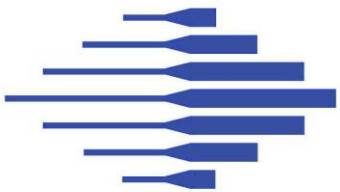


VELOCITY
SOFTWARE

Introduction to REXX

James Vincent / Rich Smrcina
Velocity Software, Inc
VM Workshop – June 2026

References

What REXX is

REXX programming, syntax, styles…

Reference Materials

REXX/VM User's Guide

- Phenomenal resource for learning REXX

REXX/VM Reference

- The paper version of HELP REXX MENU

IBM z/VM Education

- https://www.ibm.com/support/pages/zvm/education/roadmaps/
  Rexx Coding Techniques – Part 1 & 2

REXX Language Association

- https://rexxla.org

Open Object REXX

- http://www.oorexx.org

Regina REXX

- https://regina-rexx.sourceforge.io

What is REXX?

REXX (REstructured eXtended eXecutor)

Interpreted Language

- There is a compiler, but it is rarely used

"Typeless"

- Data is not a particular type such as binary, packed-decimal, etc.

Runs on many platforms

- z/VM, z/OS, VSEn, Linux, Windows

REXXTRY EXEC

REXX EXEC

```
/* try rexx
command */
Numeric Digits 25
Parse Arg command
Interpret command
```

From “A brief history of REXX”

The first specification for the language is dated 29 March 1979. This was written before any implementation was even designed, and it was circulated to a number of people for comment: this began the tradition of documentation before implementation that characterized the development of Rexx. This first specification included three sample programs written in Rexx to show how the language would look; those programs would seem familiar to today's Rexx programmers, although some details have changed.

- https://www.rexxla.org/rexxlang/history/mfc/rexxhist.html

Is it REXX or Rexx?

Writing a REXX program

REXX programs must have a filetype of EXEC in CMS

```
XEDIT MYFILE EXEC A
```

REXX programs must ALWAYS begin with a comment

```
/* this is a comment */
```

```
 MYFILE   EXEC      A1   V 255  Trunc=255 Size=2 Line=0 Col=1 Alt=7
====>
      T...+....1....+....2....+....3....+....4....+....5....+....6..
00000 * * * Top of File * * *
00001 /* this is the comment for my rexx program */
00002
00003 * * * End of File * * *
```

VELOCITY SOFTWARE

Writing a REXX program

Syntax

- REXX clauses are typically delimited by a new line

```
a = "cat"
b = "dog"
```

- They can also be delimited by a semicolon

```
a = "cat"; b = "dog";
```

Clauses can be:

Assignments	a = "Fish"
Labels	LABEL1:
Host Command	"QUERY DISK"
Null line	
REXX Instructions	Signal On ERROR

VELOCITY SOFTWARE

Style

Discussions of style fall under coding standards and can vary

- Variables lowercase
- Labels uppercase
- Host commands uppercase enclosed in double quotes
- REXX clauses/keywords should have the first character capitalized

Examples

```
company = "Acme, Inc"

LABEL1:

"CP QUERY TIME"

Call LABEL1
```

Style - Indents

- Usually two characters (spaces)
- 'Do/End' and 'Select/End' have the same indentation
- Anything in the 'Do block' or 'Select block' should be indented
- Indent clauses for 'If', 'Else', 'When', 'Otherwise'

Examples

```
      |...+....1....+....2....+....3....+....4....+....5..
00000 * * * Top of File * * *
00001 /* Simple Style */
00002   Parse Arg fname lname
00003   If fname <> '' Then
00004     Say 'Hello' fname '!'
00005   Else
00006     Do
00007       Say 'I expected to see your name'
00008       Say 'Try: MYEXEC firstname lastname'
00009     End
00010 Exit
00011 * * * End of File * * *
```

Variables

- Unassigned variables have a value of their name translated to uppercase

  name → NAME

- An initialized variable has a value of what it was assigned to

  name = “Andrew” → Andrew

- It is always good practice to initialize variables to avoid errors or confusion later

  name = “”

Variables – Examples and a Quiz

```
      |...+....1....+....2....+.
00000 * * * Top of File * * *
00001 /* rexx comment */
00002
00003 name1 = "Sam"
00004 name2 = ""
00005 name3 = sam
00006
00007 Say "Name1 is:" name1
00008 Say "Name2 is:" name2
00009 Say "Name3 is:" name3
00010 Say "Name4 is:" name4
00011
00012 * * * End of File * * *
```

```
Name1 is: Sam
Name2 is:
Name3 is: SAM
Name4 is: NAME4
```

VELOCITY SOFTWARE

Quotes

- Quotes are your friend
- Use either single (‘) or double quotes (“)
- Strings written without quotes are translated to upper case
  - Why? Or worse, the string will not be what you intended

```
Say “That’s his cat”   → That’s his cat

Say That’s his cat     → THAT’S HIS CAT
```

Strings

- Multiple blanks between strings/variables are replaced with one blank

```
name = "Sam"

Say "Hello"      name
Say "Hello"name
Say "Hello      " name
```

```
Hello Sam
HelloSam
Hello      Sam
```

Strings

- Use double vertical bars to concatenate values

```
say "C"||"A"||"T" → CAT
```

- Concatenated hex values must ALWAYS have a space after unless concatenation (||) is used

```
say "c3"x          → C
say "c3"x"AT"      → c3XAT
say "c3"x||"AT"    → CAT
```

Compound variables

- Also known as stems or arrays

```
day.1 = 'Monday'
day.2 = 'Tuesday'
```

- Variables are substituted right to left

```
num = 1
Say day.num   → 'Monday'
```

- Initialize the entire compound variable

```
day. = ''   /* sets all potential values to null */
```

Compound variables

- Stem references need to be numbers or variables
  - day.today = “Monday” /* today var can be set, or not */
  - day.1 = “Monday”

- A reference to a function will fail

```
day.datenum() = “Monday”
```

Running that code would likely produce:

```
DMSOPN062E Invalid character . in fileid RXDAY.DA MODULE
  6 +++ day.datenum() = “Monday“
DMSREX475E Error 40 running TRY EXEC, line 6: Incorrect call to
routine
```

1. A REXX program must always start with what?

2. What is the value of an unassigned variable?

REXX Instructions

- Say
  - Write the subsequent values/expressions to a single line

```
Say "Hello there" yourname

Say "Multiply 3 by 5=" 3*5
```

Arg

- Gets values from the command line that are passed to the EXEC

```
Arg var1 var2 var3
```

- Command line arguments are translated to upper case
- Each word in the command line is assigned to a symbol/variable
  - Too many args, the last symbol contains the remainder of the parameter list
  - Too few args, the unused symbols are set to null
- Use Parse Arg to respect mixed case values

Arg - Examples

```
/* */
arg var1 var2 var3

say 'var1=' var1
say 'var2=' var2
say 'var3=' var3
```

```
examp3 mine yours
var1= MINE
var2= YOURS
var3=
```

```
examp3 mine yours ours theirs
var1= MINE
var2= YOURS
var3= OURS THEIRS
```

```
/* */
Parse Arg var1 var2 var3

say 'var1=' var1
say 'var2=' var2
say 'var3=' var3
```

```
examp3 mine yours
var1= mine
var2= yours
var3=
```

VELOCITY SOFTWARE

Pull

- Read values from the console or program stack

```
Pull var1 var2 var3
```

- Works like Arg; each word entered from the console or from the program stack is assigned to a symbol/variable
- Use Parse Pull to respect mixed case values

```
/* */
Pull var1 var2 var3

say 'var1=' var1
say 'var2=' var2
say 'var3=' var3
```

```
examp4
one two three four
var1= ONE
var2= TWO
var3= THREE FOUR
```

Quiz!

1. Where does Arg get values from?
2. Where does Pull get values from?
3. What keyword is used to preserve the value case for Arg/Pull?

If/Then/Else

```
If expression Then instruction1
Else instruction2
```

- Evaluates an expression (the condition)
- If true executes instruction1
- If false, executes instruction2

```
/* */
num1 = 10
num2 = 20

If num1 = num2 Then
  say 'Equal'
Else
  say 'Not equal'
```

- Instructions can be
  - nop — No operation
  - Do/End — A block of clauses/instructions
  - If/Then/Else — Another If structure

Arithmetic operators

Addition	+	7 + 3 = 10
Subtraction	-	10 – 6 = 4
Multiplication	*	8 * 4 = 32
Division	/	14 / 2 = 7
Power	**	6 ** 2 = 36
Integer Divide	%	10 % 3 = 3
Remainder Divide	//	10 // 3 = 1

Comparison operators

Equal	=
Strictly equal	==
Not equal	<> != ^= \= ¬=
Greater than	>
Less than	<
Logical And	&
Logical Or	|

Do Loops

```
Do condition
  instructions
End
```

- Evaluates condition
- If true, executes instructions
- If false, skips the loop

- Additional Do features
  - Do Forever — Loop until told to terminate
  - Do … Until *expr* — Evaluate expression at **end** of loop
  - Do … While *expr* — Evaluate expression at **top** of loop

Do examples

```
/* */
Do i = 1 to 10
  say i 'Cat'
End
```

```
1 Cat
2 Cat
3 Cat
4 Cat
5 Cat
6 Cat
7 Cat
8 Cat
9 Cat
10 Cat
```

```
Do Forever
  say 'Cat'
End
```

```
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
Cat
```

```
/* */
Do i = 1 to 10 by 2
  say 'Cat'
End
```

```
Cat
Cat
Cat
Cat
Cat
```

Do Loops

- Use the LEAVE command to get out of a loop
- Use the ITERATE command to go to the top of a loop

```
/* */
i = 0
Do Forever
  i = i + 1
  say i 'Cat'
  If i > 10 Then
    Leave
End
```

```
1 Cat
2 Cat
3 Cat
4 Cat
5 Cat
6 Cat
7 Cat
8 Cat
9 Cat
10 Cat
11 Cat
```

```
/* */
Do i = 1 to 10
  If i = 4  Then
    Iterate
  say i 'Cat'
End
```

```
1 Cat
2 Cat
3 Cat
5 Cat
6 Cat
7 Cat
8 Cat
9 Cat
10 Cat
```

Do it again

- A Do...Forever loop needs LEAVE or EXIT to terminate
- A FOREVER loop can increment a counter with no end

```
Do i = 1
  say 'This will not stop' i
End
```

- A repetitive loop and conditional loop can be combined

```
Do i = 1 to 10 Until name <> "James"
  ...
End
```

Functions and Subroutines

- Both are code that is used to perform repetitive tasks
  - Avoid repeating the same logic in your code
- Can be internal or external
- Zero or more parameters can be passed
- A function must return a value
- A subroutine can return a value, but is not required to

Functions and Subroutines - Examples

```
/* */
arg num1 num2
sum = addthem(num1 num2)
say 'Sum=' sum
Exit

addthem: procedure
arg val1 val2
res = val1 + val2
return res
```

```
/* */
arg num1 num2
Call addthem num1 num2
say 'Sum=' result
Exit

addthem: procedure
arg val1 val2
sum = val1 + val2
return sum
```

```
/* */
arg num1 num2
Call addthem num1 num2
say 'Sum=' sum
Exit

addthem:
arg val1 val2
sum = val1 + val2
return
```

```
examp7 2 2
Sum= 4
```

Functions and Subroutines – Variable references

- All variables from the caller are available to a function or subroutine by default
  - Conversely, any variable changed in the function or subroutine is available to the caller
- Variable isolation can be achieved by using 'Procedure' after the label

```
MyCode: Procedure
```

- The code can have visibility to the caller's variables with 'Expose' after the 'Procedure'

```
MyCode: Procedure Expose var1 var2 var3
```

  - The caller's variables are protected from the function/subroutine unless included in the Expose list

String instructions

Length(string)	*Returns length of string*
Right(string, length, pad)	*Right justify string with pad to length*
Left(string, length, pad)	*Left justify string with pad to length*
Strip(string, opt, char)	*Strip characters from string (leading, trailing, both)*
Space(string, n, pad)	*Returns a string with n pad characters between each word; If n=0, removes all pad characters*
Substr(string, n, len, pad)	*Returns a portion of the string starting at position n for len. If not enough chars, use pad character*
Center(string, len, pad)	*Centers string within len characters, padded on each side*
Copies(string, n)	*Returns n copies of string*

Datatype

- Returns the datatype of a variable (NUM or CHAR)
- Returns true (1) or false (0) if the datatype matches second parameter

```
If Datatype(value, 'N') = 1 Then
    say "It's a number"
```

Alphanumeric	a-z A-Z 0-9
Binary	0 or 1
Lowercase	a-z
Mixed	a-z A-Z
Number	A valid number
Symbol	Valid REXX variable
Uppercase	A-Z
Whole	Whole number
heXadecimal	Hex characters

Select

- Similar to a switch or case statement in other languages
- 'Otherwise' is required and 'nop' can be used
- Evaluates multiple expressions
- Really should be considered when nesting multiple "IF"s

```
Select
    When expression1 Then
       Instruction1
    When expression2 Then
       Instruction2
    When expression3 Then
       Instruction3
    Otherwise
       Instruction4
  End
```

Select - Example

```
/* */
arg age
Select
  When age < 50 Then
    say 'Full warranty'
  When age >= 50 Then
    say 'Time to renew'
  Otherwise
    Nop
End
```

```
examp6 38
Full warranty
```

```
examp6 51
Time to renew
```

RC (Return Code) and Exit

- RC is a special REXX variable that contains the return code from the last host command

```
'CP QUERY MAINT4'
say 'Return code from query' rc
```

```
HCPCQV003E Invalid option - MAINT4
Return code from query 3
```

- The Exit command can be used to terminate a REXX program and optionally pass back a return code

```
Exit 4
```

- It is *good practice* to use Exit at the logical end of the program, even though REXX will exit when it reaches the end anyway

Signal

- Signal with a label name transfers control to the label
- Acts like GOTO in other languages

```
Signal MyRoutine
…
MyRoutine:
```

- Much better used to trap errors in a program

```
Signal ON|OFF condition
```

Conditions:	ERROR	-non-zero rc from Host Command
	HALT	-any external interrupt to halt
	NOVALUE	-an un-initialized variable used
	SYNTAX	-interpretation error detected
	NOTREADY	-input/output not available

Signal

- Trapping errors can use special REXX variables
  - SIGL - Line number of the error
  - Sourceline - A function that returns the error line
  - Errortext - A function that returns the REXX error message

Signal - Example

```
/* */
Signal On Syntax
a = 1 + a

Syntax:
crc=rc
Say Errortext(crc)
Say 'On line' sigl':'
Say Sourceline(sigl)
Exit
```

```
Bad arithmetic conversion
On line 3:
a = 1 + a
```

Host commands

- A command that would typically be entered on the command line

```
Address Command '...'

Address CMS '...'
```

- Address Command is very specific, passing the parameter to CP/CMS without searching
- Address is important in multiple command environments
  - CP, CMS, XEDIT, CGI, etc.

Host commands

- Recognized when a clause
  - Is not a comment
  - Is not a label
  - Is not an assignment
  - Does not begin with a REXX keyword (Do, If, Select, etc.)

Tracing

- Produces diagnostic information about the program
- Various trace options

Error	*Host commands with rc <> 0*
Command	*Host commands before execution*
All	*All clauses*
Results	*All clauses; the result of any expression*
Intermediate	*Intermediate results during expression evaluation*
Labels	*Passing through any labels*
Scan without execution	*Trace without executing; Check for missing ends*
Normal / Failure	*Traces commands with negative return code*
Off	*Turn trace off*

Tracing

- 'S' is good to check code structure
- 'I' is good for comprehensive tracing
  - Can be difficult to read with compute intensive routines
- 'L' is good to see which routines are called

Reading a trace – common tags

-	The source of a single clause/data is in the routine
+++	Trace message
>>>	Result of an expression

Tracing

- Intermediate trace tags

>V>	Contents of a variable
>L>	Literal; string or uninitialized variable
>F>	Result of a function call
>P>	Result of a prefix operation
>O>	Result of an operation on two terms
>C>	Name of a compound variable after substitution

Tracing - Example

```
/* */
trace i
arg age
Select
  When age < 50 Then
    say 'Full warranty'
  When age >= 50 Then
    say 'Time to renew'
  Otherwise
    Nop
End
```

```
examp6 44
     3 *-* arg age
       >>>   "44"
     4 *-* Select
     5 *-* When age < 50
       >V>   "44"
       >L>   "50"
       >O>   "1"
       *-*  Then
     6 *-*  say 'Full warranty'
       >L>    "Full warranty"
Full warranty
```

Input/Output*

- Line input

```
Linein(name, line, count)
```

  - name = Name of the stream
  - line = Starting position
  - count = Number of lines to read

- Omitting name causes Linein to read from the default input stream (eg: stdin)
- Linein() is the equivalent to a Pull
  - Reading input from the stack or console

*There are easier ways! Go to the CMS Pipelines session to learn more

Input/Output

- Line output

```
Lineout(name, string, line)
```

  - name = Name of the stream
  - string = Data to be written
  - count = Number of lines to read

- Returns 0 if successful, 1 if not
- Lineout(name) is the equivalent to closing the stream (file)

Input/Output

- Lines()
- Returns the number of lines left in the stream

```
/* EXAMP11 EXEC */
Do until Lines(examp11 exec) = 0
  say linein(examp11 exec)
End
```

```
examp11
/* EXAMP11 EXEC */
Do until Lines(examp11 exec) = 0
  say linein(examp11 exec)
End
```

Input/Output

- Charin()
  - Reads characters from an input stream
- Charout()
  - Write characters to an output stream
- Chars()
  - Number of characters remaining
- Rarely (if ever) used on z/VM
  - More relevant to other platforms (Linux, Windows, etc)

Value

- Value('count')
  - Returns the value of the variable 'count'
- Value('count', 4)
  - Returns current value of 'count' then sets 'count' to 4
- If the variable is not initialized, the variable name is returned in upper case

Value QUIZ:

```
a33="Hello"
k=3
fred = "K"
/* what do the following produce on the console? */
Say Value("A")              A
Say Value("A"||k||k)        Hello
Say Value(fred)             3
```

Translate

- Translate(string, tableo, tablei, pad)
- Translates characters in string from tablei to tableo
- tablei is a string of source characters
- tableo is a string of target characters

Examples

Translate(‘abcdef’) → ‘ABCDEF’

Translate(‘abbc’, ’&’, ’b’) → ‘a&&c’

Translate(’abcdef’, ’12’, ’ec’) → ‘ab2d1f’

Translate(‘abcdef’, ’12’, ’abcd’,’.’) → ‘12..ef’

Translate – Example in detail

- Translate(“4123”, “abcd”, “1234”)

(“4123”,”abcd”,”1234”) → 4a23

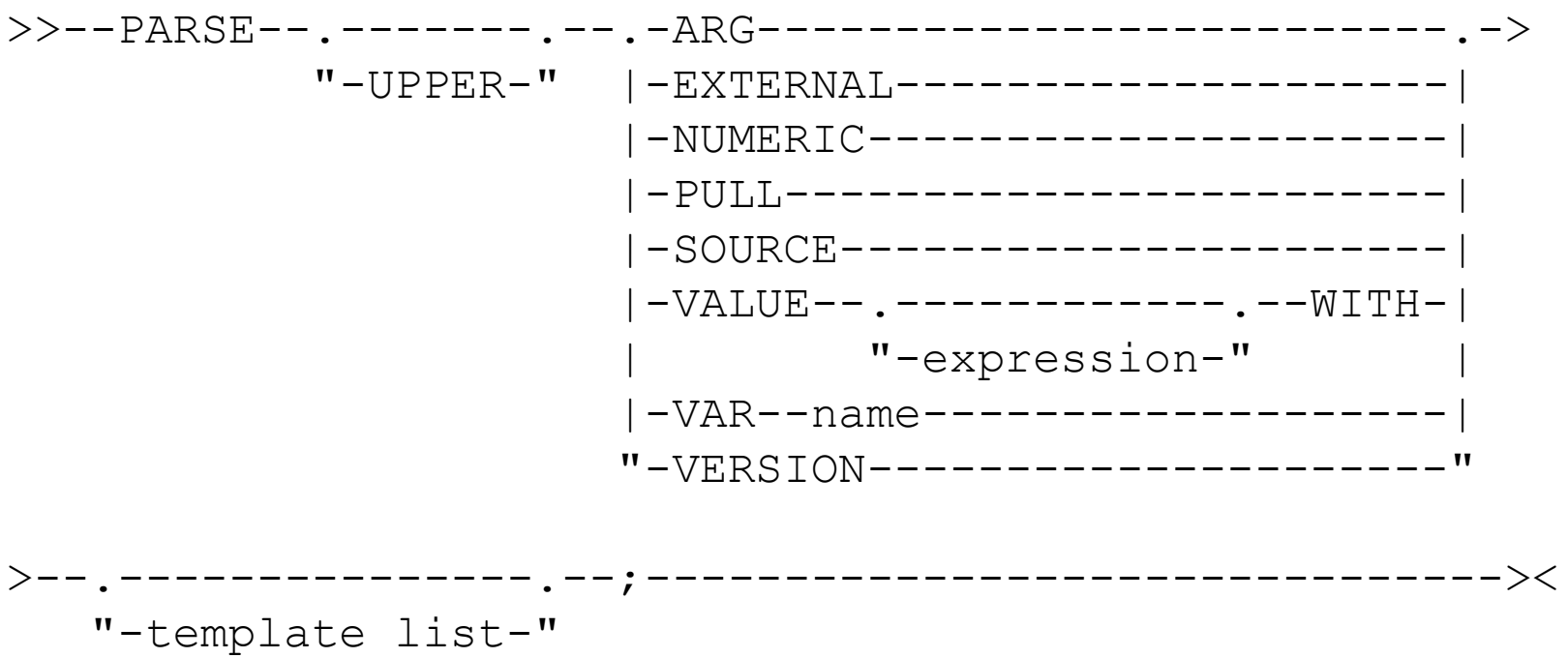
(“4123”,”abcd”,”1234”) → 4ab3

(“4123”,”abcd”,”1234”) → 4abc

(“4123”,”abcd”,”1234”) → dabc

Parse

- A very powerful mechanism to extract value(s) from a string



Parse

- Assigns one or more values from a source string to variables based on a template list

```
Parse Value expression With template
```

- Parses the data that is the result of evaluating *expression*

```
Parse Var variable template
```

- Parses value of *variable* with template

Parse - Examples

```
/* EXAMP13 EXEC */
Parse Value OneTwo() With val1 val2
say val1 val2
Exit

OneTwo:
Return 1 2
```

```
/* EXAMP14 EXEC */
x = OneTwoThree()
Parse Var x val1 . val2
say val1 val2
Exit

OneTwoThree:
Return 1 2 3
```

- Example 13 parses the result of a function

- Example 14 parses the value of a variable ‘x’
  uses ‘.’ as a placeholder for one of the words of the value

Parse - Examples

Parse Value Time() With hh ‘:’ mm ‘:’ ss

– Breaks the current time into hours, minutes, seconds

```
/* EXAMP15 EXEC */
Parse Value Time() With hh ':' mm ':' ss
say hh '-' mm '-' ss
```

```
examp15
12 - 37 - 17
```

Parse – Examples

- Positional parsing

```
Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

Parse Value Time() With hh +2 . +1 mm +2 . +1 ss
10:15:30

…etc…
```

Parse - Examples

`Parse Source template`

- Parses information about the environment and the source of the program being executed

```
/* EXAMP16 EXEC */
Parse Source vals
say vals
Parse Source . . fname ftype fmode .
say fname ftype fmode
```

```
examp16
CMS COMMAND EXAMP16 EXEC A1 examp16 CMS
EXAMP16 EXEC A1
```

Parse - Examples

```
Parse External template
```

- Gets data from the terminal input buffer
- Leaves the program stack intact
  - If you need to have data from the user and don't want to deal with any data in the program stack
- Also allows a REXX program to retrieve data passed when a machine is XAUTOLOGged

External Functions

- An *internal* function is contained within the REXX program
- An *external* function is in a separate REXX program file
- Both are called exactly the same way

```
/* MYNAME EXEC */
Return 'Capt. Obvious'
```

```
/* EXAMP25 EXEC */
Say 'The time is' Time()
Say 'My name is' Myname()
Exit
```

```
examp25
The time is 14:03:31
My name is Capt. Obvious
```

DIAG/DIAGRC

- The DIAG variants can be used to issue some CP Diagnose functions from REXX
  - Diag 8 to issue a CP command

```
Parse Value Diag(8,'CP QUERY USERID') with uid . sysid '15'x
```

Parses the CP QUERY USERID response into vars uid and sysid; '15'x is always included on the end of each returned line

Interpret

- Execute dynamically built REXX instructions

```
Interpret expression
```

- Expression is interpreted as REXX code and then executed
- If you need to execute commands based on the contents of a variable, use Interpret

Interpret - Example

```
/* EXAMP18 EXEC */
arg value
Interpret value 'QUERY VIRTUAL DASD'
```

```
examp18
DASD 0190 3390 M01RES R/O        214 CYL ON DASD  0320 SUBCHANNEL = 0004
DASD 019D 3390 M01RES R/O        292 CYL ON DASD  0320 SUBCHANNEL = 0006
DASD 019E 3390 M01RES R/O        500 CYL ON DASD  0320 SUBCHANNEL = 0005
```

```
examp18 say
QUERY VIRTUAL DASD
```

Program Stack

- `Push` - puts an item on the top of the stack (LIFO)
- `Queue` - puts an item on the bottom of the stack (LIFO)
- `Pull` – retrieves the next item from the top of the stack
- `Queued()` - returns the number of items on the stack
- `Externals()` - returns the number of items on the external queue

Note that REXX only *pulls* info from the top of the queue

Program Stack - Example

```
/* EXAMP19 EXEC */
Push 'Line 1'
Push 'Line 2'
say queued()
Do queued()
  Pull item
  Say item
End
```

```
examp19
2
LINE 2
LINE 1
```

Questions!

Exciting News!!
Velocity Software is now your place for z/VM education!

- **Self-Study and Instructor-led classes**
- **Upcoming Instructor-led Class:**
  - **July 8-10 2026 – Modules 1, 2 and 3 (from our education page)**

Ask about it here at the workshop!
See our website – VelocitySoftware.com/Educate/Training
Send an email to – education@velocitysoftware.com

VELOCITY SOFTWARE