IBM Rexx Language Update: Classic Rexx and The Rexx Compiler

Virgil Hein IBM
vhein@us.ibm.com

March 2018

Disclaimers

- The information contained in this presentation is provided for informational purposes only.
- While efforts were made to verify the completeness and accuracy of the information contained in this presentation, it is provided “as is”, without warranty of any kind, express or implied.
- In addition, this information is based on IBM’s current product plans and strategy, which are subject to change by IBM without notice.
- IBM shall not be responsible for any damages arising out of the use of, or otherwise related to, this presentation or any other documentation.
- Nothing contained in this presentation is intended to, or shall have the effect of:
  - Creating any warranty or representation from IBM (or its affiliates or its or their suppliers and/or licensors); or
  - Altering the terms and conditions of the applicable license agreement governing the use of IBM software.

Agenda

- HLASM TextBook
- REXX products
- External environments and interfaces
- Key functions and instructions
- REXX compound variables vs. data stack
- I/O
- Troubleshooting
- Programming style and techniques
- REXX Enhancements (z/OS)

HLASM TextBook

- HLASM TextBook V 1.00
  - Marist College web site:
    http://idcp.marist.edu/enterprisesystemseducation/Assembler%20Language%20Programming%20for%20IBM%20z%20System%20Servers.pdf

- HLASM TextBook V 2.00
  - PDF: Assembler Language Programming for IBM Systems Z Servers V2.00

Assemb Lang
Programming

REXX Interpreter and Libraries

- The Interpreter executes (interprets) REXX code “line by line”
  - Included in all z/OS and z/VM releases
- A REXX library is required to execute compiled programs
  - Compiled REXX is not an LE language
- Two REXX library choices:
  - (Runtime) Library – a priced IBM product
  - Alternate library – a free IBM download
    - Uses the native system’s REXX **interpreter**
- At execution, compiled REXX will use whichever library is available:
  - (Runtime) Library
  - Alternate Library

The REXX Products

- IBM Compiler for REXX on zSeries Release 4
  - z/VM, z/OS: product number 5695-013
- IBM Library for REXX on zSeries Release 4
  - z/VM, z/OS: product number 5695-014
- z/VSE
  - Part of operating system
- IBM Alternate Library for REXX on zSeries Release 4
  - Included in z/OS base operating system (V1.9 and later)
  - Free download for z/VM (and z/OS)
    - http://www.ibm.com/software/awdtools/rexx/rexxzseries/altlibrary.html
- REXX Interpreter
  - Included in all z/OS and z/VM releases

Why Use a REXX Compiler?

- Program performance
  - Known value propagation
  - Assign constants at compile time
  - Common sub-expression elimination
  - stem.i processing
- Source code protection
  - Source code not in deliverables
- Improved productivity and quality
  - Syntax checks all code statements
  - Source and cross reference listings
- Compiler control directives
  - %include, %page, %copyright, %stub, %sysdate, %systime, %testhalt

REXX Compiler Issues / Updates

- Interpreter issue:
  - If daylight saving time ended while a long-running REXX program was executing, the REXX elapsed time failed due to a negative interval
    - Fixed in Interpreter
    - Queued to be fixed in the Compiler
- 11 PMRs addressed in Compiler in 2016 (no common pattern)
- 1 REXX library APAR (first APAR for several years)
  - Occasional program check due to scanning past the end of a page when checking whether a stem tail was numeric if the length was zero (which wrapped to 255).
- Reported problem (PMR)
  - Time stamp on compiler listing incorrect by 27 seconds
    - Does not take leap seconds into account
    - Leap second support added to MVS in 1993
    - REXX compiler PMR 03766,124,848 noting time stamp fails to allow for leap seconds
      - PTF available, awaiting customer feedback

REXX User Example 1

- Received email noting:
  - Have successfully copied large datasets to 64-bit memory and manipulated them using STORAGE function.
  - Your samples also appear in the REXX Reference manual with several typos, e.g. mixing 24 and 25 bytes when retrieving data.
    - One obvious matter - left to the clever programmers - is that a 64-bit memory object must pre-exist before you can copy data into it.
    - User wrote REXX functions to create (IARV64 REQUEST=GETSTOR) and delete (IARV64 REQUEST=DETACH) such memory objects. Commented that REXX should provide such functions? (Or at least that the documentation should mention that objects must exist before 64-bit storage can be successfully referenced? - - STORAGE function could do it behind the covers, but it does not.)
    - User assisted client (Lloyds Banking Group) utilizing the above, noting that “**64 bit access by STORAGE solved a huge practical problem**” (3 week effort).

REXX Reference manual used was: SA32-0972 TSO/E REXX Reference.

z/OS V2R2 TSO/E REXX Reference (SA32-0972-02)

REXX User Example 2

- Received email noting:
  - Currently I work an ReXX to update some error messages in a DB2-Table and face the problem that the value of the error message is set to upper case by calling DSNREXX. (But in Germany we use mixed case for everything, esp. error messages ;-))
  - I double checked the ReXX-Variable for the error message and it contains mixed case before calling DSNREXX.
  - In your presentation you suggested to use “Use upper case for calls to external routines (commands)” for Capitalization. Is this really a suggestion or does DSNREXX always set everything to upper case?
  - If yes/no - do you have an idea how I can force DSNREXX to avoid translating my error message to upper case?

  - There is no rule on whether or where you use all lower-case, mixed case, or all caps in REXX. It is strictly a style argument, and the suggestions the presentation makes are included (as a FYI) accordingly.
    - Note that if one prefers to use some other style or standard, they may certainly do so.

REXX External Environments

External Environments

- ADDRESS instruction is used to define the external environment to receive host commands
  - For example, to set TSO/E as the environment to receive commands

```
ADDRESS  TSO
```

- Several host command environments available in z/OS
- A few host command environments available in z/VM

Host Command Environments in z/OS

- TSO
  - Used to run TSO/E commands like ALLOCATE and TRANSMIT
  - Only available to REXX running in a TSO/E address space
  - The default environment in a TSO/E address space
  - Example:

```
Address TSO “ALLOC FI(INDD) DA(‘USERID.SOURCE’) SHR”
```

- MVS
  - Use to run a subset of TSO/E commands like EXECIO and MAKEBUF
  - The default environment in a non-TSO/E address space
  - Example:

```
Address MVS “EXECIO * DISKR MYINDD (FINIS STEM MYVAR”
```

Host Command Environments in z/OS

- ISPEXEC
  - Used to invoke ISPF services like DISPLAY and SELECT
  - Only available to REXX running in ISPF
  - Example:

```
Address ISPEXEC “DISPLAY PANEL(APANEL)”
```

- ISREDIT
  - Used to invoke ISPF edit macro commands like FIND and DELETE
  - Only available to REXX running in an ISPF edit session
  - Example:

```
Address ISREDIT “DELETE .ZFIRST .ZLAST”
```

- Many more!

Host Command Environments in z/OS …

- CONSOLE
- LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM
- SYSCALL
- SDSF
- DSNREXX

Host Command Environments in z/OS …

- CONSOLE
  - Used to invoke MVS system and subsystem commands
  - Only available to REXX running in a TSO/E address space
  - Requires an extended MCS console session
  - Requires CONSOLE command authority
  - Example:

```
“CONSOLE ACTIVATE”
 Address Console “D A”     /* Display system activity */
“CONSOLE DEACTIVATE”
```

Result:

```
IEE114I 04.50.01 2011.173 ACTIVITY 602        ACTIVE/MAX VTAM   OAS
 JOBS      M/S    TS USERS SYSAS    INITS
00002    00014    00002      00032    00005    00001/00020    00010
```

Host Command Environments in z/OS …

- LINK, LINKMVS, LINKPGM, ATTACH, ATTCHMVS, ATTCHPGM
  - Host command environments for linking to and attaching unauthorized programs
  - Available to REXX running in any address space
  - LINK & ATTACH – can pass one character string to program
  - LINKMVS & ATTCHMVS – pass multiple parameters; half-word length field precedes each parameter value
  - LINKPGM & ATTCHPGM – pass multiple parameters; no half-word length field
  - Example:

```
“FREE FI(SYSOUT SORTIN SORTOUT SYSIN)”
 “ALLOC FI(SYSOUT)   DA(*)”
 “ALLOC FI(SORTIN)   DA('VANDYKE.SORTIN') REUSE”
 “ALLOC FI(SORTOUT)  DA('VANDYKE.SORTOUT') REUSE”
 “ALLOC FI(SYSIN)    DA('VANDYKE.SORT.STMTS') SHR REUSE”
 sortparm = “EQUALS”
 Address LINKMVS “SORT sortparm”
```

Host Command Environments in z/OS …

- SYSCALL
  - Used to invoke interfaces to z/OS UNIX callable services
  - The default environment for REXX run from the z/OS UNIX file system
  - Use syscalls('ON') function to establish the SYSCALL host environment for a REXX run from TSO/E or MVS batch
  - Example:

```
Call Syscalls 'ON'
Address Syscall 'readdir / root.'
Do i=1 to root.0
 Say root.i
End
```

Result:

```
…
bin
Dev
etc
   …
```

Host Command Environments in z/OS ...

- SDSF
  - Used to invoke interfaces to SDSF panels and panel actions
  - Use isfcalls('ON') function to establish the SDSF host environment
  - Use the ISFEXEC host command to access an SDSF panel
  - Panel fields returned in stem variables
  - Use the ISFACT host command to take an action or modify a job value
  - Example:

```
rc=ISFCalls("ON")
Address SDSF "ISFEXEC ST"
Do ix = 1 to JNAME.0
If Pos("MYREXX",JNAME.ix) = 1 Then
  Do
    say "Cancelling job ID" JOBID.ix "for MYREXX"
    Address SDSF "ISFACT ST TOKEN('"TOKEN.ix"')
  PARM(NP
P)"
   End
End
rc=ISFCalls("OFF")
Exit
```

Host Command Environments in z/OS …

- DSNREXX
  - Provides access to DB2 application programming interfaces from REXX
  - Any SQL command can be executed from REXX
    - Only dynamic SQL supported from REXX
  - Use RXSUBCOM to make DSNREXX host environment available
  - Must CONNECT to required DB2 subsystem
  - Can call SQL Stored Procedures
  - Example:

```
RXSUBCOM(‘ADD’,’DSNREXX’,’DSNREXX’)
subSys = ‘DB2PRD’
Address DSNREXX “CONNECT” subsys
owner = ‘PRODTBL’
recordkey = ‘ROW2DEL’
sql_stmt = “DELETE * FROM” owner”.MYTABLE” ,
           "WHERE TBLKEY = ‘”recordkey”’”
Address DSNREXX “EXECSQL EXECUTE IMMEDIATE”
sql_stmt Address DSNREXX “DISCONNECT”
```

Other External Environments in z/OS

- IPCS
  - Used to invoke IPCS subcommands from REXX
  - Only available when run from in an IPCS session
- CPICOMM, LU62, and APPCMVS
  - Supports the writing of APPC/MVS transaction programs (TPs) in REXX
  - Programs can communicate using SAA common programming interface (CPI) communications calls and APPC/MVS calls

Other "Environments" and Interfaces in z/OS

- System REXX
  - A function package that allows REXX EXECs to be executed outside of conventional TSO/E and Batch environments
  - Can be invoked using assembler macro interface AXREXX or through an operator command
  - Easy way for Web Based Servers to run commands/functions and get back pertinent details
  - EXEC runs in problem state, key 8, in an APF authorized address space under the MASTER subsystem
  - Two modes of execution

• TSO=NO	runs in MVS host environment address space shared with up to 64 other EXECs limited data set support
• TSO=YES	runs isolated in a single address space can safely allocate data sets does not support all TSO functionality

Other “Environments” and Interfaces . . .

- RACF Interfaces
  - IRRXUTIL
    - REXX interface to R_admin callable service (IRRSEQ00) extract request
    - Stores output from extract request in a set of stem variables

```
myrc=IRRXUTIL(“EXTRACT”,”FACILITY”,”BPX.DAEMON”,”RACF”,””,”FALSE”)
Say “Profile name: “||RACF.profile
Do a=1 to RACF.BASE.ACLCNT.REPEATCOUNT
   Say “ “||RACF.BASE.ACLID.a||”:”||RACF.BASE.ACLACS.a
 End
```

  - RACVAR function
    - Provides information from the ACEE about the running user
    - Arguments: USERID, GROUPID, SECLABEL, ACEESTAT

```
If racvar(‘ACEESTAT’) <> ‘NO ACEE’ Then
   Say “You are connected to group “ racvar(‘GROUPID’)”.”
```

Other "Environments" and Interfaces . . .

- Other ISPF Interfaces
  - Panel REXX
    - Allows REXX to be run in a panel procedure
    - *REXX statement used to invoke it
    - REXX can be coded directly in the procedure or taken from a SYSEXEC or SYSPROC DD member
    - REXX can modify the values of ISPF variables
  - File Tailoring Skeleton REXX
    - Allows REXX to be run in a skeleton
    -)REXX control statement used to invoke it
    - REXX can be coded directly in the procedure or taken from a SYSEXEC or SYSPROC DD member
    - REXX can modify the values of ISPF variables

Host Command Environments in z/VM

- CMS (default)
  - Commands treated as if entered on the CMS command line
    - Translation of parameter list
      - **Uppercasing and tokenizing**
  - Same search order as CMS command line
- COMMAND
  - Basic CMS CMSCALL command resolution
    - No translation of parameter list
      - **No uppercasing of tokenized parameter lists**
    - To call an EXEC, **prefix the command** with the word EXEC
    - To send a command to CP, use the prefix CP
- CPICOMM, CPIRR, OPENVM
- Generally, best practice is to use “Address Command” at the top of REXX EXECs that will be run in CMS environment

Key Instructions and Functions

Instructions vs. Functions

- Keyword instruction
  - One or more clauses
  - First word is a keyword that identifies the instruction

```
Arg, Do, If, Parse, …
```

- Instruction
  - Statement that performs an assignment of a value to a variable

```
counter = 1
```

- Function
  - Must return a single result string (i.e. must be on the right side of an equal sign)
  - Built-in - provided as part of the REXX language
  - Internal - create your own
  - External – create your own or use platform unique functions
- Subroutine
  - Called like a function, but may not return data

Key Instructions – Parse

- Parse
  - Allows the use of a template to split a source string into multiple components
  - Syntax:

```
>>-PARSE--+-------+--+-ARG-----------------------+-------------->
          '-UPPER-'  +-EXTERNAL------------------+
                     +-NUMERIC-------------------+
                     +-PULL----------------------+
                     +-SOURCE--------------------+
                     +-VALUE-+------------+-WITH-+
                     |       '-expression-'      |
                     +-VAR--name-----------------+
                     '-VERSION-------------------'

>--+---------------+--;----------------------------------------><
   '-template_list-'
```

- **Short forms** to some of these commands exist
  - **NOT RECOMMENDED**
  - But you may see them in another user’s code you must maintain
    - ARG
      - Short form for PARSE UPPER ARG
    - PULL
      - Short form for PARSE UPPER PULL

Parse Templates

- Simple template
  - Divides the source string into **blank-delimited** words and assigns them to the variables named in the template
    - The last variable gets the rest of the string exactly as entered

```
datastring = ‘   Write the     blank-delimited  string ’
Parse Var datastring firstvar secondvar thirdvar fourthvar

firstvar  -> ‘Write’
secondvar -> ‘the’
thirdvar  -> ‘blank-delimited’
fourthvar -> ‘   string  ’
```

Parse Templates

- Simple template
  - A period is a placeholder in a template
    - A “dummy” variable used to collect unwanted data
    - Notice the double quotes so the single quote is recognized as part of the string

```
datastring = “Last one gets what's left”
Parse Var datastring firstvar . secondvar

firstvar -> “Last”
secondvar -> “gets what’s left”
```

    - Often used at the end of Parse statement to take “the rest of the data”

```
datastring = “Last one gets what's left”
Parse Var datastring firstvar secondvar .

firstvar -> “Last”
secondvar -> “one”
```

    - Causes the last variable to get the last word without leading and trailing blanks

```
datastring = ‘   Write the   blank-delimited  string ’
 Parse Var datastring firstvar secondvar thirdvar fourhvar
.     firstvar -> ‘Write’
secondvar -> ‘the’
thirdvar -> ‘blank-delimited’
fourthvar -> ‘string’
```

Parse Templates . . .

- String pattern template
  - A **literal or variable string pattern** indicating where the source string should be split
  - Assumes blank-delimited if no other pattern specified

```
datastring = '  Write the    blank-delimited   string  '
```

Literal:

Literal delimited

```
Parse Var datastring firstvar '-' secondvar .
```

Blank delimited

Variable:

```
delim = '-'
Parse Var datastring firstvar (delim) secondvar .
```

Result (the same in both cases):

```
firstvar -> '  Write the    blank'
secondvar -> 'delimited'
```

Parse Templates . . .

- Positional pattern template
  - Use numeric values to identify the **character positions** at which to split data in the source string
  - An absolute positional pattern is a number or a number preceded by an equal sign

```
----+----1----+----2----+----3----+----4----+
     Datastring = ‘Cowlishaw          Mike           UK          ’
     Parse Var datastring =1
                             surname =20 chrname =35 country =46 .

     surname -> ‘Cowlishaw                ’
     chrname -> ‘Mike           ’
     country -> ‘UK          ’
```

  - A relative positional pattern is a number preceded by a plus or minus sign
    - Plus or minus indicates movement right or left, respectively, from the last match

```
              ----+----1----+----2----+----3----+----4----+
     datastring = ‘Cowlishaw          Mike           UK          ’
     Parse Var datastring =1 surname +19 chrname +15 country +11 .

     surname -> ‘Cowlishaw      ’
     chrname -> ‘Mike          ’
     country -> ‘UK         ’
```

Parse Templates . . .

- Positional pattern template – removing blanks
  - Specify an absolute positional pattern
  - Insert periods to strip blanks

```
              ----+----1----+----2----+----3----+----4----+
datastring = ‘Cowlishaw          Mike           UK     ’
Parse Var datastring =1 surname . =20 chrname . =35 country .

surname -> ‘Cowlishaw’
chrname -> ‘Mike’
country -> ‘UK’
```

If data starts in column 1 and is blank-delimited, this is the same as

```
Parse Var datastring surname chrname country
```

- **Warning –** won’t work if any of the data has more than one “word”

```
              ----+----1----+----2----+----3----+----4----+
datastring = ‘Cowlishaw, Jr.     Mike           UK         ’
Parse Var datastring =1 surname . =20 chrname . =35 country .

surname -> ‘Cowlishaw,’
chrname -> ‘Mike’
country -> ‘UK’
```

Blank delimited

Compound Variables and Data Stack

What is a Compound Variable?

- A way to reference a collection of related values
  - Also called a *stem variable* or *stem array*
- Variable name is *stem* followed by zero or more *tails*
  - *stem* must be simple variable ending in a period
  - *tail* must be simple variable or decimal integer
  - Multiple *tails* are separated by periods
- Each *tail* variable is replaced by its value
  - Default value of *stem* and *tail* is the variable names used for *stem* and *tail*
  - Each *tail* references a dimension of the collection
- The resulting *derived name* is used to access a specific value from the collection
- Tails which are variables are replaced by their respective values
  - If no value assigned, takes on the uppercase value of the variable name

```
day.1                              stem:  DAY.
                                   tail:  1

array.j                            stem:  ARRAY.
                                   tail:  J

name = ‘Smith’
phone = 12345

employee.name.phone                stem:  EMPLOYEE.
                                   tail:  Smith.123
                                          45
```

Compound Variable Values

- Initializing a stem to some value automatically initializes every compound variable with the same stem to the same value

```
Say month.12  ⟶  MONTH.12
month. = 'Unknown'
month.3 = 'March'
month.6 = 'June'

Say month.12  ⟶  Unknown
monthnum = 3
Say month.monthnum  ⟶  March
```

- Easy way to reset the values of compound variables

```
month. = ''
Say month.6  ⟶  ''
```

- Drop instruction can be used to restore compound variables to their uninitialized state

```
Drop month.
Say month.6  ⟶  MONTH.6
```

Processing Compound Variables

- Compound variables provide the ability to process one-dimensional arrays
  - Use a numeric value for the tail
  - **Good practice** - store the number of array entries in the compound variable with a tail of 0 (zero)
  - Often processed in a **Do** loop using the tail as the loop control variable

```
 invitee.0 = 10
Do j = 1 to invitee.0
  Say ‘Enter the name for invitee’ j
  Parse Pull invitee.j
End
```

- Stems can be used with I/O functions to read data from and write data to a file on z/VM or data set on z/OS
  - Stream I/O
  - EXECIO
  - PIPE
- Stems can also be used with the external function OUTTRAP (z/OS) or PIPE (z/VM) to capture output from commands

Processing Compound Variables . . .

- The tail for a compound variable can be used as an index to related data
- The tail (index) and data can contain blanks
- Given the following input data:

```
----+----1----+----2----+----3----+----4----+
Employee# Name          Location
A1234     M Cowlishaw   United Kingdom
B5678     T Dean        Portland
C9012     V Hein        Austin
```

- The unique employee number value can be used as the tail of compound variables that hold the rest of the person’s data

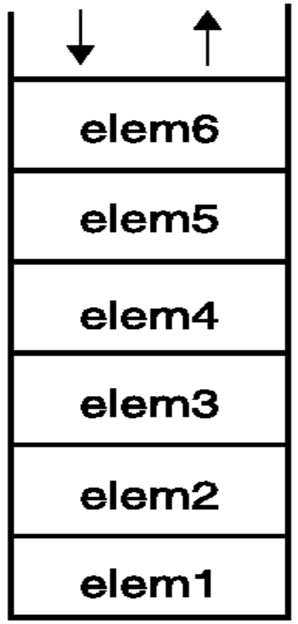
```
'PIPE < EMPLOYEE INFO A | STEM rec.'
 Do j = 2 To rec.0
  Parse Var rec.j =1 empnum name.empnum =25 location.empnum
 End j
 Say 'Which employee number do you want to learn about?'
 Parse Upper Pull empnum
 Say 'The name of employee' empnum 'is' Strip(name.empnum)'.'
 Say 'The location of employee' empnum 'is' Strip(location.empnum)'.’
 Exit
```

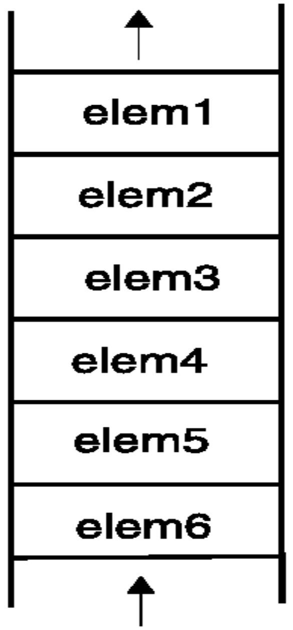
What is a Data Stack?

- An expandable data structure used to temporarily hold data items (elements) until needed
- When an element is needed it is **always removed** from the **top** of the stack
- A new element can be **added** either to the **top** (LIFO) or the **bottom** (FIFO) of the stack
  - FIFO stack is often called a queue

LIFO

Stack





FIFO

Stack

(Queue)

Manipulating the Data Stack

- 3 basic REXX <u>instructions</u>
  - **Push** - put one element on the <u>top</u> of the stack

    ```
    elemone = ‘new top element’
     Push elemone
    ```

  - **Queue** - put one element on the <u>bottom</u> of the stack

    ```
    elemtwo = ‘new bottom element’
     Queue elemtwo
    ```

  - **Parse Pull** - remove an element from the (top) of the stack

    ```
    Parse Pull nextthing
    ```

  - Result:

    ```
    nextthing → ‘new top element’
    ```

- 1 REXX <u>function</u>
  - **Queued()** - returns the number of elements in the stack

    ```
    num_elems = Queued()
    ```

Why Use the Data Stack?

- To store a large number of data items for later use
  - Size may be unpredictable or unknown
- Pass a large or unknown number of arguments between EXECs or routines
- Specify commands to be run when the EXEC ends
  - Elements left on the data stack when an EXEC ends are treated as commands

```
Queue “TSOLIB RESET QUIET”
Queue “ALLOC FI(ISPLLIB) DA(‘ISP.SISPLOAD’
'SYS1.DFQLLIB‘) SHR REUSE”
Queue “TSOLIB ACTIVATE FILE(ISPLLIB) QUIET”
Queue “ISPF”
```

- Pass responses to an interactive command that runs when the EXEC ends
  - Example: z/VM DDR program

Quick Example of Processing the Data Stack

- A receiving (or called) program collects data from the stack
  - Passed from sending/calling program

```
/* Sample stack processing */
Address Command
element.0 = Queued()
Do i = 1 To element.0
  Parse Pull element.i
  …
End
…
```

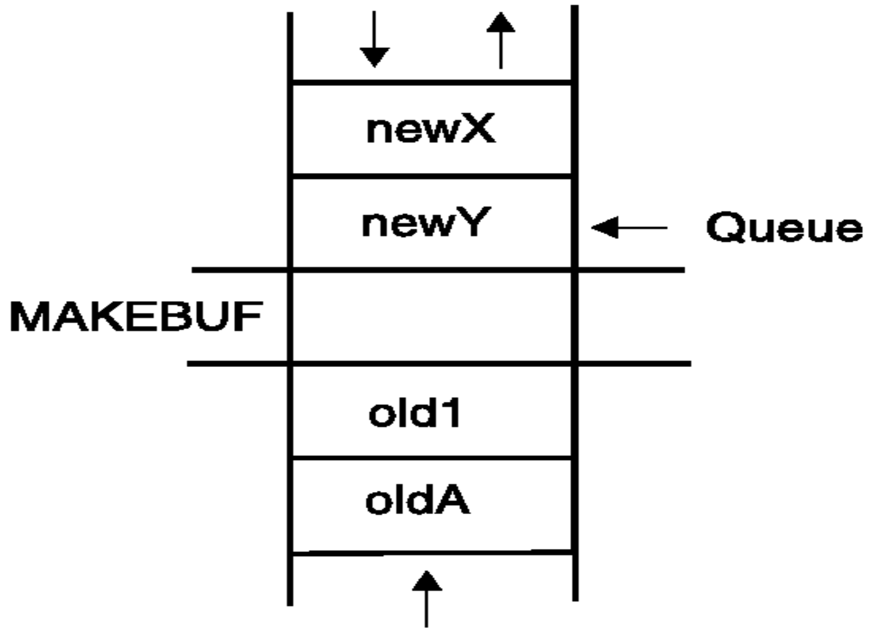
Full parsing capability

More Stack Functions and Options

- Buffers
- Additional stacks
- Some functions are z/OS only

Using Buffers in the Data Stack

- An EXEC can create a buffer in a data stack using the **Makebuf** command
- All elements added after a Makebuf command are placed in the **new buffer**
  - Makebuf changes where the Queue instruction inserts new elements
    - Remember **Queue** inserts at the **"bottom"** of the stack (or **buffer**)



Using Buffers in the Data Stack . . .

- An EXEC can use **Makebuf** to **create** multiple buffers in the data stack
  - Makebuf returns in the RC variable the number identifying the newly created buffer
- **Dropbuf** command is used to **remove** a buffer from the data stack
  - Allows an EXEC to easily remove temporary storage assigned to the data stack
  - A buffer number can be specified with Dropbuf to identify the buffer to remove
    - Default is to remove the most recently created buffer
  - Dropbuf 0 results in an empty data stack (use with caution)
- z/OS only
  - The **Qbuf** command is used to find out **how many buffers** have been created
  - The **Qelem** command is used to find out the **number of elements** in the most recently created buffer

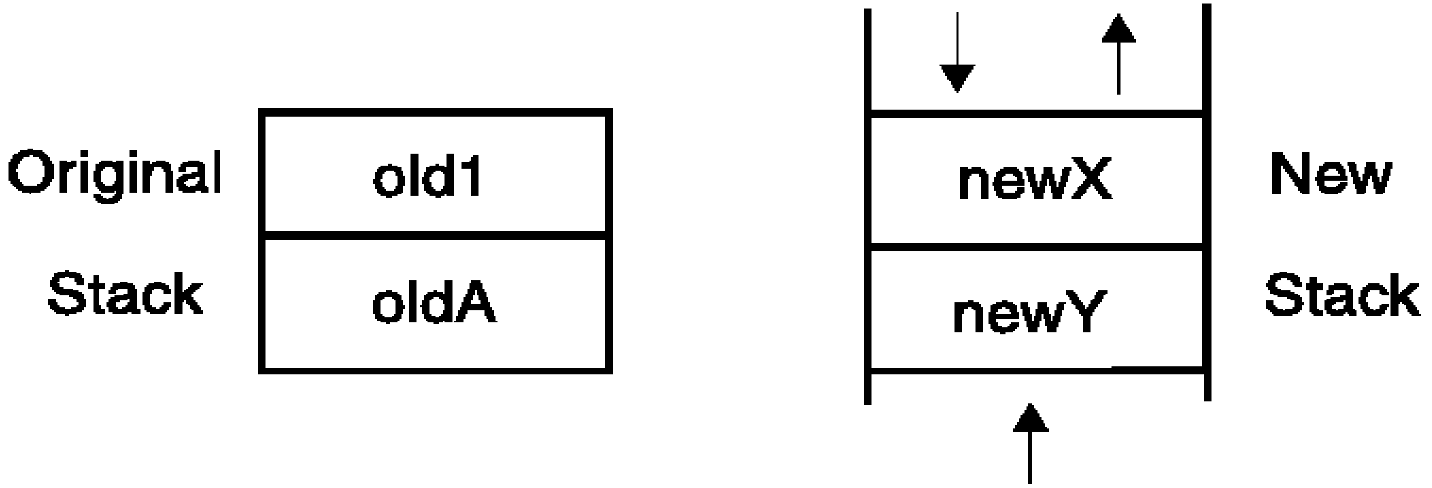
Using Buffers in the Data Stack . . .

- Important notes
  - When an element is removed from an empty buffer, the buffer disappears with no error or indication
  - Keep track of where you are in buffers within the stack
  - Used Queued() to keep track of the total number of elements in the stack
  - To remove a buffer
    - Issue Dropbuf (the recommended approach)

      or
    - Remove an element (via Parse Pull) when the buffer is already empty
  - The next request to remove an element will move
    - To the next buffer if there is one (including buffer 0)
    - To the external input queue if the stack (all buffers) are empty

Protecting Elements in the Data Stack – z/OS Only

- REXX code can use the stack, but protect itself from inadvertently removing someone else's data stack elements
  - Create a **new private data stack** using the **NEWSTACK** command
- All elements added after a NEWSTACK command are placed in the new data stack
  - Elements on the original data stack cannot be accessed by an EXEC or any called routines until the new stack is removed (not just emptied)
  - When there are no more elements in the new data stack, information is taken from the terminal (not the original data stack)



Protecting Elements in the Data Stack - z/OS Only

- DELSTACK - removes a data stack
  - Removes the most recently created data stack
    - Including all remaining elements in the stack
  - Caution
    - If no stack previously created with NEWSTACK, then DELSTACK removes all the elements from the **original stack**
- QSTACK - returns the number of data stacks
  - Including the original stack
  - Puts the value in the variable RC
- Note: For z/OS, the QUEUED() function returns the number of elements in the current data stack.

Data Stack vs Buffers

- Data Stack
  - Advantages
    - Protects data in the original stack
      - Never defaults back to the “previous” stack in the chain
      - Must specifically delete current stack to move to previous stack
      - Can easily request terminal input if also have items in the stack
        - Just create a new stack with nothing on it and issue “Pull”
  - Disadvantages
    - Only available on z/OS
      - z/VM must issue “Parse External” to request terminal input if data is in the stack

Data Stack vs Buffers

- Buffers
  - Advantages
    - Create a stack on top of the existing stack for new list of items
    - Use “QElem” (z/OS only) to keep track of how many items in this buffer
  - Disadvantages
    - No guaranteed protection of previous stack in the chain
      - If current stack is empty, will proceed to next one automatically

Compound Variables vs Data Stack

- Compound Variables
  - Advantages
    - Basically variables - REXX will manage them like other variables
    - Only one step required to assign a value
    - Provide opportunities for clever and imaginative processing
  - Disadvantages
    - Can not be used to pass data between external routines

- Conclusion
  - Try to use compound variables whenever appropriate
    - They are simpler

Compound Variables vs Data Stack

- Data Stack
  - Advantages
    - Can be used to pass data to external routines
    - Able to specify commands to be run when the EXEC ends
    - Can provide response(s) to an interactive command that runs when the EXEC ends
  - Disadvantages
    - Program logic required for stack management
    - Processing needs 2 steps
      - Take data from input source and store in stack
      - Read from stack into variables
    - Stack attributes and commands are OS dependent

I/O and Troubleshooting

EXECIO Command – z/OS

- A TSO/E REXX command that provides record-based processing
  - Used to read and write records from/to a z/OS sequential data set or z/OS partitioned data set member
  - Requires a DDNAME to be specified
    - Use ALLOC command to allocate data set or member to a DD
- Records can be read into or written from compound variables or the data stack
- Can also be used to:
  - Open a data set without reading or writing any records
  - Empty a data set
  - Copy records from one data set to another
  - Add records to the end of a sequential data set
  - Update data in a data set, one record at a time

EXECIO Command – z/VM

- CMS EXECIO command provides record-based processing
- Recommend using CMS Pipelines (PIPE command) instead
  - Simpler to use

```
‘EXECIO * DISKR EMPLOYEE INFO A (STEM REC. FINIS’
```

vs

```
‘PIPE < EMPLOYEE INFO A    | STEM rec.’
```

- PIPEs has much more function

Special Variables

- RC variable
  - Return code from external commands and special REXX commands/statements
- RESULT variable
  - Value of an expression returned by a subroutine

Troubleshooting – Condition Trapping

- Signal On and Call On instructions can be used to trap exception conditions
- Syntax:

```
►►───SIGNAL ON ┬ERROR───┬ NAME labelname───►◄
               ├FAILURE┤
               ├HALT───┤
               ├NOVALUE┤
               └SYNTAX─┘

►►───CALL ON ┬ERROR───┬ NAME trapname───►◄
             ├FAILURE┤
             └HALT───┘
```

- **Condition types:**

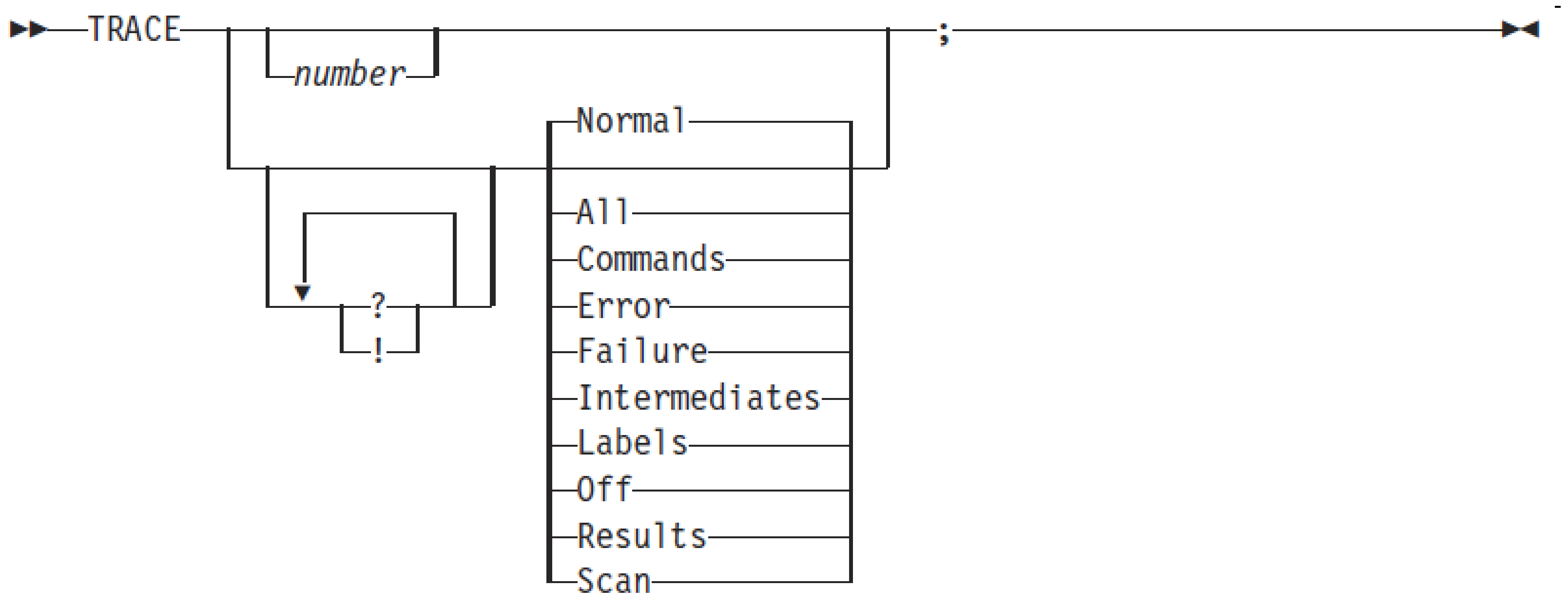
– ERROR	- error upon return (positive return code)
– FAILURE	- failure upon return (negative return code)
– HALT	- an external attempt was made to interrupt and end execution
– NOVALUE	- attempt was made to use an uninitialized variable
– SYNTAX	- language processing error found during execution
– NOTREADY	- z/VM only. Error during input or output operation

Troubleshooting – Condition Trapping. . .

- Good practice to enable condition handling to process unexpected errors
  - Specifically `Signal On NoValue`
- Use REXX provided functions and variables to identify and report on exceptions
  - CONDITION function – returns information on the current condition
    - Name and description of the current condition
    - Indication of whether the condition was trapped by SIGNAL or CALL
    - Status of the current trapped condition
  - RC variable
    - For ERROR and FAILURE - ontains the command return code
    - For SYNTAX - contains the syntax error number
  - SIGL variable – line number of the clause that caused the condition
  - ERRORTEXT function – returns REXX error message for a SYNTAX condition
    `Say ErrorText(rc)`
  - SOURCELINE function – returns a line of source from the REXX EXEC
    `Say Sourceline(sigl)`

Troubleshooting – Trace Facility

- Provides powerful debugging capabilities
  - Displays the results of expression evaluations
  - Displays the variable values
  - Follows the execution path
  - Interactively pauses execution and runs REXX statements
- Activated using the Trace instruction and function
- Syntax:



Troubleshooting – Trace Facility . . .

- Code example:

```
A = 1
B = 2
C = 3
D = 4
Trace R
If (A > B) | (C < 2 * D) Then
  Say 'At least one expression was true.'
Else
  Say 'Neither expression was true.'
```

Trace Results

- Result:

```
  7 *-* If (A > B) | (C < 2 * D)
    >>>   "1"
    *-*  Then
  8 *-*  Say 'At least one expression was true.'
    >>>    "At least one expression was true."
At least one expression was true.
```

Troubleshooting – Trace Facility . . .

- Code example:

```
A = 1
B = 2
C = 3
D = 4
Trace I
If (A > B) | (C < 2 * D) Then
  Say 'At least one expression was true.'
Else
  Say 'Neither expression was true.'
```

Trace Intermediates

- Result:

```
 6 *-* If (A > B) | (C < 2 * D)
       >V>   "1"
       >V>   "2"
       >O>   "0"
       >V>   "3"
       >L>   "2"
       >V>   "4"
       >O>   "8"
       >O>   "1"
       >O>   "1"
       *-*   Then
     7 *.*   Say 'At least one expression was true.‘
       >L>   “At least one expression was true.”
At least one expression was true.
```

Troubleshooting – Trace Facility . . .

- Interactive trace provides additional debugging power
  - Pause execution at specified points
  - Insert instructions
  - Re-execute the previous instruction
  - Continue to the next traced instruction
  - Change or terminate interactive tracing
- Starting interactive trace
  - ? option with the TRACE instruction
  - In TSO, use EXECUTIL TS command (Trace Start)
    - Code in your REXX EXEC
    - Issue from the command line to debug next REXX EXEC run
    - Cause an attention interrupt and enter TS

Programming Style and Techniques

- Be consistent with your style
  - Helps others read and maintain your code
  - Having style rules will make the job of coding easier
- Indentation
  - Improves readability
  - Helps identify unbalanced or incomplete structures
    - Do - End pairs
- Comments
  - Provide them!
  - Choices:
    - In blocks
    - To the right of the code

Programming Style and Techniques . . .

- Capitalization
  - Can improve readability
  - Suggestions
    - Use all lowercase for variables
    - Use mixed case (capitalize the first letter) for keywords, labels, calls to internal subroutines
    - Use upper case for calls to external routines (commands)
- Variable names
  - Try to use meaningful names
    - Helps understanding and readability
  - Avoid 1 character names
    - Easy to type but difficult to manage and understand
    - Exception – indices to compound variables
  - Avoid ending names with letter O or lowercase L
    - Hard to distinguish between numbers 0 and 1

Programming Style and Techniques . . .

- Subroutines
  - Subroutines are useful to break code into 'functional units' of not more than one page.
    - Eases learning and debugging since the programmer can concentrate on a close-knit piece of code that only does one thing; No observable performance impact
- Comparisons
  - REXX supports exact (e.g. "==") and inexact (e.g. "=") operators
  - Only use exact operators when appropriate

```
if action == "SAVE" then …
```

Extra blank

  - Above comparison will fail if variable action is "SAVE "
  - Avoid using non-standard NOT characters: "¬" and "/"
    - Portability problem when transferring code to an ASCII platform
    - Use "\=", or less commonly used "\>" "\<=

Programming Style and Techniques . . .

- **Semicolons**
  - Can be used to combine multiple statements in one line
    - DON'T – detracts from readability
  - Languages like C and PL/I require a “;” to terminate a line
    - Can also be done in REXX
    - DON'T – doubles internal logic statement count for interpreted REXX
- **Conditions**
  - For complex statements, REXX evaluates all Boolean expressions, even if first fails:

```
If 1 = 2 & 3 = 4 & 5 = 5 Then Say 'Impossible‘
```

    - Divide-by-zero can still occur if a=0

```
If a \== 0 & b/a > 1 Then ...
```

    - Can be avoided by nesting IF statements:

```
If a \== 0 Then
 If b/a > 1 Then ...
```

Programming Style and Techniques . . .

- Literals
  - Important to use literals where appropriate
    - For example: external commands
  - Lazy programming can lead to unfortunate results
    - For uninitialized variables: value=name
      `control errors cancel`
    - This usually works
      - Breaks if any of the 3 words is a variable with value already assigned
    - Also a performance cost for unnecessary variable lookups (20%+ more CPU)
    - Instead enclose literals in quotation marks
      `‘CONTROL ERRORS CANCEL’`

Programming Style and Techniques . . .

- External commands
  - Best practices
    - Enclose in quotation marks
    - Use uppercase
    - Fully spell out the command
      - Don’t assume any abbreviations that may not be present if the EXEC is moved to another system
      - Preface with the external environment as needed

REXX Enhancements in z/OS V2.1

REXX Enhancements in z/OS V2.1 and later

- EXECIO enhanced to support I/O with RECFM=U, VS, VBS
- RECFM=U,VS,VBS support also added to callable I/O interface
- New TRAPMSG function allows IRX... messages, if issued from a command invoked by the EXEC, to be captured via OUTTRAP
- STORAGE function now supports 64-bit addresses for both reading from and writing to storage.
- Empty sequential data set can be part of a concatenation accessed by EXECIO,CLIST I/O, PRINTDS if it is SMS managed
- LISTDSI enhanced (REXX and CLIST)
- RACF/NORACF operand
- Multi Volume Support
- Handles data sets with extended attributes
- APAR OA48348 - MVSVAR function allows symbol names up to 16 chars
- Other smaller requirements

- **z/OS v2.2 enhancement allows for “long symbols” to be used**
  - APAR release to permit use of long system symbol names in REXX and CLIST
- **Z/OS v2.3 comment:**
  - While significant zOS work for supporting 8-character USERID, not a specific REXX item

Long symbols example

```
|  Assume the following symbols have been defined with SYMDEF
|  statements in the active IEASYMxx member of 'SYS1.PARMLIB'.
|   LONG_SYMBOL_NAME having  value: "SY1T_ON_PLEX_A44T"
|   EXTSYM_ having value:
|              "<==THIS VALUE CAN BE UP TO 44 CHARS LONG===>"
|
|  That is, IEASYMxx contains the definitions:
|SYMDEF(&LONG_SYMBOL_NAME='SY1T_ON_PLEX_A44T')
|SYMDEF(&EXTSYM_='<==THIS VALUE CAN BE UP TO 44 CHARS LONG===>')
|
|  Then the MVSVAR function can be used to retrieve the values
|  of these symbols as shown:
|   z1 = MVSVAR('SYMDEF','LONG_SYMBOL_NAME')
|   z2 = MVSVAR('SYMDEF','EXTSYM_')
|   say z1
|    <- Returns z1: SY1T_ON_PLEX_A44T
|   say z2
|    <- Returns z2: <==THIS VALUE CAN BE UP TO 44 CHARS LONG===>
```

Overview - EXECIO

- Over the years many customers have asked for the capability to handle I/O to data sets containing records with Variable Spanned (VS, VBS) RECFM, and with data sets having undefined (U) RECFM. This includes the ability to handle spanned files generated by SMF, or to read load library type undefined files.

- Problem Statement / Need Addressed
  - Provide the capability to read or write RECFM=VS, VBS, U type data sets under REXX.

  Note: RECFM=VS/VBS files do not support update mode (DISKRU).

- Solution
  - EXECIO support extended

- Benefit / Value
  - The power of REXX and EXECIO can be used to process data sets with RECFM attributes that were formerly not supported.

Usage & Invocation

- Example 1 continued

```
 ELSE
  do
    say 'File allocation error ...'
    error = 1                                /* Error occurred         */
  end
IF error = 0 then                        /* If no d is ok              */
  DO
    "execio "inrec.0" DISKW OUTVBS (STEM inrec. FINIS"  /* Write all
                                               records read to the new file */
    if rc=0 then
      do
        say 'Output to new VBS file completed successfully'
        say 'Number of records copied ===> ' inrec.0
      end
    else
      do
        say 'Error writing to new VBS file '
        error = 1                            /* Error occurred         */
      end
  END
```

Usage & Invocation

- Example 2. Use EXECIO to read a member of a RECFM=U file and change the first occurrence of the word 'TSOREXX ' within each record to 'TSOEREXX' before rewriting the record. If a record is not changed, it need not be rewritten.

```
/* REXX */
/* Alloc my Load Lib data set having RECFM=U BLKSIZE=32000 LRECL=0 */
"ALLOC FI(INOUTDD) DA('apar2.my.load(mymem)') SHR REUSE"
readcnt = 0                         /* Initialize rec read cntr    */
updtcnt = 0                         /* Initialize rec update cntr  */
error = 0                           /* Initialize flag             */
EoF = 0                             /* Initialize flag             */
do while (EoF=0 & error=0)          /* Loop while more recs/no err */
  "execio 1 DISKRU INOUTDD (STEM inrec." /* Read a rec for update  */
  if rc = 0 then                    /* If read ok                  */
    do  /* Replace 1st occurrence of 'TSOREXX' in record by 'TSOEREXX'
                                   and write it back               */
      readcnt = readcnt + 1       /* Records read                */
      z = POS('TSOREXX ',inrec.1,1)    /* Find target within rec   */
      if z /= 0 then               /* If found, replace it         */
         do
           inrec.1 = SUBSTR(inrec.1,1,z-1)||'TSOEREXX'|| ,
                  SUBSTR(inrec.1,z+LENGTH('TSOEREXX'))  /*Replace it*/
           "execio 1 DISKW INOUTDD (STEM inrec." /* Rewrite the update
                                            made to the last record read*/
```

Usage & Invocation

- Example 2 continued

```
        if rc > 0 then                  /* If error                     */
                error=1                   /* Indicate error                 */
              else
                updtcnt = updtcnt + 1     /* Incr update count              */
            end
          else                            /* Else nothing changed, nothing
                                             to rewrite                     */
            NOP                           /* Continue with next record      */
        end
      else                                /* Else non-0 RC                  */
        if rc=2 then                      /* if end-of-file                 */
          EoF=1                           /* Indicate end-of-file           */
        else
          error=1                         /* Else read error                */
    end                                   /* End do while                   */
    "execio 0 DISKW INOUTDD (FINIS"       /* Close the file                 */
    if error = 1 then
      say '*** Error occurred while updating file '
    else
      say updtcnt' of 'readcnt' records were changed'
    "FREE FI(INOUTDD)"
    exit 0
```

Overview – TRAPMSG function

- TRAPMSG – a new TSO/E REXX function used in conjunction with OUTTRAP to permit REXX to trap REXX messages (i.e. IRX..... msgs) in some instances. Prior to this, no IRX.... msg could be trapped.

- Problem Statement / Need Addressed
  - REXX IRX..... messages should be trappable via OUTTRAP just as other output (e.g. such as say output from nested execs) is trappable.

- Solution
  - Use TRAPMSG('on') to tell REXX to treat REXX msg output in the same was as any other output, for purposes of trapping.

- Benefit / Value
  - REXX msgs issued by nested execs, and by host commands invoked by REXX (e.g. execio) can now be trapped into an OUTTRAP variable, rather than always being written to screen.
  - CLIST error msgs from CLISTs invoked by REXX also now trappable.

Usage & Invocation

- TRAPMSG() - returns current setting. /* OFF perhaps   */
- TRAPMSG('ON' | 'OFF') - enables or disables output trapping for IRX.... msgs. Default is 'OFF'

Usage & Invocation

- Example 1: A REXX exec invokes execio without allocating the indd file. EXECIO will return RC=20 and an error message. By trapping the message with OUTTRAP, the exec can decide what to do with the error. (This same technique can be used to trap the IRX0250E message if execio were to take an abend, like a space B37 abend.)

```
==============================================
msgtrapstat = TRAPMSG('ON')     /* Save current status and set
                   TRAPMSG ON to allow REXX msgs to be trapped  */
outtrap_stat = OUTTRAP('line.')   /* Enable outtrap              */
/***************************************************************/
/* Invoke TSO host cmd, execio, and trap any error msgs issued  */
/***************************************************************/
"execio 1 diskr indd (stem rec. finis"

if RC = 20 then             /* If execio error  occurred     */
  do i=1 to line.0
    say '==> ' line.i       /* Write any error msgs          */
  end
outtrap_stat = OUTTRAP('OFF')   /* Disable outtrap             */
msgtrapstat = TRAPMSG('OFF')    /* Turn it off                 */
exit 0
```

Usage & Invocation

- Example 2: A REXX exec turns on OUTTRAP and TRAPMSG and invokes a second REXX exec. The second REXX exec gets an IRX0040I message due to   an invalid function call. Exec1 is able to trap the message issued from exec2.

Note that if exec1 had made the bad function call, it could not trap the error message because a function message is considered at the same level as the exec. This is similar to the fact that an exec can use OUTTRAP to trap SAY statements from an exec that it invokes, but it cannot trap its own SAY output.

```
 =============================================
     /* REXX - exec1 */
    trapit = OUTTRAP('line.')
    trapmsg_stat = TRAPMSG('ON')
    call exec2
    do i=1 to line.0   /* Display any output trapped from exec2 */
      say '==> ' line.
    end
    trapit = OUTTRAP('OFF')
    trapmsg_stat = TRAPMSG('OFF')
    exit 0

    /* REXX - exec2 */
    say 'In exec2 ...'
    time = TIME('P')    /* Invalid time operand, get msg IRX0040I*/
    return time
```

Overview – STORAGE function

- z/OS can use address 64-bit storage, providing vastly expanded addressable areas. REXX cannot read or write to these areas.

- Problem Statement / Need Addressed
  - REXX STORAGE needs ability to view or change storage within 64-bit addressable areas above the BAR.

- Solution
  - STORAGE extended to handle 64-bit addresses, in addition traditional 24-bit and 31-bit addresses.

- Benefit / Value
  - Clever programmers can make use of 64-bit storage to greatly expand the amount of data than can be maintained, in storage, by REXX.

Usage & Invocation

- STORAGE function now supports 64-bit address represented by 9-17 hexidecimal chars, consisting of 8-16 hex chars and an optional underscore (“_”) separating high and low order half

- Retrieve 25-bytes from addr 000AAE35:
  storet = STORAGE(000AAE35,25)

- Replace data at 0035D41F with 'TSO/E REXX'
  storet = STORAGE(0035D41F,,'TSO/E REXX')

- The following illustrate valid 64-bit addresses that can be used with storage
  storet = STORAGE(00000001EF_80000010,60) – read 60-bytes from
  64-bit address 1EF_80000010

Usage & Invocation

```
----------------------------------------------------------------------------
    The following illustrates some valid and invalid 64-bit addresses:
    -------------------------------------------------------------------
     Hex Address passed    Binary Address
     to STORAGE            used by STORAGE        Comment
     ================      ==================     ===================
            _00000010      '0000000000000010'x    - Valid 64-bit addr.
                                                    (Padded to left with
                                                    0's to 64-bits.)
                                                    Addresses same area as
                                                    31-bit '00000010'x addr.
           0_00000010      '0000000000000010'x    - Valid 64-bit addr.
                                                    Addresses same area
                                                    as _00000010.
           0_80000010      '0000000080000010'x    - Valid 64-bit addr.
                                                    Addr is 2GB beyond
                                                    the 0_00000010 addr.
           000001EF10      '000000000001EF10'x    - Valid 64-bit addr.
         1EF_80000010      '000001EF80000010'x    - Valid 64-bit addr.
          1EF80000010      '000001EF80000010'x    - Valid 64-bit addr
                                                    without "_" separator.
    000001EF_80000010      '000001EF80000000'x    - Valid 64-bit addr.
    000001EF_10             Invalid Addr          - Right half of 64-bit
                                                    addr <8 chars.
```

Usage & Invocation

```
  Hex Address passed   Binary Address
  to STORAGE           used by STORAGE       Comment
  ================     ==================    ===================

 00000001EF_000010      Invalid Addr         - Left half of addr >8
                                               chars, right half <8
                                               chars.
 0000001EF_80000010     Invalid Addr         - More than 16 hex chars
                                               Also, left half more
                                               than 8 chars.
 00001EF8000001000      Invalid Addr         - More than 16 hex chars
```

```
 As an example of what you might expect, consider STORAGE used to
retrieve 25 bytes from a 64-bit addressable area:
say  '<'C2X(STORAGE(1EF_80000010,25))'>'
                       /* Returns  ...
                      <IARST64 COMM SIZE 000512 >  perhaps */
```

Overview – LISTDSI enhanced

- Keep LISTDSI REXX function/ CLIST statement current with new features added to z/OS, and improve current capabilities.

- Problem Statement / Need Addressed
  - As new features are introduces to data sets, LISTDSI should be improved to report on those. Also LISTDSI should be able to handle multi-volume data sets.

- Solution
  - New variables have been added to LISTDSI.
  - LISTDSI now provides information on all volumes of a multi-volume data set, not just the first.
  - RACF/NORACF operand added.

- Benefit / Value
  - New capabilities help keep LISTDSI current.

Usage & Invocation

- LISTDSI 'dsname'... RACF/NORACF MULTIVOL/NOMULTIVOL
  - Specifying NORACF means LISTDSI will not determine the RACF status. This implies that LISTDSI will not attempt to open the data set to gather additional information, even if open is necessary based on another keyword. For example, for a PDS, if DIRECTORY is specified, LISTDSI would open the data set to get directory info, but will not if NORACF is specified.
  - Specify NORACF if you do not want LISTDSI to query RACF as to whether a data set is protected. (Default is RACF.)
  - Specify MULTIVOL if you want information on the totality of all volumes of a multi-volume data set. NOMULTIVOL provides information on just the first volume (as prior to this support).

Usage & Invocation

- New LISTDSI variables set
  - SYSNUMVOLS - Number of volumes used, always returned
    - SYSVOLUMES - Volume names separated by blanks, up to number in SYSNUMVOLS. Returns 7-char per volume (6-char volume name plus 1 blank separator). Up to 412 chars (59 vols) .
    - SYSVOLUME – existing variable, returns name of first volume
  - SYSUSEDPERCENT - Percent pages used for PDSEs. Always returned for PDSEs along with previously existing SYSUSEDPAGES. One or all vols.

Usage & Invocation

- For EAV volumes:
  - SYSCREATEJOB - Jobname that created data set, if available
    e.g. PAYROLL
  - SYSCREATESTEP- Stepname that created data set, if available
    e.g. IKJEFT01
  - SYSCREATETIME- Time that data set was created, if available in format hh:mm:ss. (e.g. 02:35:15)
  - SYSCREATE - Previously existing var, returns Create Date
    (e.g. 2012/193)
- Existing variables with modified meaning
  - SYSALLOC - one or all vols. Space allocated.
  - SYSUSED – one or all vols. Space used.
  - SYSEXTENTS – one or all vols. Number of extents.
  - SYSRACFA - blank if NORACF. **'NONE'/'GENERIC'/'DISCRETE'** if RACF was specified or defaulted.

Enhancement Summary

New features of REXX now include

- Long symbols support
- Enhancements to EXECIO to support I/O to RECFM=VS,VBS, U data sets.
- New TRAPMSG function.
- Enhancements to REXX STORAGE function to support 64-bit addresses.
- Null SMS managed data sets allowed in a sequential concatenation for EXECIO, CLIST I/O, PRINTDS.
- Enhancements to LISTDSI

More Details

- SA22-7790-11, z/OS TSO/E REXX Reference
- SA22-7781-08, z/OS TSO/E CLISTs
- SA22-7786-12, z/OS TSO/E Messages

Related Programs

CMS and TSO Pipelines

- A powerful method of processing or manipulating data
- Can be called within REXX programs
- A collection of data processing elements connected in a series
  - Output of one element becomes the input to the next element
  - For example, on z/VM

    ```
    'PIPE CP QUERY DASD | STEM dasd.'
    ```

    - Issues the CP command QUERY DASD
    - Response is written into the pipeline
    - Next stage (STEM) receives the input and places it into the stem variable “dasd”, setting dasd.0 to the number of lines of data
- Included in all current releases of z/VM
- Available as a separate product for TSO
  - Batchpipes (5655-D45)

Open Object REXX

- Open Object REXX is available via open source community
  - Runs on Linux on z Systems
  - Many other platforms
- www.oorexx.org
  - Managed by REXX Language Association
- 99% compatible with other System z REXX programs
- Informal testing with SLES on memory and CPU constrained system
  - Compare PERL and OOREXX – OOREXX is much faster!
  - Memory footprint of OOREXX is similar to PERL with several modules loaded

(Open Source) NetRexx

- An object oriented Rexx for the Java Virtual Machine (JVM)
  - Write in REXX (or REXX-like)
  - Compiler converts to Java source statements and bytecode
- Available via open source community since 2011
- netrexx.org
  - Managed by REXX Language Association

Additional Information and Contacts

- ➢ IBM REXX Website
  http://www.ibm.com/software/awdtools/rexx
- ➢ IBM Contacts
  - Virgil Hein, vhein@us.ibm.com
    - Compiler and Library for REXX on zSeries

References

- Compiler and Library for REXX User's Guide and Reference (SH19-8160)
- REXX/VM User's Guide (SC24-6222)
- REXX/VM Reference (SC24-6221)
- TSO/E REXX Reference (SA32-0972)
- z/OS V2R2 TSO/E REXX Reference (SA32-0972-02) *added recently*
- z/OS TSO/E CLISTs (SA32-0978)
- z/OS TSO/E Messages (SA32-0970)
- ISPF Services Guide for z/OS (SC19-3626)
- ISPF Dialog Developer's Guide and Reference for z/OS (SC19-3619)
- ISPF Edit and Edit Macros for z/OS (SC19-3621)
- Using REXX and z/OS UNIX System Services (SA23-2283)
- SDSF Operation and Customization (SA23-2274)
- DB2 for z/OS Application Programming and SQL Guide (SC19-4051)
- z/OS MVS IPCS Commands (SA23-1382)
- z/OS MVS Authorized Assembler Services Guide (SA23-1371)
- Security Server RACF Macros and Interfaces (SA23-2288)

(Previous) References

- Compiler and Library for REXX User's Guide and Reference (SH19-8160)
- REXX/VM User's Guide (SC24-6222)
- REXX/VM Reference (SC24-6221)
- TSO/E REXX Reference (SA22-7790)
- SA22-7781-08, z/OS TSO/E CLISTs
- SA22-7786-12, z/OS TSO/E Messages
- ISPF Services Guide
  - z/OS V1, SC19-3626
  - z/OS V2, SC34-4819
- ISPF Dialog Developer's Guide and Reference
  - z/OS V1, SC19-3619
  - z/OS V2, SC34-4821
- ISPF Edit and Edit Macros
  - z/OS V1, SC19-3621
  - z/OS V2, SC28-1312
- Using REXX and z/OS UNIX System Services (SA22-7806)
- SDSF Operation and Customization (SA22-7670)
- DB2 Application Programming and SQL Guide (SC19-4051)
- MVS IPCS Commands (SA22-7594)
- MVS Programming Authorized Assembler Services Guide (SA22-7605)
- Security Server RACF Macros and Interfaces (SA22-7682)

धन्यवाद

Hindi

多謝

Traditional Chinese

감사합니다

Korean

Спасибо

Russian

Gracias

Spanish

Thank You

English

Obrigado

Brazilian Portuguese

شكراً

Arabic

Danke

German

Grazie

Italian

多谢

Simplified Chinese

Merci

French

நன்றி

Tamil

ありがとうございました

Japanese

ขอบคุณ

Thai