocating datasets to system DD

- REXX program can be run implicitly if the library containing the EXEC is allocated to system ddnames SYSEXEC or SYSPROC.
- "ALLOC FILE(SYSEXEC) DA(‘USERID.REXX.TOOLS') SHR REUSE“
- Run TSO ISRDDN command to check the status of allocation.

ISPF Edit Macro

- Using ISPF EDIT Macro written in REXX we can invoke all services provided by ISPF Editor.
- The REXX exec should contain MACRO command to indicate that this is a REXX macro.
  - ADDRESS ISREDIT “MACRO”
  - Example of a EDIT MACRO. This MACRO issues ISPF find command through REXX exec.

```
/*  REXX */
ADDRESS ISREDIT "MACRO"
STR = ‘SOME DATA'
ADDRESS ISREDIT
"FIND " STR " ALL"
```

ISPF Panels

- If SDF II can be used to design ISPF panels also.
- In the PROFILES -> SYSTEM ENVIRONMENT set the option to ISPF.

- Sections in a ISPF Panel
  - ATTR – to assign meaning for attributes
  - BODY – Structure of Panel
  - INIT – Initialization section of the Panel
  - PROC – Procedural section for PFkey pressing
  - END – End of the Panel

ISPF Panels

- To display a panel the panel library must be defined to ISPPLIB system library.

```
ADDRESS ISPEXEC
"CONTROL ERRORS RETURN"
"LIBDEF ISPPLIB DATASET ID(‘USERID.PANEL.LIB') STACK"
"DISPLAY PANEL(PAAN1001)"
```

ISPF File Tailoring

- File Tailoring using ISPF skeleton:

  Skeletons are pre-prepared templates. ISPF provides a powerful feature called file tailoring, which can be used on skeletons to configure them on-the-fly without destroying the original template.

- Skeleton should be defined as a member of PDS. The variables are referred by prefixing ‘&’ to the variable name.

```
)CM This is test skeleton
HELLO &name
NICE MEETING YOU
)SEL &resp = Y
DO NOT FORGET TO TAKE RAIN COAT
)ENDSEL
```

  -)CM denotes the comment
  - &name and &resp are the variables to be populated in REXX EXEC before calling file tailoring service.
  -)SEL and)ENDSEL denotes the optional part in the output file.

ISPF File Tailoring

- Skeleton should be defined as a member of PDS. The variables are referred by prefixing ‘&’ to the variable name.

```
/*REXX*/
&name = ‘your name’;
&resp = ‘Y’
/***** INVOKING  ISPF FILE TAILORING SERVICES *****/
ADDRESS ISPEXEC
"ISPEXEC LIBDEF ISPSLIB DATASET ID('“USERID.REXX.SKELETON"')"
"ISPEXEC LIBDEF ISPFILE DATASET ID('“USERID.REXX.OUTPUT"')"
"FTOPEN"
"FTINCL TESTSKEL"
"FTCLOSE NAME(SKELOUT) LIBRARY(ISPFILE)"
EXIT
```

  - The output file will be saved in USERID.REXX.OUTPUT(SKELOUT).

REFERENCES:

- Rexx Language Association http://www.rexxla.org/
- Rexx Language page at IBM Hursley http://www2.hursley.ibm.com/rexx/
- An online Rexx tutorial http://www.kyla.co.uk/other/rexx1.htm
- An online Rexx tutorial http://www.ilook.fsnet.co.uk/index/rexx_idx.htm
- Refer to REXX programming guides in TSO/E and ISPF book shelves in the IBM book manager.

FINALLY…….

THANK YOU