

REXX programming for the z/OS programmer

Session #14019

Wednesday, August 14 at 1:30 pm
Hynes Convention Center Room 207

Brian J. Marshall



Abstract and Speaker



- Rexx is a powerful yet relatively simple High Level language that provides great flexibility and power on the z platform as well as many other platforms.
- Brian J. Marshall is the Vice President of Research and Development at Vanguard Integrity Professionals



Trademarks

- The following are trademarks or registered trademarks of the International Business Machines Corporation:
 - IBM Logo
 - z/OS
 - MVS/ESA
 - S/390
 - DB2
 - OS/390
 - MVS/DFP
 - RACF
 - Series z
 - IBM Business Partner emblem
- UNIX is a registered trademark of The Open Group in the United States and other countries.

Agenda

The Start of a rexx program

Comments

Variables

Operators

Conditional Expressions

Iterations, iterations, iterations...

SIGNAL, EXIT, CALL, RETURN & LABEL

Functions (built-in and user defined)

PARSE

Stacks and Queues

Environments

Execio

Questions

How to Start a REXX Program

The first line of a REXX program must be a comment and must contain the Word or at least the letters REXX

Ex:

VALID:

```
/* THIS IS MY FIRST REXX PROGRAM */  
/* REXX */
```

NOT VALID

```
/* This does not
```

Count as valid REXX program start */

Comments

Comments in REXX can start at any position and end in any position.

Comment Start is a /*

Comment End is an */

Careful with embedded comments.

USE HILITE Language: #14 (for REXX) and Coloring 3 (Both IF and DO Logic)

Variables

Variable names can consist of Alpha, Numerics, Special characters and DBCS Characters when OPTIONS ETMODE is specified.

Periods should only be used in STEM Variables

Variables names should NOT BE: RC, SIGL or RESULT

Variables ARE NOT strongly typed and are NOT CASE SENSITIVE.

STEM Variables are variables that like arrays can have multiple values. The STEM of the variable can actually be anything, usually a number where BY CONVENTION The 0th element contains the number of elements in the variable.

Stem Variables

When a stem is used as the target of an assignment, *all possible* compound variables whose names begin with that stem receive the new value, whether they previously had a value or not. Following the assignment, a reference to any compound symbol with that stem returns the new value until another value is assigned to the stem or to the individual variable.

Since Stem variables can use lots of available storage, DROP them when your done with them.

For example:

hole. = "empty" ← Initialized all elements of the stem to "empty"

hole.9 = "full"

say	hole.1	hole.9	hole.anything
Drop	hole.	← An example of dropping the variable.	
	empty	full	empty

OPERATORS - Arithmetic

+	Add
-	Subtract
*	Multiply
/	Divide
%	Integer Divide
//	Modulo – Integer divide and return whole remainder
**	Power

Order or Parenthesis, Operations, etc all apply as one would expect.

EX: Sum2Nums = Num1 + Num2

OPERATORS – Logical or Boolean

&	And
 	Or
&&	Exclusive Or
\	Not (Logical Negation) or caret or

Order or Parenthesis, Operations, etc all apply as one would expect.

Logical expressions return a true (1) or a false (0) value

Ex: A = 0 and B = 1

(A & B)	0 False	(B && B)	0 False
(A B)	1 True	(\A)	1 True

OPERATORS – Comparison / Equality

=	Equal	<>	Same as \=
==	Strictly Equal	>=	GT or Equal To
\=	Not Equal	<=	LT or Equal To
\==	Strictly Not Equal	\<	Not Less Than
>	GT	\>	Not Greater Than
<	LT	>>=	Strictly greater than or equal to
\<<, ¬<<	Strictly NOT less than	<<=	Strictly less than or equal to
\>>, ¬>>	Strictly NOT greater than		

Strictly vs. Not Strictly When two expressions are strictly equal, everything including the blanks. Otherwise with an = 'word' is equal to 'word' but not true with ==.

Note: I have seen documented that = also ignores case. This is not true to my knowledge.

OPERATORS – Concatenation

- Blank** Concatenates terms and places one blank between them.
Multiple blanks becomes a single blank.
- || or abuttal** Concatenates terms without a blank. Use || as it is easier to read and debug later. Careful of | vs. ||

EX:

say “first” “second”

say “first”||”second”

say “first””second”

output

first second

firstsecond

firstsecond

OPERATORS – Precedence

Unary operators before Binary Operators

Binary Operators by precedence

Equal Operators Left to Right

Prefix Operations

Power

Multiple and Divide

Add or Subtract

Conditional Expressions – IF/THEN/ELSE

Non compound instructions:

IF expression THEN

instruction

ELSE

instruction

For compound instructions:

IF expression THEN

Do

instruction-1

instruction-2

End

Nested IF/THEN/ELSE's are allowed. Make sure to match each IF with an ELSE and each DO with an END. Line up the DO and ENDS and have hilite turned on For both IF and DO Logic.

Conditional Expressions - Select

SELECT

WHEN expression-1 THEN instruction-1

WHEN expression-2 THEN instruction-2

WHEN expression-3 THEN instruction-3

OTHERWISE

instruction(s)

END

Make sure you END your SELECT. Expressions should evaluate to T or F.

**For THENs, multiple instructions can be used but must be encapsulated in DO
End.**

For Otherwise, all instructions until the END associated with the SELECT are

WHEN expression-1 THEN instruction-1

executed.

Complete your sessions evaluation online at SHARE.org/BostonEval

Iteration, iteration, Iteration: DO

The DO is the simplest form of Iteration:

DO 2

say ' RED SOX RULE – NOT'

End

Result:

RED SOX RULE – NOT

RED SOX RULE – NOT

Btw, Say works online and in batch. Online to terminal, batch to SYSTSPRT

Iteration, iteration, Iteration: DO Controlled

The DO is the simplest form of Iteration:

DO i = 1 to 3

say i

END

1

2

3

DO i=1 to 10 by 3

say i

END

1

4

7

10

The DO in use with a STEM variable

The DO loop is great for filling or displaying STEM variables...

Parse arg n

Stem.0 = N

DO i=1 to N

 Stem.i = i

END

Do i=1 to stem.0

Say stem.i

End

Outputs numbers 1 to N where N is passed in via call or exec.

Iteration, iteration, Iteration: DO FOREVER

The DO FOREVER is a loop that will continue until the LEAVE or EXIT statement are encountered

DO FOREVER

Pull Var1

if Var1 = 'STOP' then Leave

say Var1

END

Echoes output until stop or STOP is entered.

LEAVE tells Program to leave the loop, EXIT ends a rexx program

PULL will uppcase any value entered. Use PARSE PULL to avoid this if desired.

Iteration, iteration, Iteration: DO WHILE

The DO WHILE is a loop that will continue WHILE the condition is TRUE

Var1 = ‘

DO While Var1 <> ‘STOP’

 Pull Var1

 say Var1

END

Echoes output until stop or STOP is entered but will echo STOP as well

Iteration, iteration, Iteration: DO UNTIL

The DO UNTIL is a loop that will continue until the condition is TRUE

Var1 = ‘

DO UNTIL Var1 = ‘STOP’

 Pull Var1

 say Var1

END

Echoes output until stop or STOP is entered but will echo STOP as well

Note: Loops can be nested and nested and nested and nested

SIGNAL, EXIT, CALL, RETURN & LABEL

- SIGNAL** causes an unconditional branch to another instruction.
Signal should really only be used with events.
- EXIT** Causes an exec to unconditionally end and return to where it was invoked. EXIT can return a value to caller as well via the variable RESULT
- CALL** causes control to be passed to an internal or external subroutine. Internal subroutines are referenced by a LABEL
- RETURN** returns control back to the calling exec and may return a value as well.
- LABEL** Symbolic name followed by a colon.

REXX Built in Functions

Rexx has a number of built in functions like most languages. See Appendix A For explanation of each.

Arithmetic	(ABS,DIGITS,FORM,FUZZ,MAX,MIN,RANDOM,SIGN,TRUNC)
Comparison	(Compare, Datatype, Symbol)
Conversion	(B2X, C2D, C2X, D2C, D2X, X2B, X2C, X2D)
Format	(Center, Copies, Format, Justify, Left, Right, Space)
String	(Abbrev, Delstr, Delword, Find, Index, Insert, Lastpos Length, Overlay, Pos, Reverse, Strip, Substr, Subword, Translate, Verify, Word, Wordindex, Wordlength, words)
Misc	(Address, Arg, Bitand, Bitor, Bitxor, Condition, Date, Errortext, Externals, Linesize, Queued, Sourceline, Time, Trace, Userid, Value, Xrange)

Writing your own Subroutine

Rexx allows subroutines to act as either a procedure or true function.

A Function is a callable routine that calculates and MUST return a value

A Subroutine is a set of code that accomplishes a task.

You can pass up to 20 arguments into a subroutine but return only ONE (Stored in variable called RESULT).

A Subroutine suffixed with the word PROCEDURE will protect the variables in and make them all local variables. This can be changed with

A PROCEDURE EXPOSE, where the exposed variables are not local

Ex:

subroutine1:

Subroutine2: Procedure

Subroutine3: Procedure Expose Answer

Keyword Instructions

A **keyword instruction** is one or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer.

See Appendix B for a list of Keyword Instructions

Writing your own Subroutine

Consider the following examples:

N1=5	N1=5	N1=5
N2=10	N2=10	N2=10
Call subr	call subr	Call Subr
Say ans	say ans	say ans
Exit	Exit	Exit
Subr:	Subr: PROCEDURE	Subr: PROCEDURE EXPOSE ANS
Ans = n1 + n2	Ans = n1 + n2	n1=50
Return	Return	n2 =100
		ans= n1+n2
		Return

Output:

15 Error Line: Ans=n1 +n2 150

Pass Values into Subroutines

You can pass values into a Subroutine on the CALL statement by specifying up to 20 arguments separated by commas.

You place the passed variables/values into new variables via the ARG function, either with an ARG or definitive assignment $X = \text{ARG}(1)$ or through the use of $\text{ARG}(1)$ as a variable in the subroutine.

Ex:

Call perimeter L,W

Say “Perimeter is: “ Result

Exit

Perimeter: Procedure

Arg Length, width

Return $2 * \text{length} + 2 * \text{width}$

Call perimeter L,W

Say “Perimeter is: “ Result

Exit

Perimeter: Procedure

return $2 * \text{Arg}(1) + 2 * \text{Arg}(2)$

Pass Values into a Function

You can invoke a true **FUNCTION** without a **CALL** statement and again pass up to 20 arguments separated by commas. In this case encapsulated by ()'s. You place the passed variables/values into new variables via the **ARG** function, either with an **ARG** or definitive assignment **X = ARG(1)** or through the use of **ARG(1)** as a variable in the subroutine.

Ex:

X = perimeter(l,w)

Say “Perimeter is: “ x

Exit

Perimeter: Procedure

Arg Length, width

Return 2*length + 2* width

Say “Perimeter is: “perimeter(l,w)

Exit

Perimeter: Procedure

return 2*Arg(1) + 2*Arg(2)

The PARSE Command

REXX is an EXCELLENT language for parsing. It makes otherwise difficult Parsing scenarios easier.

PARSE PULL reads input from data stack or terminal and assigns them to variables w/o Modification. PULL otherwise will uppcase

PARSE PULL A B C

will take three values from stack or user and place into Variables A B and C

PARSE ARG reads variables from calling routine and assigns them to variable w/o Modification. ARG otherwise will uppcase
a='This' b='is'

The PARSE Command

PARSE ARG reads variables from calling routine and assigns them to variable w/o Modification. ARG otherwise will uppcase

Var1 = 'This'

Var1 = 'This'

Var2 = 'is'

Var2 = 'is'

Var3 = 'Passed'

Var3 = 'Passed

Call subr Var1 Var2 Var3

Call subr Var1 Var2 var3

Exit

Exit

Subr: Procedure

Subr: Procedure

Parse Arg s1 s2 s3

Arg s1 s2 s3

Say s1 s2 s3

Say s1 s2 s3

Return

Return

This is Passed

THIS IS PASSED

The PARSE Command

PARSE VAR Parses a variable into one or more variables that follow it

ParsedString = “This is the String to be Parsed”

PARSE VAR ParsedString X1 X2 X3 X4 X5 .

Say X1 This

Say X2 is

Say X3 the

Say X4 String

Say X5 to

Exit

The period at the end or anywhere in the parse variables is used as a Placeholder. It is a good practice when not parsing all data or potential data in a variable to end the parse with a period.

The PARSE Command

PARSE VAR Separator

Parses a variable into one or more variables by the Separator

ParsedString = “This is the String to be Parsed”

PARSE VAR ParsedString X1 X2 “be” X5 .

Say X1 This

Say X2 is

Say X5 Parsed

Exit

The PARSE Command

PARSE VAR (Separator)

Parses a variable into one or more variables by a variable
Separator .

ParsedString = "This is the String to be Parsed"

X4= "be"

PARSE VAR ParsedString X1 X2 (X4) X5 .

Say X1 This

Say X2 is

Say X5 Parsed

Exit

The PARSE Command

PARSE VAR Number

Parses a variable into one or more variables by the nth character in the VARIABLE.

ParsedString = “This is the String to be Parsed”

PARSE VAR ParsedString X1 X2 =13 X5 .

Say X1	This
Say X2	is the
Say X5	String

Exit

Note:

This can be used to split a string.

The PARSE Command

PARSE VALUE WITH

Parses a value into a set of variables using the blank as a separator

Parse Value “This is the String to be parsed” with X1 X2 X3 X4 X5 . X6

Say x1	This
Say x2	is
Say x3	the
Say x4	String
Say x5	to
Say x6	parsed

exit

STACKS and QUEUES

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

Stacks and queues are similar types of data structures used to temporarily hold data items (elements) until needed. When elements are needed, they are removed from the top of the data structure.

The basic difference between a stack and a queue is where elements are added.

Elements are added to the top of a stack and to the bottom of a queue.

STACKS and QUEUES

PUSH - puts one item of data on the top of the data stack. There is virtually no limit to the length of the data item.

elem1 = 'String 1 for the data stack'

PUSH elem1

QUEUE - puts one item of data on the bottom of the data stack. Again, there is virtually no limit to the length of the data item.

elemA = 'String A for the data stack'

QUEUE elemA

STACKS and QUEUES

To remove data elements from the Stack we use PULL

```
PULL stackitem  
SAY    stackitem
```

If you do not want the values uppercased then use:

```
PARSE PULL stackitem  
SAY stackitem
```

STACKS and QUEUES

When an exec calls a routine (subroutine or function) and both the exec and the routine use the data stack, the stack becomes a way to share information. However, execs and routines that do not purposely share information from the data stack, might unintentionally do so and end in error. To help prevent this, TSO/E provides the MAKEBUF command that creates a buffer, which you can think of as an extension to the stack, and the DROPBUF command that deletes the buffer and all elements within it.

Although the buffer does not prevent the PULL instruction from accessing elements placed on the stack before the buffer was created, it is a way for an exec to create a temporary extension to the stack. The buffer allows an exec to:

Use the QUEUE instruction to insert elements in FIFO order on a stack that already contains elements.

Have temporary storage that it can delete easily with the DROPBUF command. An exec can create multiple buffers before dropping them. Every time MAKEBUF creates a new buffer, the REXX special variable RC is set with the number of the buffer created. Thus if an exec issues three MAKEBUF commands, RC is set to 3 after the third MAKEBUF command

STACKS and QUEUES

Sometime a STACK is NOT meant to be shared. If this is the case then 'NEWSTACK' is better suited as opposed to MAKEBUF

To protect elements on the data stack, you can create a new data stack with the TSO/E REXX NEWSTACK command.

Any routine that uses 'NEWSTACK' should issue a DELSTACK for each stack created

The DELSTACK command removes the most recently created data stack. If no stack was previously created with the NEWSTACK command, DELSTACK **removes all the elements from the original stack. (THIS CAN HURT)**

Environments

Rexx can run in a number of different environments. Here is a sample of them
The environment you are running in dictates the commands that are available.

You can move between environments using the ADDRESS command

ADDRESS TSO	Will put you in a TSO environment (assuming one is available)
--------------------	--

ADDRESS ISPEXEC	Will put you into an ISPF environment
------------------------	--

ADDRESS ISREDIT	Execute a macro
------------------------	------------------------

ADDRESS MVS	Will put you into an MVS environment
--------------------	---

ADDRESS SYSCALL	Unix commands.
------------------------	-----------------------

ADDRESS SH	Unix Shell
-------------------	-------------------

ADDRESS()	Will return you your current environment
------------------	---

EXECIO DISKR DISKRU

- EXECIO** Is a method to read z/OS PDS members and Sequential files.
- DISKR** Open and read from a file (read only)
- DISKRU** Open and read from a file (update allowed)
- DISKW** Open and write to a file
- “EXECIO xxx DISKR(U) INDD yyy (FINIS”**
(OPEN”
(STEM stem.”
(LIFO”
(FIFO”
(SKIP”
- xxx** is the number of lines to READ. * means read to EOF
- yyy** is the starting line a which to begin READING from (optional, defaults to either beginning of file or last line read +1)
- INDD** FILE DD to read from
- OPEN** OPEN dataset and position before first record
- STEM** Specifies the stem variable into which the records will be placed.
 If not specified, records will be placed onto the Datastack
- FINIS** Close the dataset after reading
- LIFO, FIFO and SKIP – PUSH or QUEUE onto STACK or SKIP xxx lines**

EXECIO DISKW

EXECIO Is a method to write to z/OS PDS members and Sequential file

DISKW Open and write to a file

“EXECIO xxx DISKW OUTDD (FINIS”
(OPEN”
(STEM stem.”

xxx is the number of lines to Write. * means write all from stack or STEM

OUTDD File DD to write to

OPEN OPEN dataset and position before first record

STEM Specifies the stem variable from which records are read from and written to the file.

If not specified, records will be placed onto the Datastack

FINIS Close the dataset after writing. Forces i/o completion.

Quick Example

```
“ALLOC FI(INDD)   DA(‘my.input.dataset’) SHR REUSE”
“ALLOC FI(OUTDD) DA(‘my.output.dataset’) SHR REUSE”
Stem.0 = 0
Myrc   = 0                               /* assume 0 return code */
Execio * DISKR INDD (Stem stem.”        /* read entire file into stem. Variable*/
If rc > 4 then call ERRORRTN “READING FILE”
Execio * DISKW OUTFF (STEM stem.”      /* writes entire STEM to file */
If rc > 4 then call ERRORTN “WRITING FILE”
Exit:
    “EXECIO 0 DISKW OUTDD (FINIS”      /* closes files */
    “EXECIO 0 DISKR INDD  (FINIS”
EXIT
ERRORTN:
ARG S1
say “ An Error occurred during “ S1
myrc = 12
return myrc
Return          /* never executed*/
```

Questions?



Appendix A

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

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

Function	Description
B2X	Returns a string, in character format, that represents the input binary string converted to hexadecimal. (Binary to Hexadecimal)
C2D	Returns the decimal value of the binary representation of the input string. (Character to Decimal)
C2X	Returns a string, in character format, that represents the input string converted to hexadecimal. (Character to Hexadecimal)
D2C	Returns a string, in character format, that represents the input decimal number converted to binary. (Decimal to Character)
D2X	Returns a string, in character format, that represents the input decimal number converted to hexadecimal. (Decimal to Hexadecimal)
X2B	Returns a string, in character format, that represents the input hexadecimal string converted to binary. (Hexadecimal to Binary)
X2C	Returns a string, in character format, that represents the input hexadecimal string converted to character. (Hexadecimal to Character)
X2D	Returns the decimal representation of the input hexadecimal string. (Hexadecimal to Decimal)

Appendix A

Function	Description
ADDRESS	Returns the name of the environment to which commands are currently being sent.
ARG	Returns an argument string or information about the argument strings to a program or internal routine.
BITAND	Returns a string composed of the two input strings logically ANDed together, bit by bit.
BITOR	Returns a string composed of the two input strings logically ORed together, bit by bit.
BITXOR	Returns a string composed of the two input strings eXclusive ORed together, bit by bit.
CONDITION	Returns the condition information, such as name and status, associated with the current trapped condition.
DATE	Returns the date in the default format (dd mon yyyy) or in one of various optional formats.
ERRORTXT	Returns the error message associated with the specified error number.
EXTERNALS *	Returns the number of elements in the terminal input buffer. In TSO/E, this function always returns a 0.
LINESIZE *	Returns the current terminal line width minus 1.
QUEUED	Returns the number of lines remaining in the external data queue at the time when the function is invoked.
SOURCELINE	Returns either the line number of the last line in the source file or the source line specified by a number.
TIME	Returns the local time in the default 24-hour clock format (hh:mm:ss) or in one of various optional formats.
TRACE	Returns the trace actions currently in effect.
USERID *	Returns the TSO/E user ID, if the REXX exec is running in the TSO/E address space.
VALUE	Returns the value of a specified symbol and optionally assigns it a new value.
XRANGE	Returns a string of all 1-byte codes (in ascending order) between and including specified starting and ending values

Appendix B Keyword Instructions

ADDRESS temporarily or permanently changes the destination of commands. Commands are strings sent to an external environment. You can send commands by specifying clauses consisting of only an expression or by using the ADDRESS instruction

ARG retrieves the argument strings provided to a program or internal routine and assigns them to variables. It is a short form of the instruction: PARSE UPPER ARG

CALL calls a routine (if you specify *name*) or controls the trapping of certain conditions (if you specify ON or OFF).

DO groups instructions together and optionally processes them repetitively. During repetitive execution, a control variable (*name*) can be stepped through some range of values.

DROP "unassigns" variables, that is, restores them to their original uninitialized state. If *name* is not enclosed in parentheses, it identifies a variable you want to drop and must be a symbol that is a valid variable name, separated from any other *name* by one or more blanks or comments

Appendix B Keyword Instructions

EXIT leaves a program unconditionally. Optionally EXIT returns a character string to the caller.

IF conditionally processes an instruction or group of instructions depending on the evaluation of the *expression*. The *expression* is evaluated and must result in 0 or 1.

INTERPRET processes instructions that have been built dynamically by evaluating *expression*.

ITERATE alters the flow within a repetitive DO loop (that is, any DO construct other than that with a simple DO).

LEAVE causes an immediate exit from one or more repetitive DO loops (that is, any DO construct other than a simple DO).

NOP is a dummy instruction that has no effect. It can be useful as the target of a THEN or ELSE clause

Appendix B Keyword Instructions

NUMERIC changes the way in which a program carries out arithmetic operations

OPTIONS passes special requests or parameters to the language processor. For example, these may be language processor options or perhaps define a special character set.

according to the rules of parsing.

PROCEDURE, within an internal routine (subroutine or function), protects variables by making them unknown to the instructions that follow it. After a RETURN instruction is processed, the original variables environment is restored and any variables used in the routine (that were not exposed) are dropped.

PULL reads a string from the head of the external data queue

PUSH stacks the string resulting from the evaluation of *expression* LIFO (Last In, First Out) onto the external data queue.

Appendix B Keyword Instructions

QUEUE appends the string resulting from *expression* to the tail of the external data queue. That is, it is added FIFO (First In, First Out).

RETURN returns control (and possibly a result) from a REXX program or internal routine to the point of its invocation

SAY writes a line to the output stream.

SELECT conditionally calls one of several alternative instructions

SIGNAL causes an *unusual* change in the flow of control (if you specify *labelname* or VALUE *expression*), or controls the trapping of certain conditions

TRACE controls the tracing action

UPPER translates the contents of one or more variables to uppercase. The variables are translated in sequence from left to right.

Appendix C Batch JCL

```
//STEP1 EXEC PGM=IRXJCL,PARM='MYEXEC A1 b2 C3 d4'  
//*  
//STEPLIB  
//* Next DD is the data set equivalent to terminal input  
//SYSTSIN DD DSN=xxx.xxx.xxx,DISP=SHR,...  
//*  
//* Next DD is the data set equivalent to terminal output  
//SYSTSPRT DD DSN=xxx.xxx.xxx,DISP=OLD,...  
//*  
//* Next DD points to a library of execs  
//* that include MYEXEC  
//SYSEXEC DD DSN=xxx.xxx.xxx,DISP=SHR
```

Appendix D Manual

Most of the data for this presentation was taken from the following manual:

<http://publib.boulder.ibm.com/infocenter/zos/v1r13/index.jsp?topic=%2Fcom.ibm.zos.r13.ikjc300%2Fikj4c310.htm>