REXX Language Coding Techniques

SHARE 120
Session 12333

Peter Van Dyke and Virgil Hein
IBM Australia
SHARE 120, Winter 2013
pvandyke@au1.ibm.com

Important REXX Compiler Disclaimer

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

- **REXX Compiler**
- **External Environments and Interfaces**
- **Key Functions and Instructions – Power Tools**
- **REXX Data Stack Vs Compound Variables**
- **EXECIO and Stream I/O**
- **Troubleshooting**
- **Programming Style and Techniques**

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 on z/OS and z/VM

- **IBM Compiler for REXX on zSeries Release 4**
  - z/VM, z/OS: PID 5695-013
- **IBM Run Time Library for REXX on zSeries Release 4**
  - z/VM, z/OS: PID 5695-014
- **VSE part of operating system**
- **IBM Alternate Library for REXX on zSeries Release 4**
  - Included in z/OS 1.9 base operating system
    - Free download
    - http://www-01.ibm.com/software/awdtools/rexx/rexxzseries/altlibrary.html

REXX Compiler Libraries

- **A REXX library is required to execute compiled programs**
- **Compiled REXX is not an LE language**
- **2 choices: Run-time library and Alternate library**
  - Run-time library. Program product.
  - Alternate library. Free. Uses the native system's REXX interpreter.
- **Compiled and library code runs in 31-bit mode**
  - base/displacement instead of relative addressing
  - BALR and other old opcodes. Can run on old hardware.
  - No z/Architecture in plan today.
- **Compiled REXX will use whichever library (run-time or alternate) is available at execution**

External Environments

- **ADDRESS instruction is used to define the external environment to receive host commands**

```
Address TSO     - sets TSO/E as the environment to receive commands
```

- **A varied array of host command environments available 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
    - See: *TSO/E REXX Reference* - http://publibz.boulder.ibm.com/epubs/pdf/ikj4a380.pdf

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

  - ISPEXEC
    - Used to invoke ISPF services like DISPLAY and SELECT
    - Only available to REXX running in ISPF
    - See: *ISPF Services Guide* - http://publibz.boulder.ibm.com/epubs/pdf/ispzsg80.pdf

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

External Environments. . .

  - ISREDIT
    - Used to invoke ISPF edit macro commands like FIND and DELETE
    - Only available to REXX running in an ISPF edit session
    - See: *ISPF Edit and Edit Macros* - http://publibz.boulder.ibm.com/epubs/pdf/ispzem80.pdf

```
Address ISREDIT "DELETE .ZFIRST .ZLAST"
```

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

```
Address MVS "EXECIO * DISKR MYINDD (FINIS STEM MYVAR"
```

External Environments. . .

- 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
  - See: *TSO/E REXX Reference*

```
"CONSOLE ACTIVATE"
Address CONSOLE "D A"  /* Display system activity */
"CONSOLE DEACTIVATE"
```

Result:

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

External Environments. . .

- 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
  - See: *TSO/E REXX Reference*

```
"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"
```

External Environments. . .

- 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
  - See: *Using REXX and z/OS UNIX System Services* - http://publibz.boulder.ibm.com/epubs/pdf/bpxzb690.pdf

```
call syscalls 'ON'
address syscall 'readdir / root.'
do i=1 to root.0
 say root.i
end
```

```
Result:
.
..
bin
dev
etc
 .
 .
```

External Environments. . .

- 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
  - See: *SDSF Operation and Customization* - http://publibz.boulder.ibm.com/epubs/pdf/isf4cs91.pdf

```
rc=isfcalls("ON")
Address SDSF "ISFEXEC ST"
do ix = 1 to JNAME.0
  if pos("PVANDYK",JNAME.ix) = 1 then do
    say "Cancelling job ID" JOBID.ix "for PVANDYK"
    Address SDSF "ISFACT ST TOKEN('"TOKEN.ix"') PARM(NP P)"
  end
end
rc=isfcalls("OFF")
exit
```

External Environments. . .

- 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
  - See: *DB2 Application Programming and SQL Guide* - *http://publib.boulder.ibm.com/epubs/pdf/dsnapm02.pdf*

```
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

- IPCS
  - Used to invoke IPCS subcommands from REXX
  - Only available when run from in an IPCS session
  - See: *MVS IPCS Commands*

- 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
  - See: *TSO/E REXX Reference* - http://publibz.boulder.ibm.com/epubs/pdf/iea2c5a0.pdf

Other “Environments” and Interfaces

- System Rexx
  - A function package that allows REXX execs to be executed outside of conventional TSO/E and Batch environments
  - System REXX execs can be invoked using assembler macro interface AXREXX or through an operator command
  - Easy way for Web Based Servers to run commands/functions & get back pertinent details
  - Exec runs in problem state, key 8, in an APF authorized address space under the MASTER subsystem
  - 2 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

  - See: *MVS Programming Authorized Assembler Services Guide -* http://publibz.boulder.ibm.com/epubs/pdf/iea2a8a0.pdf

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')"."
```

  - See: *Security Server RACF Macros and Interfaces -* http://publibz.boulder.ibm.com/epubs/pdf/ichza3a0.pdf

Other “Environments” and Interfaces. . .

- Other ISPF Interfaces
  - Panel REXX
    - Allows REXX to be run in a panel procedure
    - *REXX statement used to invoke the REXX
    - 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 the REXX
    - 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
  - See: *ISPF Dialog Developer’s Guide and Reference -* http://publibz.boulder.ibm.com/epubs/pdf/ispzdg80.pdf

Key Instructions – ARG, PULL, and PARSE

- **ARG**
  - retrieves the argument strings provided to a program or internal routine and assigns them to variables
  - short form for PARSE UPPER ARG
- **PULL**
  - reads a string from the head of the external data queue
  - short form for PARSE UPPER PULL
- **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-'
```

PARSE Templates

- **Simple Template**
  - divides the source string into blank-delimited words and assigns them to the variables named in the template

```
string = ‘  Parse the    blank-delimited   string’
parse var string var1 var2 var3 var4 .
var1 -> ‘   Parse’
var2 -> ‘the’
var3 -> ‘blank-delimited’
var4 -> ‘string’
```

  - A period is a placeholder in a template – a “dummy” variable used to collect unwanted data

```
string = “Last one gets what's left”
parse var string var1 . var2
var1 -> “Last”
var2 -> “gets what’s left”
```

PARSE Templates. . .

- **String Pattern Template**
  - a literal or variable string pattern indicating where the source string should be split

```
string = ‘  Parse the    blank-delimited   string’

Literal:
parse var string var1 ‘-’ var2 .

Variable:
dlm = ‘-’
parse var string var1 (dlm) var2 .

Result:
var1 -> ‘   Parse the    blank’
var2 -> ‘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 with an equal sign

```
          ----+----1----+----2----+----3----+----4----+
string = ‘Van Dyke           Peter          Australia  ’
parse var string 1 surname 20 chrname 35 country 46 .
surname -> ‘Van Dyke           ’
chrname -> ‘Peter          ’
country -> ‘Australia  ’
```

  - 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----+
string = ‘Van Dyke           Peter          Australia  ’
parse var string 1 surname +19 chrname +15 country +11 .
surname -> ‘Van Dyke           ’
chrname -> ‘Peter          ’
country -> ‘Australia  ’
```

INTERPRET Instruction

- **Expression specified with the INTERPRET instruction is evaluated and then the resulting value is processed (interpreted)**
  - Adds an extra level of interpretation

```
conf = ‘SHARE’
interpret conf "= 'Orlando';say 'Location is' share"
Result:
Location is Orlando
```

  - Provides powerful test and debugging capabilities

```
parse external debug_cmd    /* Receive command from user */
interpret debug_cmd         /* Run the user’s command    */
```

STORAGE Function

- **Syntax:**

```
>>-STORAGE(address-+------------------------+-)---------------><
                   '-,-+--------+-+-------+-'
                       '-length-' '-,data-'
```

- **Returns *length* bytes of data from the specified address in storage.**
  - *address* is a character string containing the hexadecimal representation of the storage address
  - *data* is a character string that overwrites the data at *address*

```
data = storage(00FDE309,3)  /* Get 3 bytes at addr FDE309 */
```

- **A TSO/E external function but can be used in any MVS address space (TSO/E and non-TSO/E)**

- **Not all storage is available to access or update**
  - Virtual storage addresses may be fetch protected, update protected, or may not be valid
  - Null string returned

STORAGE Function. . .

- **Use the C2D and D2X functions to process addresses obtained with the STORAGE function**
  - C2D - returns the decimal value of the binary representation of a string

```
C2D('81'X)     ->    129
```

  - D2X - returns a string, in character format, that represents a decimal number converted to hexadecimal

```
D2X(249)       ->    'F9'
```

  - Example – get the Address Space Vector Table address (CVTASVT) from the Communications Vector Table (CVT)

```
cvt = STORAGE(10,4)                    /* Get CVT address     */
cvtasvt  = STORAGE(D2X(C2D(cvt)+556),4)  /* Get CVTASVT         */
```

STORAGE Function. . .

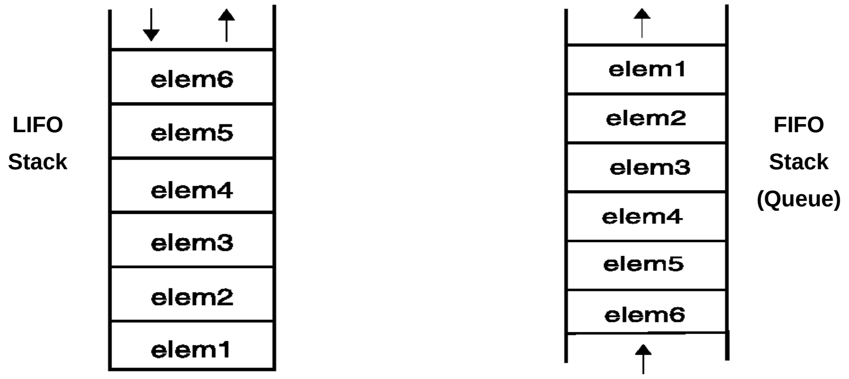
- **Use functions to simplify the job of retrieving pointers and other data**
  - PTR() - returns a 4 byte pointer as a decimal value
    - arg(1) is the decimal value of the address where the pointer is located
  - STG() - returns an EBCDIC string
    - arg(1) is the decimal value of the address where the data is located
    - arg(2) is the length of the data to be returned
  - Example – get the MVS release and FMID from the CVT prefix area

```
NUMERIC DIGITS 20              /* Set precision to 20 digits    */
cvt = PTR(16)                  /* Get CVT address               */
cvtfixa  = cvt-256             /* CVT prefix address            */
cvtprod  = STG(cvtfixa+216,16) /* MVS product level data        */
Say 'MVS release and FMID:' cvtprod
PTR: RETURN C2D(STORAGE(D2X(arg(1)),4))      /* Return pointer  */
STG:  RETURN STORAGE(D2X(Arg(1)),Arg(2))     /* Return storage  */
```

Result:
```
MVS release and FMID: SP7.1.0 HBB7750
```

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



Manipulating the Data Stack

- **3 basic REXX instructions**
  - PUSH - put one element on the top of the stack

```
elem = ‘new top element’
PUSH elem
```

  - QUEUE - put one element on the bottom of the stack

```
 elem = ‘new bottom element’
QUEUE elem
```

  - PARSE PULL - remove an element from the stack (top)

```
PARSE PULL top_elem .
```

- **1 REXX function**
  - 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 of virtually unlimited size for later use**
- **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

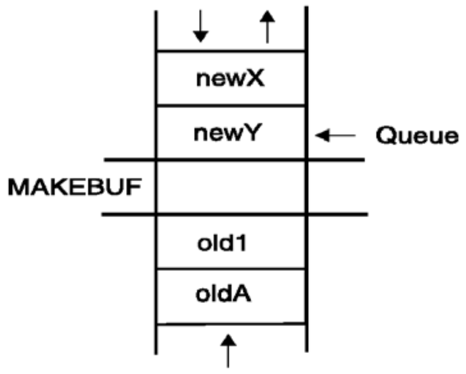
```
Queue “TSOLIB RESET QUIET”
Queue “ALLOC FI(ISPLLIB) DA(‘ISP.SISPLOAD’ 'SYS1.DFQLLIB‘) SHR REUSE”
Queue “TSOLIB ACTIVATE FILE(ISPLLIB) QUIET”
Queue “ISPF”
```

- **Pass responses to an interactive command that runs when the exec ends**

```
dest = SYSVAR('SYSNODE')"."USERID()
message = “Lunch time”
Queue “TRANSMIT”
Queue dest “LINE”
Queue message
Queue “ ”
```

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 basically changes the location the QUEUE instruction inserts new elements



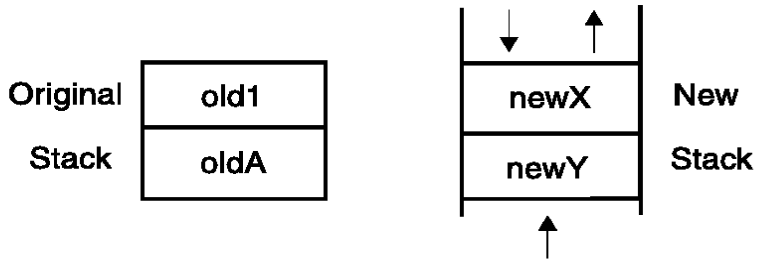
Using Buffers in the Data Stack. . .

- **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 creates an empty data stack (use with caution)
- **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**
- **CAUTION: When an element is removed below a buffer the buffer disappears.**

Protecting Elements in the Data Stack

- **An exec can protect data stack elements from being inadvertently removed by creating 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
  - When there are no more elements in the new data stack information is taken from the terminal



Protecting Elements in the Data Stack. . .

- **The DELSTACK command removes a data stack and all the remaining elements in the stack**
  - Removes the most recently created data stack

- **CAUTION: If no stack was previously created with the NEWSTACK command DELSTACK removes all the elements from the <u>original stack</u>**

- **The QSTACK command returns in the variable RC the number of data stacks (including the original stack)**

- **<u>NOTE</u>: The QUEUED() function returns the number of elements in the current data stack**

What is a Compound Variable?

- **A series of symbols (simple variable or constant) separated by periods.**
- **Made up of 2 parts – the *stem* and the *tail*.**
- **The *stem* is the first symbol <u>and</u> the first period. The symbol must be a name. Sometimes called the *stem variable*.**
- **The *tail* follows the stem and comprises one or more symbols separated by periods.**
  - **Variables take on previously assigned values**
  - **If no value assigned takes on the uppercase value of the variable name**

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

array.i                          stem:  array.
                                 tail:  i

name = 'Smith';phone = 12345;
employee.name.phone              stem:  employee.
                                 tail:  Smith.12345
```

Compound Variable Values

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

```
say month.15   → MONTH.15
month. = 'Unknown'
month.6 = 'June'
month.3 = 'March'
say month.15   → Unknown
val = 3
say month.val  → 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 in an exec**
  - Use a numeric value for the tail
  - Good practice to store the number of array entries in the compound variable with a tail of 0 (zero)
  - Often processed in a DO loop using the loop control variable as the tail

```
invitee.0 = 10
do i = 1 to invitee.0
  SAY ‘Enter the name for invitee’ i
  PARSE PULL invitee.i
end
```

- **Stems can be used with the EXECIO command to read data from and write data to a data set**

- **Stems can also be used with the OUTTRAP external function to capture output from commands**

Processing Compound Variables. . .

- **The tail for a compound variable can be used as an index to related data**

- **Given the following input data:**

```
Symbol  Atomic#   Name       Weight
H       1         Hydrogen   1.00794
HE      2         Helium     4.002602
LI      3         Lithium    6.941
. . .
```

- **The unique symbol value can be used as the tail of compound variables that hold the rest of the symbol’s values**

```
 "EXECIO * DISKR INDD (STEM rec. FINIS"
Do i = 2 To rec.0
  Parse Var rec.i symbol atomic#.symbol name.symbol weight.symbol
End i
Say "Which atomic symbol do you want to learn about?"
Parse Pull symbol
Say "The name of" symbol "is" name.symbol"."
Say "The atomic number for" symbol "is" atomic#.symbol"."
Say "The atomic weight of" symbol "is" weight.symbol"."
```

Data Stack Vs Compound Variables

- **Data Stack**
  - Advantages
    - –Can be used to pass data to external routines
    - –Able to specify commands to be run when an exec ends
    - –Can provide response 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, then read from stack into variables
    - –Stack attributes and commands are OS dependent

Data Stack Vs Compound Variables. . .

- **Compound Variables**
  - Advantages
    - –They are basically variables and REXX will manage them like other variables
    - –Only one step required to assign a value
    - –They provide opportunities for clever and imaginative processing
  - Disadvantages
    - –They cannot be used to pass data between external variables
- **Conclusion**
  - Try to use compound variables whenever appropriate. They are **<u>simpler</u>**.

EXECIO Command

- **Used to read and write records from and to a sequential data set or 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 for the following functions:**
  - 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 is a TSO/E REXX command that provides record-based processing**

REXX Stream I/O

- **Function package available as a free download from IBM**
  - ftp://ftp.software.ibm.com/s390/zos/tools/rexx/
  - Look for REXXFUNC files
- **Also shipped with the IBM Library for Rexx on zSeries (5695-014)**
- **Allows REXX execs to use stream I/O functions to process sequential data sets and partitioned data set members**
- **Why user stream I/O?**
  - Extends and enhances I/O capabilities of REXX for TSO/E
  - shields the complexity of z/OS data set I/O (to some degree)
  - A familiar I/O concept
  - Provides better portability of REXX between OS platforms

Troubleshooting – Condition Trapping

- **The CALL ON and SIGNAL 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

Troubleshooting – Condition Trapping. . .

- **Good practice to enable condition handling to process unexpected errors**

- **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 – Return Code
    - Contains the command return code for ERROR and FAILURE
    - Contains the syntax error number for SYNTAX

  - SIGL variable – line number of the clause that caused the condition

  - ERRORTEXT function – returns REXX error message for a SYNTAX condition
    ```
    say ERRORTEXT(rc)
    ```

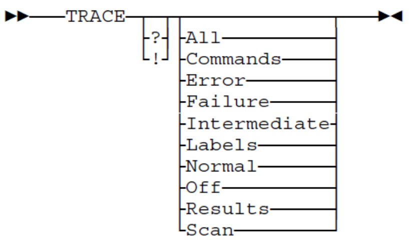
  - SOURCELINE function – returns a line of source from the REXX exec
    ```
    say SOURCELINE(sigl)
    ```

Troubleshooting – The Trace Facility

- **The Trace Facility provides powerful debugging capabilities**
  - Display the results of expression evaluations
  - Display the variable values
  - Follow the execution path
  - Interactively pause execution and run REXX statements

- **Activated using the TRACE instruction and function**

- **Syntax:**



Troubleshooting – The Trace Facility. . .

- **Trace 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.'
```

- **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 – The 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
  - EXECUTIL TS command
    - 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 to 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 groups)
- **Comments**
  - Provide them!
  - Choices:
    - In blocks
    - To the right of the code
- **Capitalization**
  - Can improve readability
  - Suggestion – use all lowercase except for labels and calls to internal subroutines

Programming Style and Techniques. . .

- **Variable names**
  - Try to use meaningful names – helps understanding and readability
  - Avoid 1 character names – easy to type but difficult to manage and understand
- **Subroutines**
  - Try to avoid the over use of subroutines or functions
  - Subroutines are useful, but have performance impact
  - If it's only called once, does it need to be a subroutine?
- **Comparisons**
  - REXX supports *exact* (e.g. "==") and *inexact* (e.g. "=") operators
  - Only use *exact* operators when appropriate

```
arg a
if a == "SAVE" then …
```

  - Above comparison will fail if argument received is "SAVE "
  - Avoid using the NOT ("¬") character
    - Portability problem when transferring code to an ASCII platform
    - Use "<>", "/=", or "\="

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 = 6 then say 'Impossible'
```

  - Nesting of IF statements sometimes required

```
if a \== 0 & b/a > 1 then ...
   –Divide-by-zero can still occur if a=0
```

  - 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 such as for external commands
  - Lazy programming can lead to unfortunate results
    - For uninitialized variables: value=name

      ```
      control errors cancel
      ```
    - This usually works, but breaks if any of the 3 words is a variable that is already assigned a value
    - Also a performance cost for unnecessary variable lookups (20%+ more CPU)

Additional Information and Contacts

- **REXX Compiler User's Guide and Reference**
  http://publibfi.boulder.ibm.com/epubs/pdf/h1981605.pdf

- **IBM REXX Web Site**
  http://www-01.ibm.com/software/awdtools/rexx

- **IBM Contacts**

Virgil Hein, vhein@us.ibm.com	(REXX at IBM)
George Kochanowski, jjkoch@us.ibm.com	(REXX Compiler)