SLAC-422
SLAC-R-422

PROCEEDINGS OF THE REXX SYMPOSIUM FOR DEVELOPERS AND USERS

May 18-20, 1993
La Jolla, California

SLAC-Report-422
September, 1993

Prepared for the Department of Energy
under contract number DE-AC03-76SF00515

STANFORD LINEAR ACCELERATOR CENTER
Stanford University • Stanford, California

2/80

SLAC–422
CONF–9205149
UC–405
(M)

PROCEEDINGS OF THE REXX SYMPOSIUM FOR DEVELOPERS AND USERS

May 18–20, 1993
La Jolla, California

Convened by
STANFORD LINEAR ACCELERATOR CENTER
STANFORD UNIVERSITY, STANFORD, CALIFORNIA 94309

Program Committee

Cathie Dager of SLAC, Convener
Forrest Garnett of IBM
Jim Weissman of Failure Analysis
Bebo White of SLAC

Prepared for the Department of Energy
under Contract number DE-AC03-76SF00515

Printed in the United States of America. Available from the National Technical Information Service, U.S. Department of Commerce, 5285 Port Royal Road, Springfield, Virginia 22161. Price: Printed Copy A11, Microfiche A01.

PROCEEDINGS OF THE REXX SYMPOSIUM
FOR DEVELOPERS AND USERS

TABLE OF CONTENTS

SUMMARY

The fourth annual REXX Symposium for Developers and Users was held on May 3–5, 1993 in La Jolla, California. Fifty-seven people attended with representatives from seven foreign countries and thirteen states.

The program of this year's symposium indicated that REXX is continuing to gain momentum and support with users of all major platforms and operating systems and that REXX developers are busy providing the tools demanded by these users. Of particular note are developments in the OS/2, Windows and UNIX arenas. Several new tools were demonstrated which indicate how adaptable REXX is with current object-oriented and graphical interface methodologies.

Attendees to the symposium were introduced to other scripting languages which users must evaluate when considering REXX. Larry Wall, the author of Perl, gave an overview of that language and "compared notes" with Mike Cowlishaw in a joint session. Bob O'Hara of Microsoft gave a presentation on the use of Visual BASIC and described the major emphasis being placed upon this language. These presentations were not intended as comparisons between these languages and REXX, but rather as important exchanges on the design and role of scripting languages.

This symposium also brought the beginning of the first serious efforts to form a REXX users group. The name agreed upon was the REXX Language Association (RexxLA). The goal of this organization is to promote the use of REXX and to address the needs and concerns of the REXX user community. An ad hoc governing committee resulted from discussions at the Symposium.

Next year's symposium will be held in Boston, Massachusetts on April 25–27, 1994.

Signed,

1993 Program Committee:

Cathie Dager (SLAC)
Forrest Garnett (IBM)
Jim Weissman (Failure Analysis Associates)
Bebo White (SLAC)

REXX I/O ON VM

GARY BRODOCK
IBM

REXX I/O on VM

Gary Brodock

IBM G09/16-4
P.O. Box 8009
Endicott, NY 13760
607-752-5134

5/18/93

Contents REXX I/O

GKB

Introduction REXX I/O

- REXX Language statements for input and output
- The VM support follows the defined REXX I/O model
  - Seven new builtin functions
  - A new variant on the PARSE instruction
- Additional related support
  - A new NOTREADY condition
  - SIGNAL and CALL enhancements
- The VM I/O model

GKB

Introduction /Notes REXX I/O

The definition for input and output is published in "The REXX Language" by Mike Cowlishaw. It was implemented in the OS/2 operating system in 1990 and this definition is what will be followed for VM. The goal is to provide input and output capability through REXX language statements on the VM platform.

I/O is defined through the use of seven new builtin functions, three for character based operations, three for line based operations, and one for general stream functions. In addition, a new variant for PARSE, PARSE LINEIN, is added to read a line from the default input stream and parse the contents according to the template.

One new condition (in addition to ERROR, HALT, SYNTAX, and NOVALUE) is provided to handle error conditions while doing I/O. This is the NOTREADY condition and it is supported by the SIGNAL and CALL instructions for error handling.

GKB

The General I/O Model

REXX I/O

- CMS is a line based system
- Line based operations are easy
- Character based operations need special support
  - Need to buffer data
  - Perhaps need to indicate line ends
- Two I/O pointers, one buffer
  - Can cause extra I/O when doing both input and output in the same data stream

GKB

The General I/O Model /Notes

REXX I/O

All CMS functions read and write lines of data, not characters. Since the line functions read and write complete lines, they fit into the CMS model very well. When character operations are requested, special processing must be done to properly interface with the line oriented operating system. On input, a data stream line is read and placed in a buffer. Then, characters are given to the user as requested. If more characters are requested than are available, another data line is read and processing continues. On output, the characters supplied by the user are placed in the buffer and written as a line to the data stream only under certain conditions. These conditions are: either I/O pointer is changed, a line end character is given by the user and the data stream is in TEXT mode (explained later), the stream is closed, or an entire buffer of data is given. Otherwise, the data just remains in the buffer and gets added to on the next output operation.

TEXT mode is specified on the STREAM function when opening a data stream. The meaning of this option is that line end characters are significant when doing character based I/O. On input, a line end character is appended to the data at the end of each line read in. On output, the character string to be written is scanned for line end characters and appropriate lines are written as they are found. If any character can be in the data stream, then the user should open the stream with the BINARY option, signifying that line end checking should not be done.

Since there are two I/O pointers and only one buffer, doing both character based input and output on the same stream can cause some flip-flop of the buffered data. This could cause performance degradation.

GKB

Line functions

REXX I/O

- LINEIN(<name><,<line><,count>>)

  returns 0 or 1 lines from the input stream
  - name - the stream name
    line - the line number to read
    count - 0 or 1, the number of lines to read
- PARSE LINEIN template
- LINEOUT(<name><,<string><,line>>)

  returns the number of lines not written (0 or 1)
  - name - the stream name
    string - the line to be written
    line - the line number to write
- LINES(<name>)

  returns the number of lines remaining in the input stream
  - name - the stream name

GKB

Line functions /Notes

REXX I/O

The three line functions are used when the I/O is to be performed a line at a time. LINEIN will read COUNT (either 0 or 1) lines from the input stream, NAME. This will by default read the line at the current read position unless another line is specified by the LINE parameter. If zero is specified for the COUNT, then the current read position is set to LINE and nothing is read from the stream. If part of a line has already been read with the CHARIN function and LINE is not specified on the command, LINEIN will return only the remainder of the line.

The new variant of PARSE, "PARSE LINEIN template", can be used to read a line from the default input stream and parse it according to the supplied template. This is a shorter form of "PARSE VALUE LINEIN() WITH template"

LINEOUT will write the line contained in the STRING parameter to the stream NAME. This write will by default start at the current write position unless a new position is specified by the LINE parameter. If there are characters in the buffer from a previous CHAROUT and a new line is not specified on the LINEOUT call, the STRING is appended to the characters already in the buffer and then the entire line is written. If STRING and LINE are both omitted, the stream will be closed.

The LINES function will return the number of completed lines remaining in the input stream. This count may include a partial line if the stream has been read with the CHARIN function.

GKB

The Character functions

- CHARIN(<name><,<start><,length>>)

  returns LENGTH characters from the input stream

  - name - the stream name
    start - starting character number (only 1 is valid)
    length - number of characters to read

- CHAROUT(<name><,<string><,start>>)

  returns the number of characters not written

  - name - the stream name
    string - the string of characters to write
    start - starting character number (only 1 is valid)

- CHARS(<name>)

  returns 0 if no more characters, 1 if there are

  - name - the stream name

The Character functions /Notes

The three character based functions are used when the I/O is to be done as a string of characters and not whole lines. The CHARIN function will read in a specified number of characters, LENGTH, from a stream, NAME. Optionally, a start position of 1 can be specified to reset the read position to the beginning of the stream. CHAROUT will write a string of characters, STRING, to a stream, NAME. Again, optionally, a start position of 1 can be specified to start writing at the beginning of the stream. A read and a write position is maintained for each persistent stream and if not reset, reading and writing will start from these positions. The CHARS function can be used to indicate if there are more characters in the input stream NAME.

If the name of the stream is omitted, characters are read or written to the default stream. If the stream was opened with the TEXT option, then a LINEEND character is appended to the input characters at the end of each line. For output operations on a stream opened with the TEXT option, the string is scanned for LINEEND characters and lines are written as appropriate. The LINEEND character is not written to the stream in this case. For streams opened with the BINARY option, no indication of line ends is given on input and output records are written when the buffer is filled.

The STREAM function

- STREAM(name<,action<,stream_command>>)

  - returns a string describing the state of, or the result of an operation upon, the stream

  - action - Command, State, or Description

    Command - perform the stream_command given on the named stream

    State - return a string indicating the state of the named stream

    Description - same as State, with additional information: return code and reason code from the last I/O

  - stream_command - specific command to be performed on the named stream

The STREAM function /Notes

The STREAM function is used to get the status of a stream or to perform an operation on a stream. Users can request the State, request the Description, or issue a Command. State will return only the current state of the stream. Description will return the state and also the return and reason codes from the last I/O done on the stream. If Command is specified for ACTION, then a STREAM_COMMAND must be given. The various commands are described on a later foil.

Stream states — REXX I/O

- ERROR - an I/O has caused an error condition
- NOTREADY - an I/O has made the stream not ready and I/O to that stream could raise the NOTREADY condition
- READY - the stream is ready for I/O
- UNKNOWN - the stream is closed or has not been opened yet

Stream states /Notes — REXX I/O

There are four states that a stream can be in. ERROR means that the stream was subject to an erroneous operation, such as a disk problem when writing to a minidisk file. NOTREADY is similar to ERROR, only it usually means that recovery is easier. This could be the case when the user tries to set the read or write pointer to a nonexistent line in the stream or to read past the end of the stream. Recovery would be to reset the pointer to a valid place and then continue the input or output. READY specifies that the stream is ready for I/O operations but does not guarantee that an operation will succeed. UNKNOWN specifies that REXX does not know the state of the stream, such as when the stream does not exist or it has not been opened yet.

Stream_commands — REXX I/O

- OPEN <options>
  - READ/WRITE/NEW/REPLACE
  - LRECL nnnn
  - TEXT/BINARY
  - LINEEND xx
- CLOSE
- LINEPOS offset type
  - offset is a whole number optionally preceded by =, <, +, or -
  - type is READ or WRITE
- QUERY option
  - DATETIME
  - EXISTS
  - FORMAT
  - INFO
  - LINEPOS READ
  - LINEPOS WRITE
  - SIZE

Stream_commands /Notes — REXX I/O

OPEN: is used to open a stream, with additional options to tell the characteristics that are desired.

READ: specifies that the stream will be opened for read only and it must exist.

WRITE: specifies that the stream will be opened for read/write and it will be created if it doesn't exist.

NEW: specifies the stream will be opened for read/write and it must not exist already.

REPLACE: specifies that the stream will be opened for read/write and will be created if it doesn't exist or be replaced (old version thrown away) if it does.

LRECL: indicates the size of the buffer that will be used for input or output. Most of the time, for an existing stream, the current lrecl will be used. However, for new streams or certain existing streams, this is not known and users will have to specify it or take the default of 1024.

TEXT: specifies that line end characters are significant when doing character based operations. This means that a LINEEND character is appended to the input string at the end of each line as an indication that the line is complete. On character output operations, lines are written when *the LINEEND character is encountered in the string.*

BINARY: means that all character codes may be present in the data stream and no indication of LINEEND characters will be provided or searched for. Line based operations are not affected by this option.

LINEEND: specifies the character to be used to indicate line ends. This can be specified as one or two hexadecimal digits with the default being 15.

CLOSE: is used to write out any data left in the buffer due to a character output operation and close the stream.

LINEPOS: is used to change the read or the write line pointer to the beginning of a specified line. The specification can be just a number, such as 10, meaning to move the pointer to line 10. The same move can be made using =10. If you want to move to an offset from the end of the stream, use the <. <0 means point just past the end of the stream, <1 means point at the last record, etc. Relative offsets from the current position are done with the + and - prefixes.

QUERY: the remainder of the commands are various querys to obtain information on a stream. DATETIME gives the date and time that the stream was last modified. EXISTS returns the fully qualified name if the stream exists, FORMAT returns the record format and the logical record *length of the stream, INFO returns format data, size data, and date/time,* LINEPOS READ/WRITE return the current position of the read or write pointer, and SIZE returns the number of lines in the stream.

VM Specific Stream names — REXX I/O

- Reader file - nnnn RDRFILE CMSOBJECTS.
- Punch - VIRTUAL PUNCH CMSOBJECTS.
- Printer - VIRTUAL PRINTER CMSOBJECTS.
- SFS file - filename filetype dirname
- Minidisk or accessed SFS file - filename filetype filemode
  - filemode is optional for an input file, or can be *
  - filemode must be specified for an output file
- Program stack - PROGRAM STACK CMSOBJECTS.
- Default stream - no name specified or name is null
- Or may use the unique ID returned on the OPEN command

GKB

VM Specific Stream names /Notes — REXX I/O

Case is insignificant when specifying the names for the reader, punch, printer or program stack. When specifying a minidisk file, an accessed SFS directory file or an SFS file, the case is significant (thus allowing you to process files with mixed case names). As a point of clarification, a little background is necessary on the names we have choosen. We are working on a standard form of I/O that accesses a variety of data streams. In creating this standard I/O model, we wanted to have names that would totally describe the data stream. That is, if you know the data stream name, you know its characteristics. The stream names used by REXX I/O are part of this model.

Reader - nnnn is the spool file number that you want to process. Specifying an asterisk for nnnn means to use the first file in the reader. Normal rules apply as to reader class and the class of spool files.

SFS file - the directory does not have to be accessed, all you have to do is specify the directory name - "dirname". Wild card characters are not permitted.

minidisk or accessed SFS file - You can omit the file mode if you are working with an input file. If you are working with an output file, you must give the file mode. Wild card characters are not permitted.

Program stack - The default for this is FIFO stacking for output. You can add FIFO or LIFO to the name to explicitly use FIFO or LIFO stacking. The name for LIFO stacking would be "PROGRAM STACK CMSOBJECTS.LIFO". The name for FIFO stacking would be "PROGRAM STACK CMSOBJECTS.FIFO".

default stream - This is the terminal input buffer for input and the users display for output. On input, if there are no lines in the terminal input buffer, a VM READ results. You can omit the name on the I/O functions (except STREAM) to specify the default stream. You can also use a null name on all functions to specify it.

When you use the STREAM function to open a data stream, the returned string on a successful open contains the string READY: followed by a unique ID. This ID can be later used on all I/O function calls in place of the name, and it will speed up processing. An example of obtaining and using the unique ID is as follows:

```
/* show use of the unique ID    GKB 5/93 */
parse value stream('TEST FILE A1','c','open read') with ok id
if ok ¬= 'READY:' then signal open_error
say linein(id)  /* will read and display a line from TEST FILE A1 */
```

GKB

Additional information — REXX I/O

- All I/O is done by calling CSL routines
  - these routines pass back a return code and a reason code on every call
- The STREAM(name,'D') command can be used to get these codes when errors occur
- NOTREADY traps should be used to handle error conditions
  - both SIGNAL ON and CALL ON are supported
  - the CONDITION function can be used to get important information

GKB

Additional information /Notes — REXX I/O

REXX uses calls to CSL routines to do the actual I/O. A return code and a reason code is always passed back from the CSL routine and this information is kept in the data stream control block. Using the STREAM function with a request for a "Description" will return these codes to the user.

It is also a very good idea to have a NOTREADY trap set up in your program to handle the error conditions as they occur. SIGNAL ON NOTREADY and CALL ON NOTREADY are both supported. While in the NOTREADY processing routine, the CONDITION function can be used to retrieve the error string passed back from the I/O routine. This string contains the return and reason codes from the error condition and will show exactly why the error occurred.

GKB

Examples — REXX I/O

```
/* This routine copies the stream or */
/* file named by the first argument  */
/* to the stream or file named by    */
/* the second, as lines.             */

parse arg inname, outname

do while lines(inname)>0
  call lineout outname, linein(inname)
end
```


Examples ... — REXX I/O

```
/* This routine collects characters  */
/* from the stream named by the       */
/* first argument until a line is    */
/* complete, and then places the     */
/* line on the external data queue.  */
/* The second argument is the single */
/* character that identifies the end */
/* of a line.                        */

parse arg inputname, lineendchar

buffer=''  /* initialize accumulator */

do forever
  nextchar=charin(inputname)
  if nextchar=lineendchar then leave
  buffer=buffer || nextchar
end
queue buffer  /* place on data queue */
```


Examples ... — REXX I/O

```
/* Read the first line of the input  */
/* file and get the number of lines  */
/* in the file.  Generate a random   */
/* number from 2 to the number of    */
/* lines in the file and then read   */
/* that line number.                 */

infile = 'SAYINGS SCRIPT A'
parse value stream(infile,'c','open read'),
          with ok handle
if ok ¬= 'READY:'
  then do
         'MSG Error in opening' infile
         'MSG Description string =',
                      stream(infile,'d')
         exit 100
       end

how_many = word(linein(handle,1),1)
num = random(2,how_many)
saying = linein(infile,num)
call lineout(infile)
```


Summary — REXX I/O

- Native REXX language Input/Output
- The general VM I/O model
- Three line based functions
  - a line based variant of the PARSE instruction
- Three character based functions
- A STREAM function for minipulation of a data stream
  - stream states
  - stream commands
- Stream names in VM
- Additional information

REXX—THE FUTURE

MIKE COWLISHAW
IBM

REXX—The Future

Mike Cowlishaw

IBM UK Laboratories
Hursley

13 May 1993

The Future of REXX

- REXX usage today
- Hardware speed and REXX
- REXX Top Ten language requests
- Trends and directions
- Discussion

REXX usage today

- 18 commercial implementations, on most significant platforms
- 41 published books and manuals

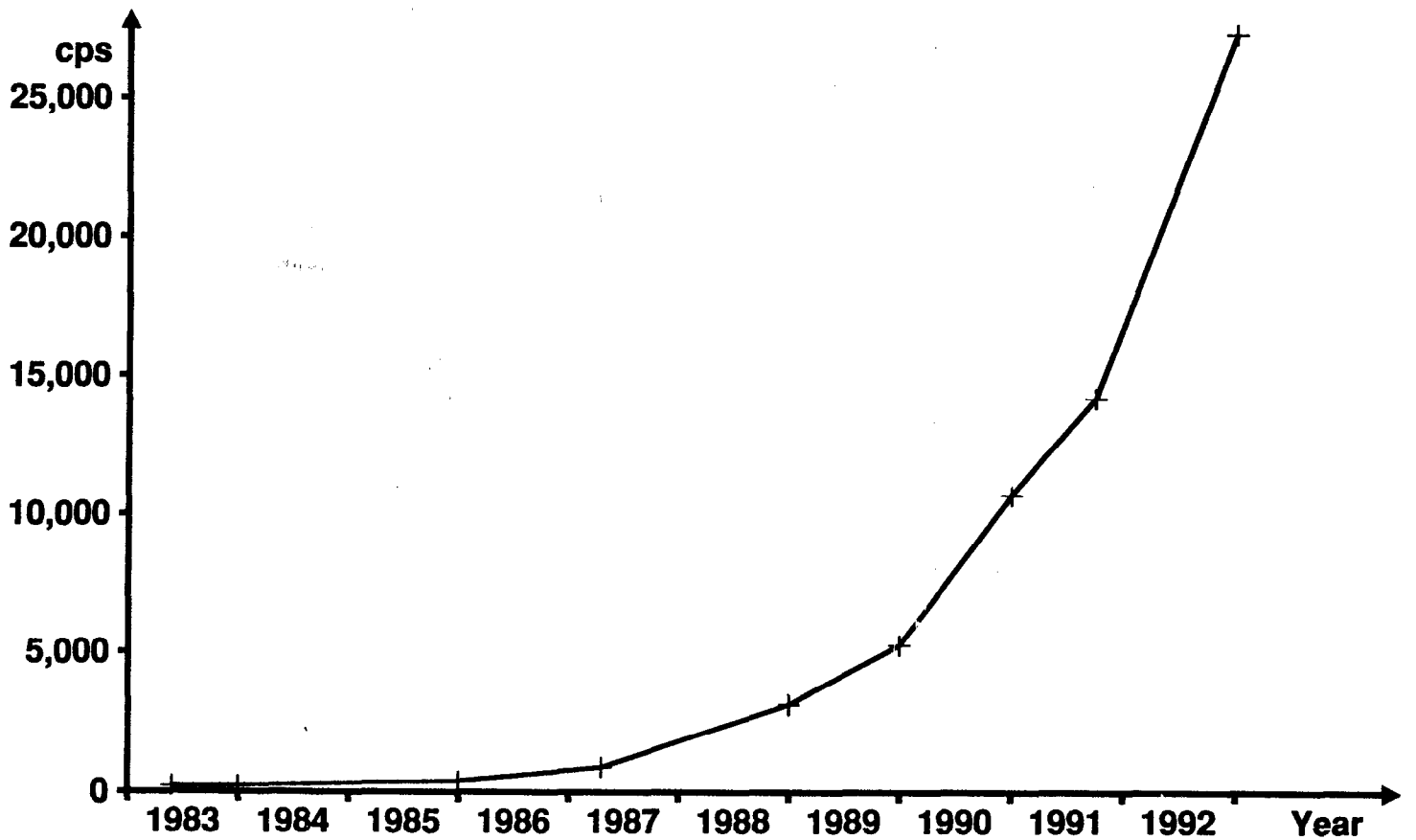
  (Nearly 60, if service guides, second editions, and translations are included.)
- Accessible to well over ten million users
- ANSI (X3J18) standard work well under way.

Hardware speed and REXX

As hardware speed increases, REXX is being used for a wider set of applications. Some informal figures:

- *x86* systems—now over 27,000 REXX clauses per second (486/66)
- RISC systems—over 42,000 REXX cps (same interpreter)
- Mainframe systems—over 90,000 REXX cps
- REXX Compiler/370—up to 465,000 REXX cps

REXX Clauses per Second—x86 platform



Top Ten language improvements

Caveats...

- This is a personal list, and does not describe any vendor's product plans (as far as I know)
- Object-Oriented extensions are not included (though these would probably be a superset of this list)
- Don't read too much into the order.

10. DIGITS condition

Allows the trapping of unexpectedly "overprecise" numeric data:

```
numeric digits 5
signal on digits
arg a
say a+1
```

... would raise the condition if the value of A had more than five digits.

9. Expressions in stem references

Today's idiom:

```
nextj=j+1
nextk=k+1
say fred.nextj.nextk
```

16

...could be written as...

```
say fred.[j+1, k+1]
```

or, perhaps

```
say fred.(j+1).(k+1)
```

7 & 8. PARSE enhancements

Two improvements:

```
parse caseless .....
```

...like `parse`, but strings will match, even if they have a different mix of uppercase and lowercase

```
parse lower .....
```

...like `parse upper`, but translates to lower case first

6. Variable CALL target

An "indirect" call:

```
where='anyname'
call (where) a, b
```

...would call the routine "anyname".

5. Change and count functions

```
needle='is'
haystack='This is the third'
new='at'

say countstr(needle, haystack)
say changestr(needle, haystack, new)
```

...would display '2' and 'That at the third'.

4. Call by Reference

Introduces aliasing to the language:

```
call fred p, q+1, r
     .
     .
fred: procedure
  use alias a, ,c
```

or

```
fred: procedure
  use arg (a), b, (c)
```

3 External Procedure Expose

Allow "externalization" and sharing of many more routines, by permitting `procedure expose...` at the start of external routines.

Inheritance of NUMERIC DIGITS and other appropriate settings must be implied by this use.

21

2. Extended DO

Iterate over the tails of a compound variable:

```
do tail over fred.
  say fred.tail
  end tail
```

22

...would display all the values held in variables whose names begin with "FRED."

Changing any FRED.xxx variable while in the loop would be an error.

1. Date and time conversions

Almost every programmer needs these at some time...

```
say date('usa', 19930827, 'standard')
```

...would display `'08/27/93'`.

23 For conversions from years with only two digits to those with four digits, the result nearest to the current year would be used, taking into account only the year and using the earlier date if a tie.

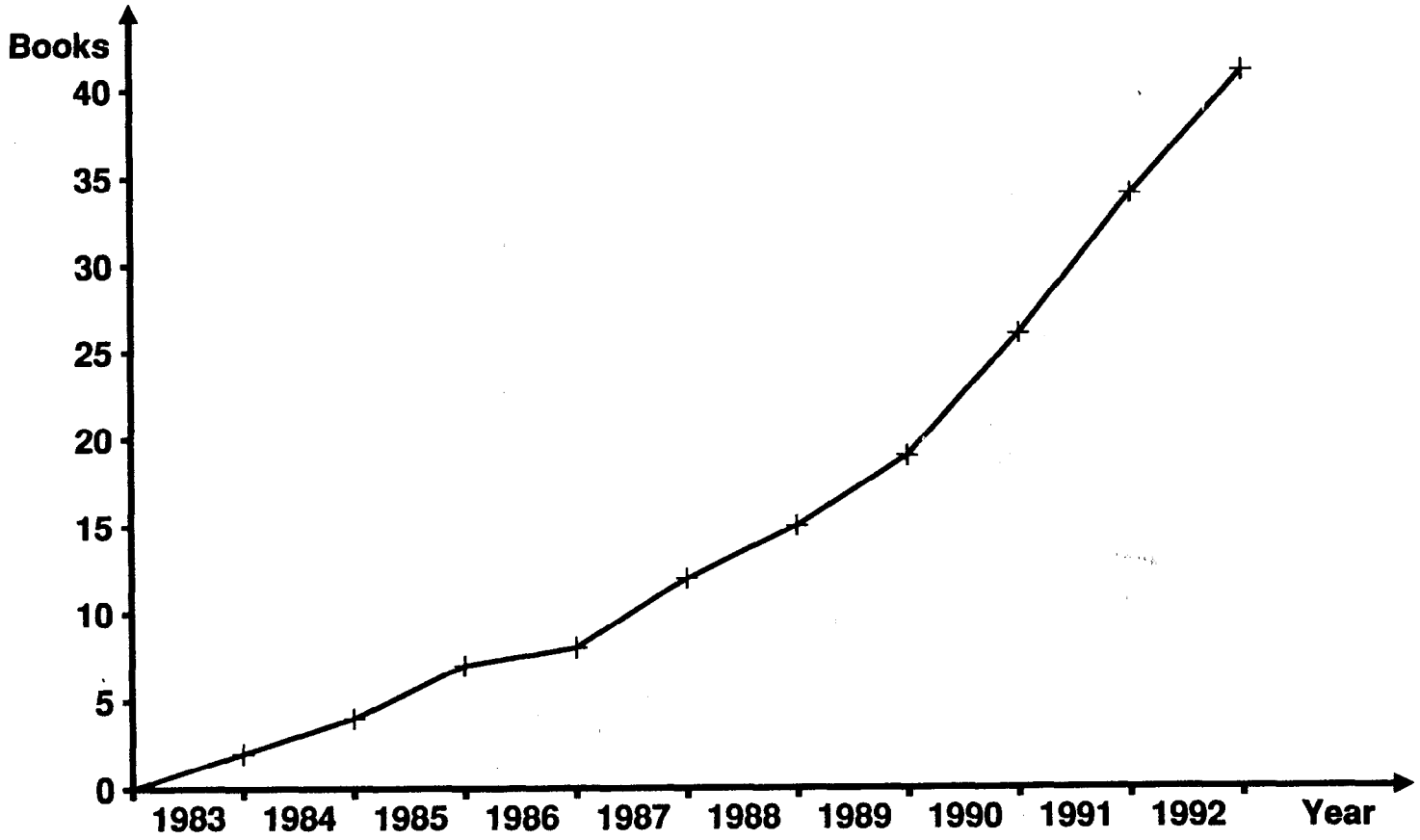
...and similarly for the TIME built-in function.

Trends and directions—Applications

- Mainframe interactive applications continue to move to the desktop
- Networking of workstations and PCs encourages standardization of applications and languages
- Increasing complexity and sophistication of applications leads users to demand extensive subsetting and customization
- Object-oriented prototype shows that REXX will be a first-class object-oriented language.

24

REXX Books and Manuals



The Future of REXX

- REXX usage today
- REXX assets
- Trends and directions
- Discussion

Which REXX assets are the most important?

- Simplicity:
  - A small, readable, language
  - Just one data type—the string
  - Decimal arithmetic
  - Few limits

More assets...

- Flexible and extendible
  - Existing and future system interfaces
  - Object-oriented extensions fit naturally

28

More assets...

- *Designed* as a multi-purpose extension language
  - Highly system and hardware independent
  - Keywords reserved only in context, so macros in source form are resistant to breakage
  - Adds value to almost all platforms and applications
- Skills reuse between platforms
  - Reduced education costs.

REXX Extensions for OS/2

Charles Daney
Quercus

A General-Purpose REXX Extension Package

REXX Symposium
May, 1993

Charles Daney
Quercus Systems
P. O. Box 2157
Saratoga, CA 95070

(408) 867-7399
Fax: (408) 867-7489
BBS: (408) 867-7488
CompuServe: 75300,2450
Internet: 75300.2450@compuserve.com

Motivation

- Make handling of various kinds of aggregates easier
  - lists
  - sets
  - collections
  - ordered pairs
  - ordered triples
- Allow easier simulation of data structures
- Treat compound variables or groups of variables as a "database"
  - defaults & user program configuration
  - representation of program state
  - externalize setup of data tables
  - "true" databases (e. g. user directory)
- Facilitate exchange of data between programs
- Faster loading of data

Functions for compound variable and array handling

- Array handling
  - copying from one array to another (overlay)
  - copying from one array to another (insert)
  - deleting portions of an array
  - sorting portions of an array
- Compound variable handling
  - copy all elements
  - save all elements to disk
  - restore all elements from disk
  - find all tails (with optional pattern match)
  - find all values (with pattern match)
- Groups of variables
  - write group to disk
  - read group from disk
  - dump group for debugging

"Quasistems"

- REXX language says that stems contain only final period
- REXX users use compound names hierarchically
  pet.katie.rabbit = 'Flopsy'
  pet.katie.cat = 'Fluffy'
  pet.katie.dog = 'Fido'
- Users expect "drop pet.katie." to behave like a stem
  (without affecting pet.lisa, pet.susy, etc.)
- Similarly for other operations like copy, read, write
- When used carefully, this seems useful and internally consistent

Conventions of library functions

- Quasistems allowed in context where stem is expected
- Final period is assumed in a stem context
- Substitution is not performed on quasistem components
- Case is significant in all but first component of quasistem

Array conventions

- Common convention is that arrays are integrally subscripted
- First array element is 1
- Zeroth element is number of array elements
- The array stem may be a quasistem

Array operations

- Copy - ARRAYCOPY
  - from-position, to-position, count are options
  - elements of target array are overlaid
- Insert - ARRAYINSERT
  - from-position, to-position, count are options
  - elements of target after insertion point move
- Delete - ARRAYDELETE
  - from-position, count are options
  - remaining elements shift position to eliminate gap
- Sort - ARRAYSORT
  - from-position, count are options
  - start, length, order, type specified for each field

Compound variable operations

- Copy - CVCOPY
  - Makes exact copy of a compound variable with different stem
  - Target is dropped first
- Write to file - CVWRITE
  - Existing file is erased
  - File contains tails only, not "stem"
- Read from file - CVREAD
  - Target is dropped first
  - File contains tails only, not "stem"
- List compound variable tails - CVTAILS
  - Creates an array with all tails
  - Enables iteration on all tails
  - Regular expression pattern match is optional
  - Case sensitivity is optional in pattern match
- Search compound variable values - CVSEARCH
  - Creates an array with tails of matches
  - Matching by regular expression
  - Case sensitivity is optional

Regular expressions

- \ - escapes special characters
- ^ - matches beginning of string
- $ - matches end of string
- . - matches anything but newline
- :a - matches alphabetic character
- :d - matches digits 0 - 9
- :n - matches alphabetics & digits
- * - matches 0 or more of expression
- + - matches 1 or more of expression
- ? - matches exactly 0 or 1 or expression
- [ ] - list of matching characters

Regular expression examples

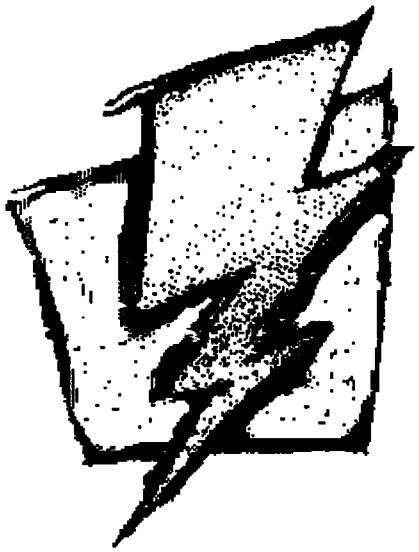
- fido - matches string "fido" anywhere
- ^fido$ - matches only "fido" by itself
- [01234567] - matches only valid octal characters
- [01234567]* - matches string of valid octal characters
- [^ABC] - matches anything but A, B, C

Variable group operations

- Write to file - VARWRITE
  - Existing file is appended
  - All variables in program may be written
  - Optional list of variables & stems to include or exclude
  - Stems are kept in the file
- Read from file - VARREAD
  - Existing stems aren't dropped - data is merged
  - All variables in file may be read
  - Optional list of variables & stems to include or exclude
- Dump variables - VARDUMP
  - Like VARWRITE except output is formatted for visual inspection

WATCOM VX REXX FOR OS/2

ERIC GIGUERE
Waterloo



WATCOM VX REXX

The Complete REXX Programming Environment for OS/2

Eric Giguère
WATCOM International
giguere@csg.uwaterloo.ca

The REXX language provides no facilities for user interaction other than through a simple console to which a program can read and write lines of text. On an advanced system like OS/2, these facilities seem primitive in comparison to those offered by the Presentation Manager (PM). VX REXX breaches this gap to let you develop REXX-based PM applications.

Features

WATCOM VX REXX is a complete REXX development environment with the following features:

- **Project management facilities.** A *project* is the REXX code and the Presentation Manager user interface that together form an application. VX REXX allows you to work on the project in small, individual pieces without limiting your ability to view and edit the complete REXX program.

- **Direct user interface design and editing.** VX REXX lets you directly design your program's user interface using *objects* based on standard Presentation Manager windows and controls. You create, size and position the objects and modify their settings as you would with the Workplace Shell objects on your desktop.

- **Testing and debugging.** Test your application directly within the VX REXX environment. If an error occurs, VX REXX will show you exactly where it happened. Use the VX REXX symbolic debugger to track errors in program logic.

- **Useful REXX extensions.** VX REXX includes sets of functions for displaying standard dialogs, performing common file operations, and creating and manipulating objects.

- **Flexibility and extendability.** VX REXX can generate standalone executables or macros for use with other applications. Your programs can use third-party REXX function packages and other REXX extensions that follow the application programming interface (API) defined by OS/2. New object types can also be added to VX REXX in C using SOM.

WATCOM VX REXX works with the standard OS/2 2.0 REXX interpreter.

A Simple Example

Figure 1 illustrates the VX REXX development environment. A few simple steps are all you need to write your first VX REXX application:

1. Select the PushButton item from the Tools menu or click on the appropriate icon in the tool palette. Move over the gridded window and press the left mouse button to position and size a new PushButton object.

2. Press the right mouse button on the PushButton object to bring up its popup menu. Choose the Properties item to bring up the PushButton's properties notebook.

3. Set the Caption property to the string "Push Me!" by selecting the Text tab of the notebook and typing the string into the Caption entry field. (See Figure 2.)

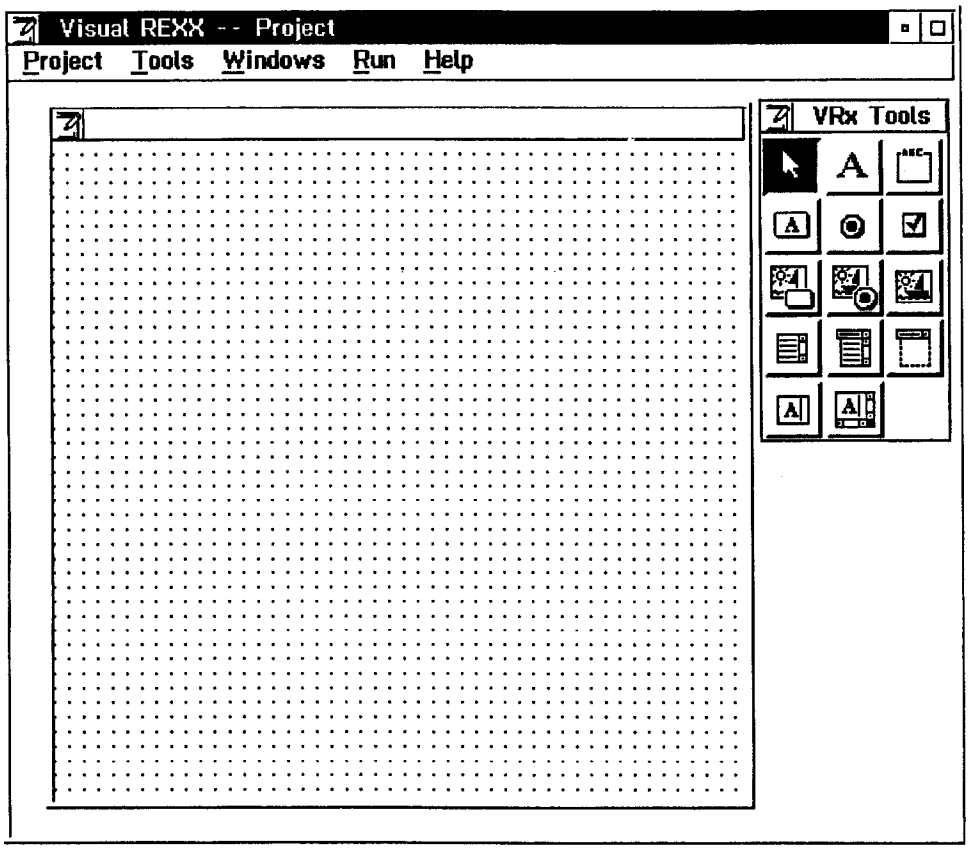


Figure 1: The VX REXX development environment.

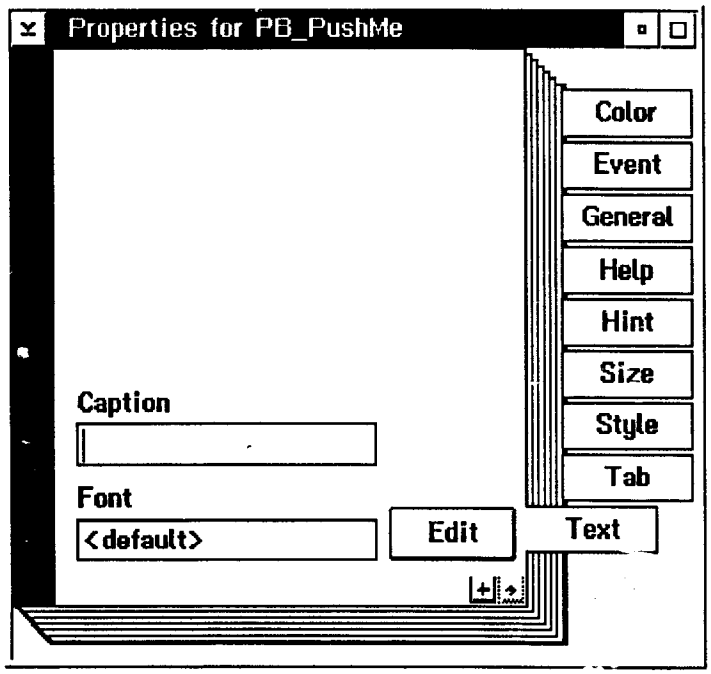


Figure 2: Changing the caption of a push button object.

4. Select the Event tab and double click on the Click item in the listbox. A text editor will appear with the skeleton for a REXX procedure. Type `say "You pushed me!"` after the procedure label and close the text editor window.

Your application is now complete. To test it, select the Run Project item from the Run menu. Press the push button several times. VX REXX will automatically open a console window when the first `say` insruction is executed. When you are finished, simply close the window to halt the REXX program.

When you are satisfied with your application, select the Make EXE item from the Project menu to build a standalone executable.

For More Information

For more information on VX REXX, contact WATCOM at:

WATCOM
415 Phillip Street
Waterloo, Ontario
CANADA N2L 3X2

Phone: (519) 886-3700
Fax: (519) 747-4971
BBS: (519) 884-2103

EXPLOITING VM/CMS REXX WITH WATERLOO C

ERIC GIGUERE AND DOUG MULHOLLAND
Waterloo

Exploiting VM/CMS REXX Facilities

with Waterloo C

Doug Mulholland

University of Waterloo

Internet: dwm@csg.uwaterloo.ca
Voice: (519) 888-4676

Overview

Run-time Library Facilities

- REXX function and variable access
- XEDIT command and file buffer access
- CMS SUBCOM support

Debugger Support

- C debugger macros

C Compiler (Preprocessor) Extension

- compile-time REXX interface

C Calls to REXX

ANSI Standard Interface

- system() – C program calls <fname> EXEC

  int system(char *cmdstr);

  Example:

  system("myexec command-line text");

  - parse arg
  - exit return_code

Waterloo C Additions: <csubcom.h>, <rexxdef.h>

- execft() – call a REXX macro with an initial default address environment:

  int execft(char *cmdstr, char *ftype);

  Example:

  execft("PROFILE command-line text", "MYAPPL");

- execrexx() – call a REXX program in memory

  int execrexx(char *cmdstr, char *ftype,
  size_t stmtcount, char **rexxstrs);

Example: execrexx()

```
#include <csubcom.h>
#include <stdio.h>
#include <string.h>

static char *rxpgm[] =
  {
    "/* INMEM REXX */",
    "say 'INMEM REXX: address() =' address()",
    "parse arg cmdline",
    "say ' cmdline = \"'cmdline'\"'",
    "parse source srcline",
    "say ' srcline = \"'srcline'\"'",
    "exit length( cmdline )"
  };

#define CMDLINE "Typical command line"

int main()
  {
    int stmtnum, retval;
    char *rxcmdline;

    /* print the REXX program, then execute it... */
    printf( "char *rxpgm[]...\n" );
    for( stmtnum = 1; stmtnum <= 7; ++stmtnum )
        printf( " %d: %s\n", stmtnum, rxpgm[ stmtnum - 1 ] );

    printf( "CMDLINE = \"%s\"\n strlen( CMDLINE ) = %d\n",
        CMDLINE, strlen( CMDLINE ) );
    rxcmdline = "INMEM " CMDLINE;
    retval = execrexx( rxcmdline, "REXX", 7, rxpgm );
    printf( "execrexx( rxcmdline, \"REXX\", 7, rxpgm ) = %d\n",
        retval );
  }
```

Example: execrexx() ...continued

Output:

```
char *rxpgm[]...
 1: /* INMEM REXX */
 2: say 'INMEM REXX: address() =' address()
 3: parse arg cmdline
 4: say ' cmdline = "'cmdline'"'
 5: parse source srcline
 6: say ' srcline = "'srcline'"'
 7: exit length( cmdline )
CMDLINE = "Typical command line"
 strlen( CMDLINE ) = 20
INMEM REXX: address() = REXX
 cmdline = "TYPICAL COMMAND LINE"
 srcline = "CMS COMMAND INMEM REXX * !INMEM REXX"
execrexx( rxcmdline, "REXX", 7, rxpgm ) = 20
```

REXX Variable Access

Stem Variable Access: <stdio.h>, <file.h>

```
fopen( "_REXX.MYVAR.", "r+" )
open( "_REXX.MYVAR.", O_WRONLY|O_TRUNC|O_CREAT )
```

- compatible with EXECIO (STEM
- MYVAR.0 contains the number of lines (n)
- MYVAR.n contains the "file" data
- all the usual I/O functions: fprintf(), fscanf(), read(), write(), ...
- supports direct access: fseek(), ftell()
- command line redirection of standard I/O

Single Variable Access: <rexxdef.h>

- C library functions to set, get and drop a REXX variable's value

```
int rexxset( char *var, char *value );
int rexxfetch( char *var, char *bufptr, size_t buflen );
int rexxdrop( char *var );
```

Program Stack Access

I/O using <stdio.h>, <file.h>

```
fopen( "_STACK.FIFO", "w" )
open( "_STK", O_WRONLY|O_TRUNC|O_CREAT )
```

- FIFO, LIFO (the default)
- compatible with REXX push, queue, parse pull
- all the usual I/O functions: fprintf(), fscanf(), read(), write(), ...
- direct access using fseek(), ftell() is diagnosed as an error
- command line redirection of standard I/O

REXX Calls to C

"Traditional" Calls to C Programs

```
/* */
'MYPGM Typical C program command-line'
say 'program return code:' rc
```

- various forms of argc/argv command line processing:
  - UNTOKENIZED: argc is 1, argv[0] is the entire command line
  - TOKENS: usual CMS tokenization (8 character, upper case tokens)
  - EXTEND: tokenization (without truncation)
  - C_STRINGS (UNIX compatible): quoted strings preserved as one token, trigraphs and '\' processed as for C strings
- integer return value from main() assigned to REXX rc

Function Calls to C Programs

```
/* */
rexx_string = mypgm( "parm 1", "parm 2" );
say 'mypgm returned:' rexx_string
```

- C program can dynamically detect environment and (optionally) exit with a string return value

```
int isrexxfn( void );
void rexxit( char *str );
```

XEDIT Access

XEDIT Subcommands

xedit(char *str);

Example

```
xedit( "MSG Bye, bye file!" );
xedit( "ALL\\:1 DEL *" );
```

XEDIT File Buffer Access

- <depsets.h>

  int setxedit(int enable_xedit);

  0 disables access, 1 enables access

- <stdio.h>, fopen(), ...

  when a file in the XEDIT file buffer ring is opened, I/O is performed through the XEDIT SUBCOM

- no CMS minidisk/SFS access required

Example Usage

- compiler can compile from an XEDIT file buffer, write errors to a second file buffer, and display a message in the editor message area
- when run from XEDIT, DIFF can compare the disk version of a file with the current file buffer contents
- CALC (a calculator program) can append its output to an editor file buffer

CMS SUBCOM Support

<csubcom.h>

- C program defines a subcommand handler function: subcomset()
- program calls REXX (or just another program component), usually with execft
- handler is called with (argc/argv) command line parameters
- run-time environment recovered by SUBCOM handling interface (signal handling, open files, etc.)
- return code from handler returned to REXX
- handler function removed from SUBCOM list with subcomclr()
- can be used to transfer data between programs

Example: subcomset(), subcomclr(), execft()

```
#include <csubcom.h>
#include <stdio.h>
#include <string.h>

static int schandler( int argc, char **argv );

int main()
  {
    csubcom_descr *csdptr;
    char *cmdline, *ftype;
    int retval;

    csdptr = subcomset( schandler, "SUBC1" );
    cmdline = "C2REXX2C";
    ftype = "SUBCSET";
    printf( "main: calling execft( \"%s\", \"%s\" )\n",
          cmdline, ftype );
    retval = execft( cmdline, ftype );
    printf( "main: retval = %d\n", retval );
    subcomclr( csdptr );
  }

static int schandler( int argc, char **argv )
  {
    int i;

    printf( "schandler() : argc = %d\n", argc );
    for( i = 0; i < argc; ++i )
          printf( " argv[ %d ] = \"%s\"\n", i, argv[ i ] );
    return( argc );
  }
```

Example: subcomset(), subcomclr(), execft() ...continued

```
/* C2REXX2C SUBCSET -- demonstrate C calling REXX calling C */

address SUBC1
say 'C2REXX2C SUBCSET: calling SUBC1...'
'Alphanumeric, mixed-case subcommand line'
say 'C2REXX2C: rc =' rc
exit 1
```

Output:

```
main: calling execft( "C2REXX2C", "SUBCSET" )
C2REXX2C SUBCSET: calling SUBC1...
schandler() : argc = 5
  argv[ 0 ] = "SUBC1"
  argv[ 1 ] = "Alphanumeric,"
  argv[ 2 ] = "mixed-case"
  argv[ 3 ] = "subcommand"
  argv[ 4 ] = "line"
C2REXX2C: rc = 5
main: retval = 1
```

C Debugger Support for REXX

CDEBUG Provides:

- a CMS SUBCOM entry point for REXX macros to call
- PROFILE CDEBUG, (NO)PROFILE <fname> command-line option
- EXTRACT subcommand for accessing debugger internal data
- OPTION OUTPUT <file-name> lets command output be written to a file, including _REXX.<stemmed-var>

Example:

```
/* CALC CDEBUG */
parse arg cmdline
address command 'MAKEBUF'
bufnum = rc
address cms 'CALC' cmdline '>_stk.fifo'
calcrc = rc
if calcrc = 0 then do queued()
      parse pull calcout
      'MSG' calcout
   end
address command 'DROPBUF' bufnum
if calcrc <> 0 then
   __"EMSG Return code '"calcrc"' from 'CALC'."
exit 0
```

Compile-time REXX Support

An Experimental Facility!

- compiler recognizes a builtin C preprocessor symbol to call REXX ("external") functions:

  __EXT__(FUNCNAME)(text of REXX argument string)

- FUNCNAME EXEC is called as a REXX function, string return value is inserted into C source stream
- compiler provides a SUBCOM entry point for accessing internal (symbol table) data

Compile-time REXX Support ...continued

Example:

```
#include <stdio.h>

int main()
   {
      printf( "hello, world\n" );
      __EXT__(SYSCMD)( CP Q TIME )
      return( 0 );
   }


/* SYSCMD EXEC - execute a system command */
parse arg cmdline
say 'SYSCMD:' cmdline
address cms cmdline
exit ""
```

Output:

```
Ready;
cw crexx
Waterloo C (Version 3.3B IBM 370)
SYSCMD. CP Q TIME
TIME IS 13:49:50 EDT TUESDAY 05/11/93
CONNECT= 00:32:24 VIRTCPU= 000:39.60 TOTCPU= 000:53.52
File 'crexx c a1': 8 lines, included 130, no errors
Ready;
```

- it's no worse than what the SQL preprocessor strips out, and certainly better than what the C preprocessor can do!

Compile-time REXX Support – Applications

So What Can We Do With It?

- access compiler data: the compiler SUBCOM entry point supports "TYPEOF" and "SYMLOOK" directives (extract symbol table data)
- call MAKE to do other updating operations (enforce updatedness)
- insert source code into the program: the REXX function's return value is "compiled"

  Example:

```
__EXT__(SRC1)()          /* HELLO C */


/* SRC1 EXEC */
src =       '/* HELLO C -- A First Program. */'
src = src 'int main()'
src = src ' {'
src = src '   printf( "hello, world\n" );'
src = src '   return( 0 );'
src = src ' }'
exit src
```

Compile-time REXX Support – Applications

- evaluate non-constant expressions

  Example:

```
#define SIN _ _EXT_ _(CWSIN)

const double sin_of_5 = SIN(5);


/* CWSIN EXEC */
parse arg sinarg
address cms 'CALC sin(' sinarg ') >_REXX.CALCOUT.'
parse var calcout.1 skip1 '=' cwsinval '(' skip2
exit cwsinval
```

- retrieve C source code from somewhere (e.g., network, database, ...), "pre-preprocess" it, then compile it

Summary

Programs and Applications

- in UNIX environments, pipes and shell programs provide the "glue" to combine programs into "applications"
- for CMS, and now MVS and OS/2, REXX lets application programmers (and even end-users) combine programs and customize applications
  - transfer program control between programs within an application
  - transfer data between programs within an application

REXX Bits

Linda Green
IBM

REXXbits

Linda Suskind Green
REXX Interface Owner

IBM
Endicott Programming Lab
G98/6C12
PO Box 6
Endicott, NY 13760

INTERNET: greenls@gdlvm7.vnet.ibm.com
Phone: 607-752-1172

May, 1993

Contents

• REXX History

• REXX Excitements

• REXX Curiosities

Contents

- **REXXbits Summary**

REXX History

REX becomes REXX

In the beginning, there was

REX (REformed eXecutor)

which eventually became

REXX (REstructured eXtended eXecutor)

REXX Firsts

- 1979 - Mike Cowlishaw (MFC) starts work on REX
- 1981 - First SHARE presentation on REX by Mike
- 1982 - First non-IBM location to get REX is SLAC
- 1983 - First REXX interpreter shipped by IBM for VM
- 1985 - First non-IBM implementation of REXX shipped
- 1985 - First REXX trade press book published
- 1987 - IBM Selects REXX as the SAA Procedures Language
- 1989 - First REXX compiler shipped by IBM for VM
- 1990 - SHARE REXX committee becomes a project
- 1990 - First SHARE presentation on Object Oriented REXX
- 1990 - First Annual REXX symposium held (organized by SLACs Cathie Dager)
- 1991 - First REXX ANSI committee meeting held

Jeopardy: REXX for $1000

Answer is:

19

Question is:

What are the number of official members of X3J18 (ANSI REXX committee)?

Jeopardy: REXX for $800

Answer is:

118

Question is:

How many people attended the first annual REXX symposium in 1990 (as listed in the proceedings)?

Jeopardy: REXX for $600

Answer is:

203

Question is:

What is the number of pages in the second edition of TRL (The REXX Language) book by Mike Cowlishaw?

Jeopardy: REXX for $400

Answer is:

646

Question is:

What are the number of pages in TRH (The REXX Handbook) written by many people in this room?

Jeopardy: REXX for $200

Answer is:

4794

Question is:

How many days has it been since REXX was started on March 20, 1979? (13 years, 45 days)

Jeopardy: REXX for $500

Answer is:

5

Question is:

How many programming languages has MFC designed?
Note that REXX is his latest!!!!

Jeopardy: REXX for $400

Answer is:

350

Question is:

What is the peak amount of REXX electronic mail MFC received per working day?

Jeopardy: REXX for $300

Answer is:

4000

Question is:

What is the approximate number of hours MFC spent on REXX before the first product shipped?

Jeopardy: REXX for $200

Answer is:

500,000

Question is:

What are the approximate number of REXX related electronic mail MFC has read since REXX started?

Jeopardy: REXX for $100

Answer is:

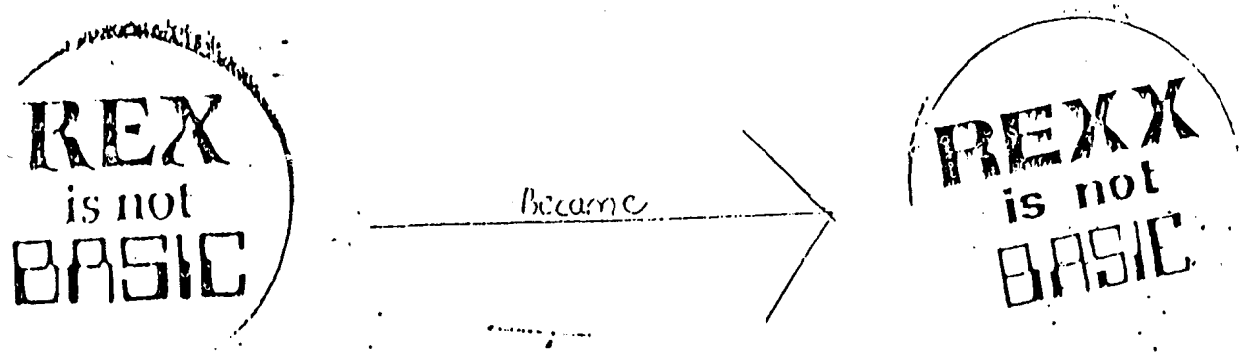
over 6,000,000

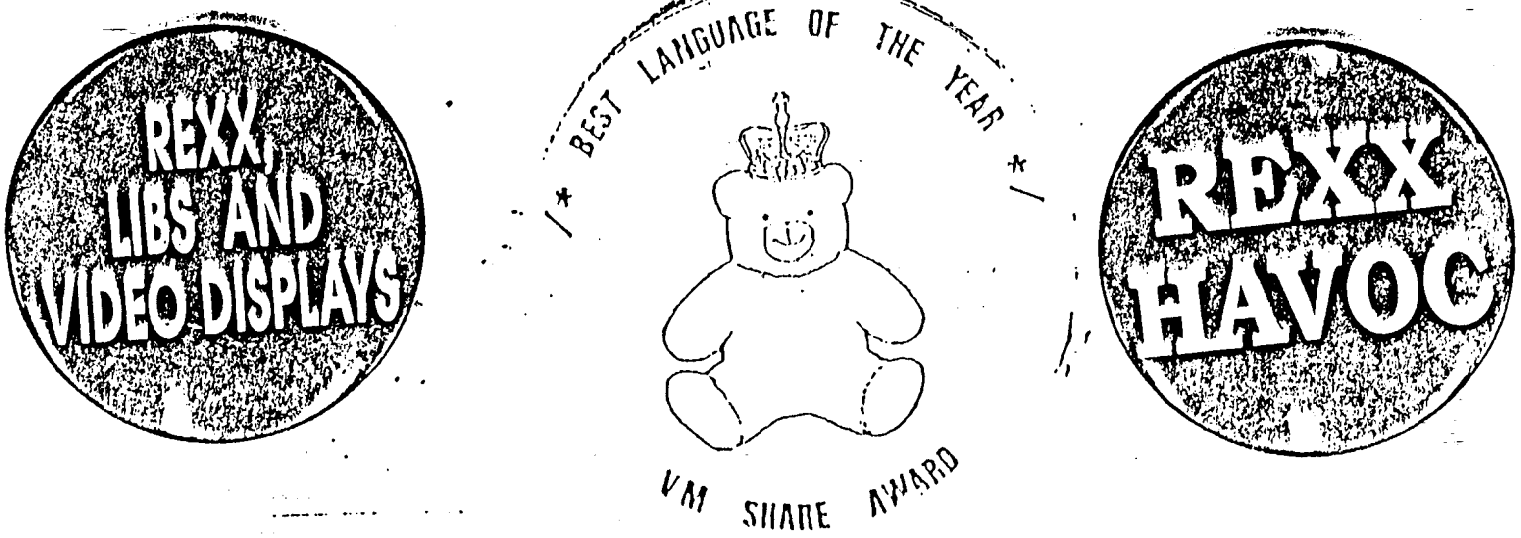
Question is:

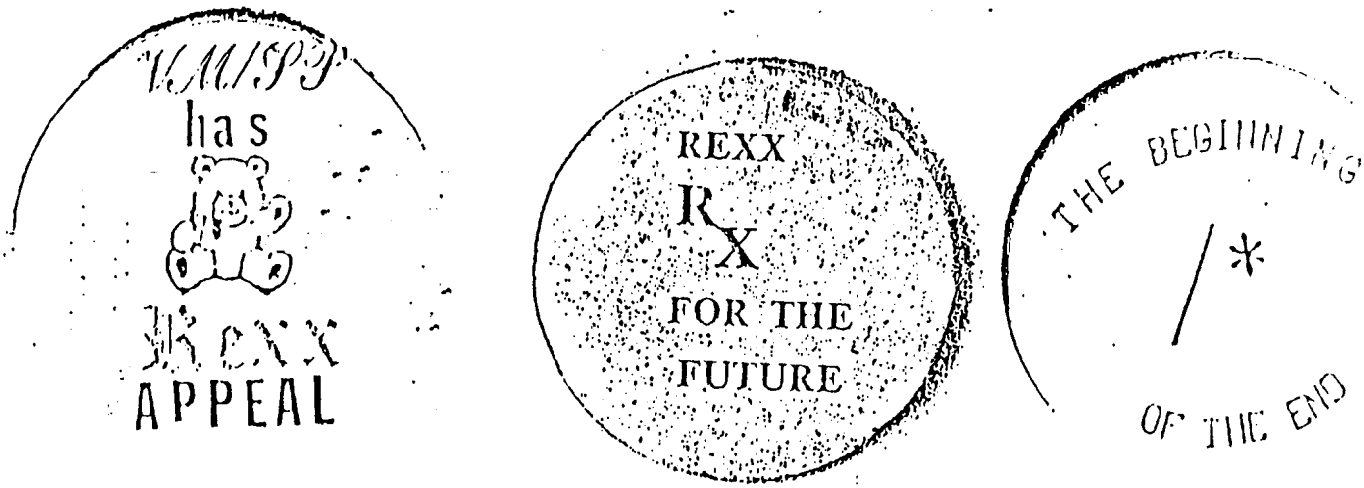
What is the largest known total number of lines of REXX code used in any one company?

REXX Buttons

Customer Created:







REXX Buttons

Customer Created:

REXX Buttons ...

IBM Created:

SAA Procedures
Language/REXX

Text of the REXX Buttons

- General
  - REX is not BASIC
  - REXX is not BASIC
  - The beginning /* of the end
  - REXX RX for the future
  - I've been REXX'd and I like it
  - REXX is not a ...
  - REXX Havoc
  - REXX, Libs and Video Displays
  - REXX Therapist
- VM
  - /* Best Language of the Year */ VM SHARE AWARD
  - VM/SP has REXX Appeal
  - RXSQL good medicine!
  - Programming Power-CUA 2001-REXX
- SAA
  - SAA Procedures Language/REXX
- TSO/E
  - I practice safe REXX (TSO/E v2)
  - TSO/E is REXX rated!
  - Tyrannosaurus REXX TSO/E v2
  - TSO/E Puttin' on the REXX

REXX Excitements

REXX Excitement!

- ANSI committee started
- REXX Users start a yearly REXX Symposium in 1990
- SHARE elevated REXX to a Project
- Increasing number of books and articles on REXX
- Increasing number of REXX Implementations on different platforms by increasing number of companies

ANSI

REXX is one of 15 languages to be worked on as an ANSI standardized language. Others are:

- APL
- APT
- BASIC
- C
- C++
- COBOL
- DATABUS
- DIBOL
- FORTH
- FORTRAN
- LISP
- PASCAL
- PL/I
- PROLOG

Note that the languages listed are at different levels of standardization.

REXX Symposium

- Annual event started in 1990
- Run by Users of REXX
- Attended by all vendors of REXX implementations and their users
- includes presentations, demos, panel discussions, etc
- Initiated by Cathie Dager of SLAC in 1990
- 1990: 118 attendees for a single day
  1991: expanded to 2 days
  1993: planned for May 18-20, 1992 in San Diego, CA
- Purpose: "a gathering where REXX users and developers could meet each other, exchange ideas, and information about the language and discuss future plans."

SHARE Interest in REXX

SHARE Meeting	Number of REXX Sessions
72 (3/89)	9
73 (8/89)	13
74 (3/90)	23
74.5 (5/90)	REXX Project approved
75 (8/90)	25
76 (3/91)	20
77 (8/91)	28
78 (3/92)	41

Note that the sessions are in the REXX Project, MVS Project, and CMS Project.

Publications

As of 12/90, REXX has been the subject of:

- 4 books (plus 4 in the works)
- 40 User Group Presentations
- 40 product manuals
- 40 articles

REXX Books as of 5/93

Published:

- The REXX Language, A Practical Approach to Programming by Mike Cowlishaw (1985, 1990)
- Modern Programming Using REXX by Bob O'Hara and Dave Gomberg (1985,1988)
- REXX in the TSO Environment by Gabriel F. Gargiulo (1990)
- Practical Usage of REXX by Anthony Rudd (1990)
- Using ARexx on the Amiga by Chris Zamara and Nick Sullivan (1991)
- Amiga Programmers Guide to AREXX by Eric Giguere (1991)
- REXX Handbook edited by Gabe Goldberg and Phil Smith (1992)
- Programming in REXX by Charles Daney (1992)
- The AREXX Cookbook by Merrill Callaway (1992)
- REXX Tools and Techniques by Barry Nirmal (1993)
- REXX: Advanced Techniques for Programmers by Peter Kiesel (1993)

Planned:

- others known to be in the works

REXX is International

REXX books and manuals have been translated into many languages, including:

- Chinese
- French
- German
- Japanese
- Portuguese
- Spanish

REXX is International - Part 2

REXX presentations have been given in the following countries:

- Austria
- Australia
- Belgium
- Canada
- England
- France
- Germany
- Holland
- Ireland
- Japan
- Jersey
- Scotland
- Spain
- United States
- Wales

As of 1982, MFC had received mail from over 30 countries!

REXX Trade Press Article Titles

REXX: A MODERN DAY KING

Comments, Labels, and System Commands

Developers Take Sides in the Continuing Battle of REXX vs. BASIC

REXX, BASIC Vie for Macro Standard

WordBASIC Eases Macro Creation; Personal REXX Shines at String Handling

WEEK 3/23 PRODUCT REVIEW

Session 9D1

Introducing REXX Into The Engineering Curriculum

ARexx is for Writing Applications, Too

rexx: Not Just a Wonder Dog

Halfway between a batch interpreter and a full-blown language, Rexx can quickly integrate applications

REXX Trade Press Article Titles

The Next Step In REXX Programming

Extending ARexx

With the help of a few libraries, you can build menus and requesters for your ARexx programs.

Best Language — REXX: A New King

The Best of Both Worlds:

CLISTs and REXX in a TSO/ISPF Environment

ARexx...for

ARexx is for everyone. . .not just programmers.

Everyone.

Here's an easy course on how to put the power of ARexx in your hand

REXX Trade Press Article Titles

EXOTIC LANGUAGE OF THE MONTH CLUB

REXX: A beginner's alternative

All Hail REXX

or what you should know about IBM's soon to be ubiquitous procedural language

For Programmers

REXX — A Multifaceted Tool

REXX: REXX the Wonder Language

WHY IS REXX SO POPULAR?

REXX—Portrait of a New Procedures Language

Capture cross-system capabilities with REXX

REXX Trade Press Article Titles

ANALYSIS

Lotus Makes a Case for REXX To Foil Microsoft's BASIC Plan

Analysis \ *Powerful REXX vs. Popular BASIC*

Rexx in Charge

You're not really multitasking in OS/2 unless you're using Rexx

HELLO, REXX— GOODBYE, .BAT

REXX is a powerful new shell language that's as easy to use as BASIC. It may replace the DOS batch processor in the future.

REXX

A USER'S DREAM IN A SYSTEM PROGRAMMER'S WORLD

REXX Trade Press Article Titles

Utilize The Power Of REXX/CP/CMS

TEST DRIVE

AREXX

(The integrative language)

A language adopted by IBM comes to the Amiga – and has started to change many other applications.

REXX

The system architecture application connection

REXX Language Level

REXX Level	Usage
3.20	CMS release 3
3.40	CMS release 4, 5, 5.5 MUSIC/SP version 2.2
3.45	CMS release 6 TSO/E version 2, release 1
3.46	CMS release 6 with SPE TSO/E ver 2, rel 1 with APAR 370 compiler SAA Procedures Language level 1

REXX Language Level ...

3.48	OS/400 rel 1.3 VM/ESA rel 1.2.0 TSO/E ver 2.4 SAA REXX/370 compiler rel 2
3.50	Cowlishaw 1985 book Portable REXX ver 1.05 (DOS) uniREXX (UNIX, AIX) Personal REXX version 2.0 (OS/2, DOS) AREXX (Amiga) TREXX (Tandem) Open REXX (DOS, OS/2 MVS, VMS)
4.00	OS/2 release 1.3, 2.0 Cowlishaw 1990 book SAA Procedures Language level 2 Personal REXX version 3.0 (WINDOWS, OS/2, DOS) Portable REXX ver 1.10 (DOS) REXX/Windows

Implementations

There are priced REXX implementations for:

- AIX
- Amiga (interpreter/compiler)
- DOS
- OS/2
- OS/400
- Tandem
- TSO (interpreter/compiler)
- UNIX
- VM (interpreter/compiler)
- VMS
- VSE (interpreter/compiler)
- WINDOWS

from 9 different sources.

There are freeware/shareware implementations for:

- MAC
- UNIX

from 4 different sources.

REXX Implementations by year First Available

Year	New Platform
1983	VM (IBM)
1985	PC-DOS (Mansfield)
1987	Amiga (W. S. Hawes)
1988	PC-DOS (Kilowatt) TSO (IBM)
1989	OS/2 (Mansfield) VM Compiler (IBM)
1990	UNIX/AIX(Workstation Group) Tandem (Kilowatt) OS/2 (IBM) AS/400 (IBM)
1991	DEC/VMS (Workstation Group) VM Compiler(Systems Center) 370 compiler (IBM) Amiga Compiler (Dineen Edwards Group)
1992	Windows (Kilowatt) Windows (Quercus) MS-DOS (Tritus) UNIX/AIX (Becket Group)

REXX Curiosities

Name of a REXX Entity

What is the name of a REXX entity??? Is it:

- Program
- Exec
- Macro
- Procedure
- Shell
- Script

Name of a REXX Entity ...

Term definitions are:

Program: A sequence of instructions suitable for processing by a computer. Processing may include the use of an assembler, a compiler, an interpreter, or a translator to prepare the program for execution, as well as to execute it.

Exec procedure: In VM, a CMS function that allows users to create new commands by setting up frequently used sequences of CP commands, CMS commands, or both, together with conditional branching facilities, into special procedures to eliminate the repetitious rekeying of those command sequences.

Macro instruction: An instruction that when executed causes the execution of a predefined sequence of instructions in the same source language.

Procedure: A set of related control statements that cause one or more programs to be performed.

Name of a REXX Entity ...

shell: A software interface between a user and the operating system of a computer. Shell programs interpret commands and user interactions on devices such as keyboards, pointing devices, and touch-sensitive screens and communicate them to the operating system.

script: In artificial intelligence, a data structure pertaining to a particular area of knowledge and consisting of slots which represent a set of events which can occur under a given situation.

Note: definitions come from the IBM "Dictionary of Computing".

Is REXX a....?

- Programming language
- Exec language
- Macro language
- Procedure language
- Command procedures language
- Extension language
- System Extension language
- Glue language
- Shell language
- Batch language
- Scripting language

Cowlishaw Book Cover

The 1990 edition of the Cowlishaw book has a new cover which includes the following changes:

- King now matches the playing cards King of Spades meaning
  - King faces the opposite way
  - King holds the sword differently

King was chosen because REX is Latin for King!

REXX Trivia Quiz Answers

Michael REXX
C. Mike Cowlishaw knights of VM name

Dame Cathie the symposiarch
N. Cathie Dager knights of VM name

Frankenrexx M. Neil Milsted's nickname

RIXX K. Rick McGuire licence plate

867-REXX L. Quercus telephone number

Procedures Language
G. Former IBM SAA name for REXX

AREXX	O. name of Amiga REXX
TREXX	Q. name of Tandem REXX
uni-REXX	D. name of AIX REXX from Workstation Group

1979	B. REXX formation started
1983	A. first REXX interpreter shipped
1985	J. first trade press book on REXX
1989	H. first REXX compiler available
1990	P. first REXX Symposium held
1991	I. ANSI work on REXX started

13	F. number of platforms with a REXX
11	E. number of trade press REXX books

REXXbits Summary

REXXbits Summary

- REXX is an international language
- REXX is growing in numbers of
  - implementers
  - different platforms available
  - users
  - books/articles.
- REXX is in the process of being formally standardized.
- REXX usage is in the "eyes of the beholder"!

Help Wanted

Please send me email when you find the following:

- New REXX books
- New REXX articles
- New REXX implementations
- Commercial products which:
  - written in REXX
  - use REXX as an extension language
  - extend REXX

I will share results on the latter category as I have done on the 3 former ones.

THE—THE HESSLING EDITOR

MARK HESSLING

Announcement of THE - The Hessling Editor

Mark Hessling M.Hessling@gu.edu.au

INTRODUCTION

THE is a full-screen character mode text editor based on the VM/CMS editor XEDIT and some features of KEDIT for DOS written by Mansfield Software.

THE has currently been ported to SUNOS 4.1.x, Xenix-386, DOS (using both Borland C++ 3.0 and Microsoft C v6.00), Esix 4.0.3a, ATT SystemV 3.2, Linux, 386BSD and to OS/2 2.0/2.1 (using Microsoft C v6.00 and C Set/2). Some work is still required to get THE working with emx 0.08f, Borland C++ for OS/2, and DJGPP under DOS.

An attempt has been made to port THE to VAX/VMS but some major work is still required. A port to the Amiga is currently being undertaken.

The DOS and OS/2 port requires the use of PDCURSES 2.1; a public domain library of curses screen handling routines, of which I am the current maintainer.

Both the OS/2 and Unix ports of THE interface to REXX interpreters to provide a full REXX macro capability. The OS/2 REXX support is provided with the REXX interpreter supplied with OS/2. The Unix REXX support is provided by the free REXX interpreter; Regina 0.05, written by Anders Christenson.

THE is distributed under the terms of the GNU General Public License.

HISTORY

Work began on THE in 1990 after my then workplace purchased a Sun workstation. This then meant that I was using DOS, VMS and Unix. This meant using 3 different editors. Having used XEDIT for a considerable period of time prior to 1990, I was keen to continue using an editor with the same capability and power. After receiving a copy of LED (Lewis Editor) from Pierre Lewis in Canada, I found that the only way to be able to use the same XEDIT-like editor on a variety of operating systems, was to write my own. Pierre assured me that writing an editor was wonderful for one's character.

The original intention of THE was to provide me with an editor I was happy with and had all the features of XEDIT and KEDIT that I used frequently. Once I had achieved this goal, I decided to make THE available to anyone who also had a need for a multi-platform XEDIT-like editor. THE 1.0 was released to the public in August 1991.

Since then, I then began to add features that I still found lacking and that other users requested. This work resulted in THE 1.1 which is publically released at this Symposium.

FUTURE

I will continue to add functionality to THE, provide support for more compilers and will also

AVAILABILITY

THE is available via anonymous ftp from the following sites:

North America:	rexx.uwaterloo.ca /pub/editors/the
Europe:	flipper.pvv.unit.no /pub/the
Australia:	ftp.gu.edu.au /src/THE

thedos11.zip	- version 1.1 DOS executable and documentation
theos211.zip	- version 1.1 OS/2 2.x executable and documentation
thesrc11.zip	- source code for version 1.1
thesrc11.tar.Z	- source code for version 1.1

BUG REPORTS

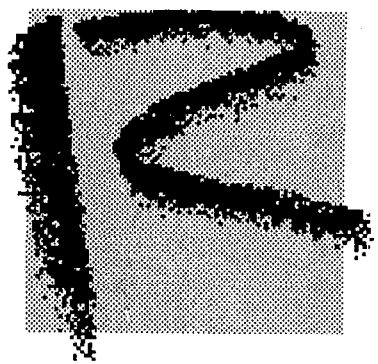
If you find bugs or major inconsistencies, please let me know. If you manage to compile and run on a different platform to the above, please send me any changes to the code and the makefile, so I can include the patches in the official release.

X-CUA

MICHAEL JOHNSON
Relay Technology

X-CUA

The X Window System under VM



RELAY

TECHNOLOGY, INC.

(Formerly VM Systems Group)

History of "X"

- **Display system developed at MIT**
  - — " Project Athena" formed in 1983; funded by MIT, DEC, and IBM
  - — addressed the need to communicate between different kinds of computers over a variety of networks
  - — adapted Stanford University's windowing system ("W") to Unix, renamed "X"
- **Goal: permit graphical presentation under UNIX**
  - — provided framework for a *Graphical User Interface* (GUI)
- **Led to formation of the X Consortium in 1987**
  - — partnership to guide future X standards
  - — founding members included IBM, Apple, Tektronics, DEC, Sun, Hewlett-Packard and AT&T

"X" Features and Functions

- **Object-oriented programming methods**
  - — application deals with whole objects vs. screen areas
- **Separates *client* and *server***
  - — application program is the client
  - — display management program is the server
  - — X Window server ("X server") handles the display; shares terminal with other X clients
- **Software comprised of:**
  - — intrinsics library–simple objects used to create "widgets"–more complex objects
  - — Xlib functions–procedures to perform low-level primitives (e.g., "DrawLine", "CreateWindow")

"X" Features and Functions *(continued)*

- **Display is performed on an X Terminal**
  - — real X Terminal–firmware X server/window manager
  - — PC running X server software
  - — UNIX/AIX workstation running X server software
- **Portable to virtually all platforms**
  - — any hardware capable of supporting an X terminal
  - — any operating system that can support C & communications
  - — communicates over Ethernet, TCP/IP, or DECnet
  - — "X" software available on UNIX, PCs, MACs, etc.
- **Does not impose GUI rules**
  - — adaptable to any GUI foundation: Open Look, OSF Motif, IBM's CUA

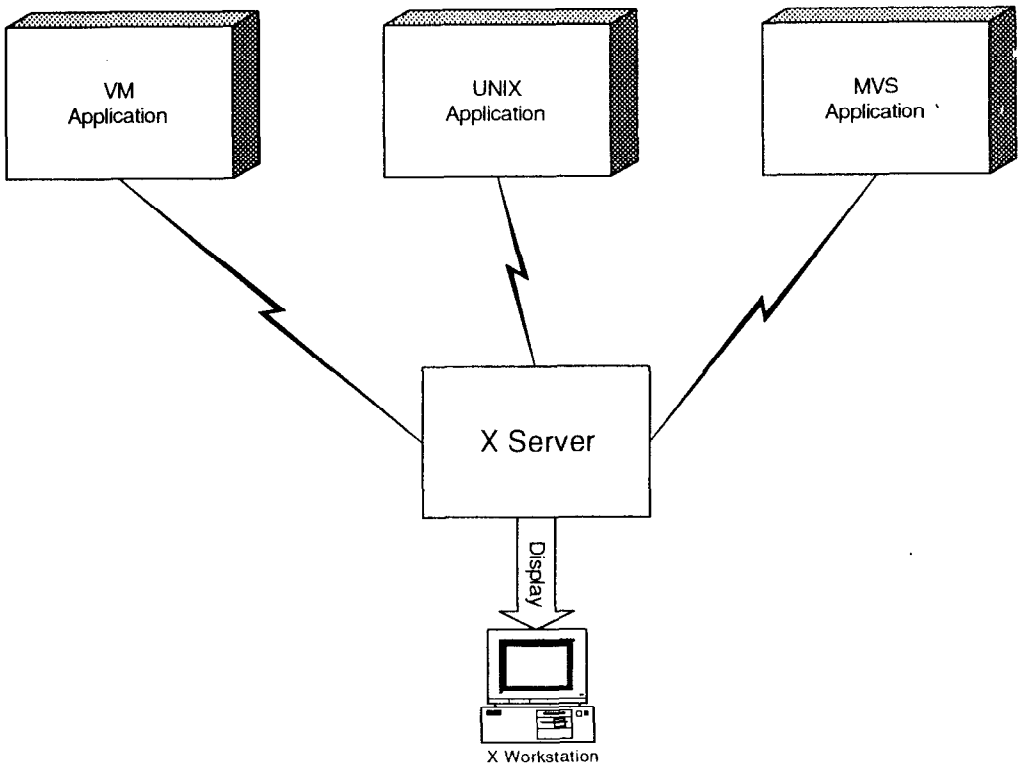
Why use "X"?

- **It is an enabling technology**
  — supports all GUIs, X terminals
- **All platforms can interoperate**
  — users may access distributed applications in parallel
- **Allows seamless integration across networks**
  — location of application is transparent (and usually irrelevant) to the user
- **Provides hardware-independent GUI foundation**
  — placement of client and server is unconstrained
- **Network transparent–TCP/IP, DECnet, etc.**
  — operates with existing communications

Why use "X"? *(continued)*

- **Supports corporate IS goals**
  — provides user-friendly GUI, increases productivity
- **An "Open" system–publicly available**
  — non-proprietary; no vendor dependencies
- **Increases product life cycle**
  — applications can be developed today, ported to other platforms in the future if necessary

The X Server



- **Controls the display screen, resides on workstation**

The X Server *(continued)*

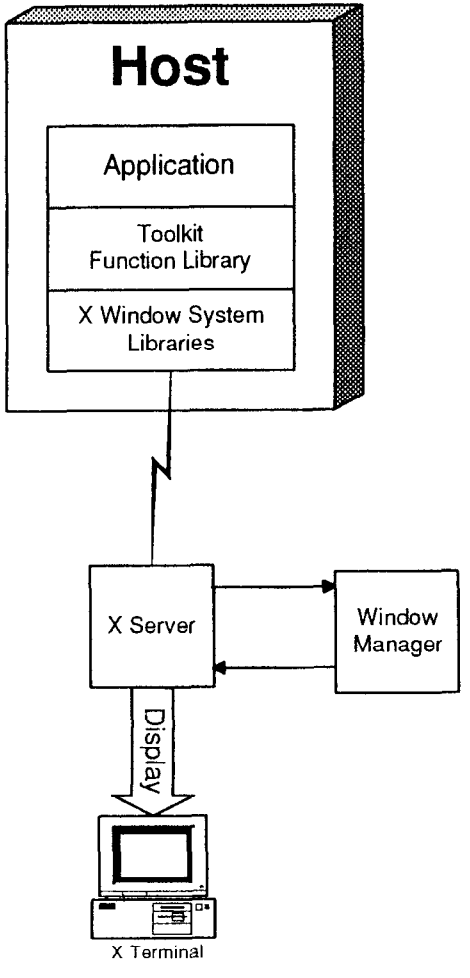
- **Receives instructions from the application in the form of X protocol messages**
- **Performs graphical tasks on behalf of the application (the X client)**
  - — understands terminal's characteristics
  - — makes application device-independent
  - — communicates geometry requests to window manager (a separate program usually included with X server)
- **Sends event messages back to the X Client**
  - — client receives notification, e.g., "ButtonPressed", etc.

The X Server *(continued)*

- **Can handle multiple, concurrent X clients (applications)**

  — single server handles all clients of user's terminal

The X Client



- The X server produces the window's contents and physically writes the display
- The window manager places the "dressing" (borders, etc.) around the window and manages the screen's "real estate" (window position, etc.)

The X Client *(continued)*

- **Oblivious to terminal's characteristics**
  — development effort focus is on tasks vs. environment
- **May exist on the same or different platform**
- **Event-driven methodology**
  — client establishes "callbacks" for desired events
  — client issues instructions, enters enabled wait state
- **Receives event notification from X server**
  — key pressed, window uncovered, etc.

What is X-CUA?

- **An X-based toolkit**
  - — adds CUA-compliant GUI functions to VM, AIX
  - — provides high-level X programming facilities for CUA-compliant applications ("CreateOpenDialog", etc.)
  - — programs can be written in C or REXX
- **Written in C**
  - — operates under UNIX, AIX, and VM
  - — X-CUA applications in C are portable
- **Under VM, uses IBM's TCP/IP product**
  - — enhanced version of the X Window support; recompiled using SAS/C to take advantage of advanced functions, permit royalty-free run time libraries.

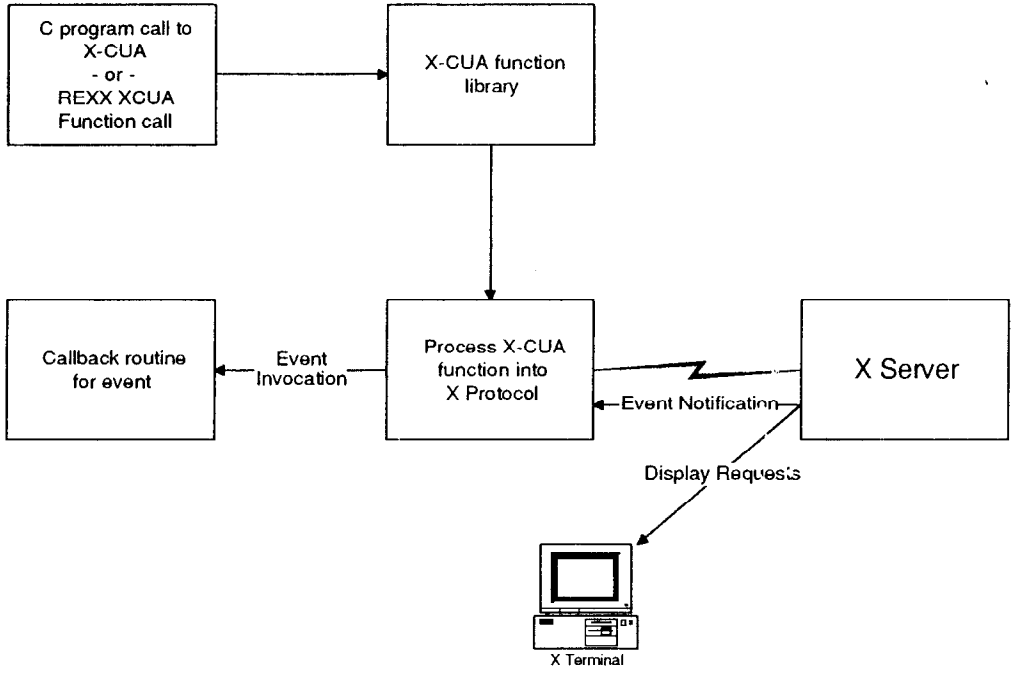
What is X-CUA? *(continued)*

- **REXX function package**
  - — it's REXX!
  - — runs in a saved segment for increased throughput
  - — relatively small applications due to non-redundancy
- **Provides enhanced 3270 emulation program**
  - — supports extended attributes, PSS, graphics
  - — want a 3279 model 4?
  - — users logon to VM, invoke X-CUA application
  - — X-CUA application begins GUI dialog
  - — So...

What is X-CUA? *(continued)*

- **Legacy systems can convert as resources permit**
  - — current system runs alongside in 3270 emulation
  - — pieces of applications can be converted gradually

X-CUA's role

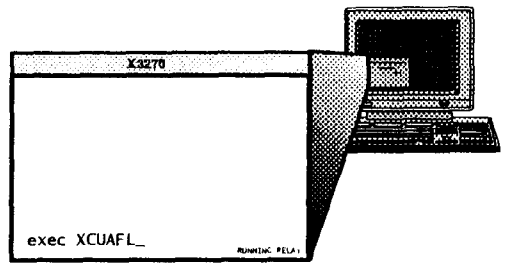


- **Applications in C or REXX call X-CUA**
  - — use supplied "widgets" – objects which implement CUA constructs

X-CUA's role *(continued)*

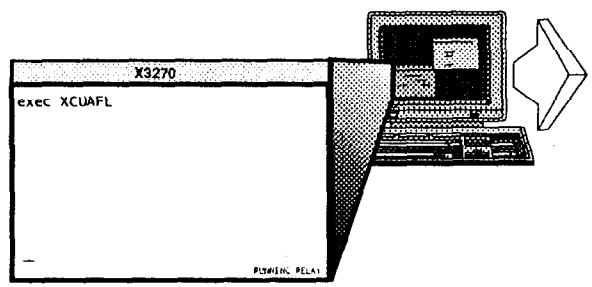
- **The process:**
  - — application calls X-CUA function to build GUI display
  - — registers callbacks for desired events:
    - external REXX function—e.g., "MYSORT EXEC" to handle "Sort" button
    - a C routine
  - — calls X-CUA to make window visible
  - — X-CUA translates requests into X protocol
  - — X-CUA transmits requests to X server
  - — X server sends back event notification
  - — X-CUA invokes callback routine

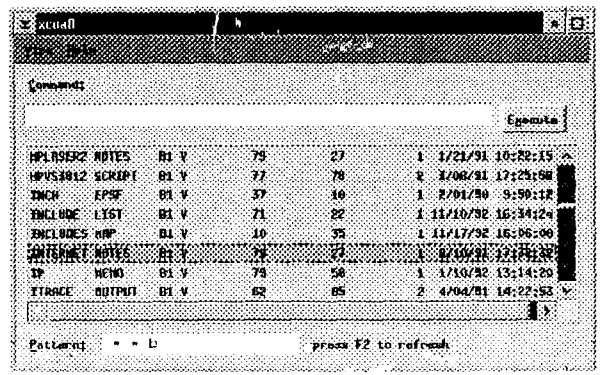
An X-CUA Application



- User logs onto VM through 3270 emulation
- Invokes X-CUA application
- X-CUA application begins initialization

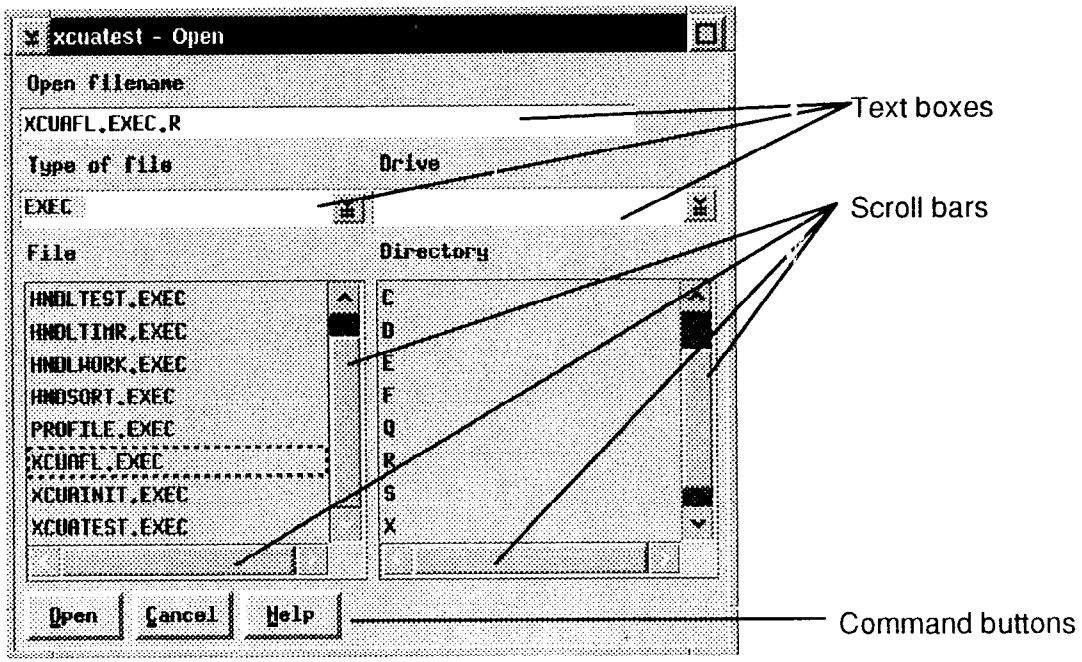
- X-CUA application opens a new window on user's terminal
- Interaction is through new window



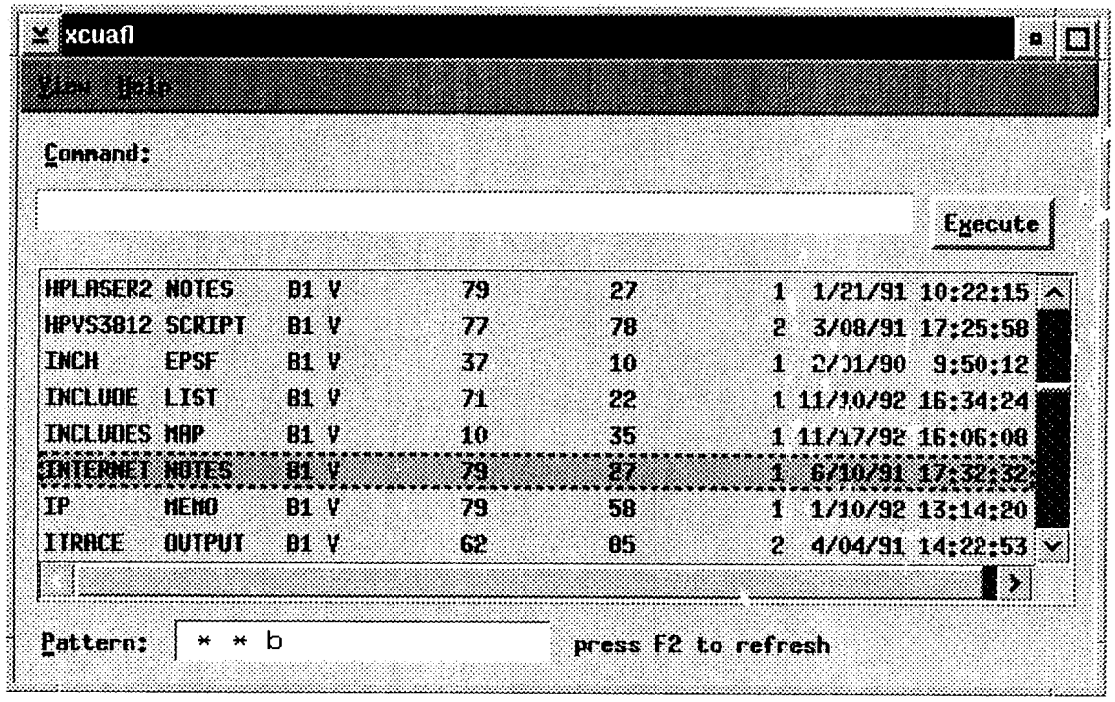


An X-CUA Widget

- **Open Dialog Box**
  - — X-CUA handles scroll bars, text edit, etc.
  - — application is presented with completed data



sis of an X-CUA Application



Analysis of an X-CUA application
(continued)

- **X-CUA REXX functions:**
  - — XCUA("SubfunctionName",args...)
    - Toolkit, windowing functions
    - Performs calls to C-level subroutines

- **GETARGS(), SETOPTION(), SETBUTTONREC(), etc.**
  - — X-CUA REXX utility functions
    - Provide bridge from REXX-think to C-think, vice versa

Analysis of an X-CUA application
(continued)

- **Let's get started:**

```
/* ------------------------------------------------------------------*/
/* X-CUA version of FILELIST utility.                                */
/* ------------------------------------------------------------------*/

address command                                    /* Good hygiene */
parse source . calltype .
signal on syntax

FALSE = 0                                        /* For clearer code */
TRUE = 1

if (calltype ¬== 'COMMAND') then      /* Figure out why we were called */
  signal CALLBACK                            /* More about this later */
```

Analysis of an X-CUA application
(continued)

- **First define the environment:**
  - — Fetch Arguments
  - — Configuration defaults
  - — Options, if there are any
  - — Create primary window
  - — Examine arguments
  - — Configure presentation space

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------------ */
/* XCUAFL mainline.                                                   */
/* ------------------------------------------------------------------ */

  'EXEC XCUAINIT'              /* Initialize the X-CUA REXX interface */

  call GETARGS 'ARGV.'      /* Get REXX args into a C-style args array */

/* ------------------------------------------------------------------ */
/* Create a Primary window, register an XcNcloseCallback routine.     */
/* ------------------------------------------------------------------ */
  fb.1 = "*TextFont: 9x15"           /* Define fallback configuration */
  fb.0 = 1                           /* In case no config file around */

  call SETOPTION 'OPTIONS.',1,"-ib","*inverseBackground","SepArg",""

  primary = XCUA("XcuaInitialize",'APP_CONTEXT',"XcuaFL",'OPTIONS.','ARGV.','FB.',)
  call XCUA "XtAddCallback",primary,"closeCallback","XCUAFL","EXIT"
```

Analysis of an X-CUA application
(continued)

```
filename = ''                          /* Examine remaining arguments */
if (argv.0 > 1) then                     /* Get file pattern if there */
  do i=2 to argv.0 for 3
    filename = filename argv.i
  end
else
  filename = '* * a'                  /* Otherwise set a good default */

/* ------------------------------------------------------------------- */
/* Set properties of the Presentation area.                            */
/* ------------------------------------------------------------------- */

presentation = XCUA("XcuaPropertyOf",primary,"clientArea")
call SETARG 'ARG.',1,"xScrollBar",FALSE
call SETARG 'ARG.',2,"yScrollBar",FALSE
call XCUA "XtSetValues",presentation,'ARG.'
```

Analysis of an X-CUA application
(continued)

- **Define the interface**
  - — Create a client area
  - — Populate it with objects
  - — Register callbacks
  - — Define secondary windows

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------------ */
/* Create a Form widget as the client area and configure it.          */
/* ------------------------------------------------------------------ */

  call SETARG 'ARG.',1,"hSpace",10
  call SETARG 'ARG.',2,"vSpace",10
  form = XCUA("XcuaCreateForm","form",presentation,'ARG.')

  call SETARG 'ARG.',1,"chars",50
  call SETARG 'ARG.',2,"inset",TRUE
  command = XCUA("XcuaCreateSingleEntry","command",form,'ARG.')

  prompt = XCUA("XcuaAddFieldPrompt",command,"Command:","C","above")
  call SETARG 'ARG.',1,"below",form
  call SETARG 'ARG.',2,"rightof",form
  call XCUA "XtSetValues",prompt,'ARG.'

  call SETARG 'ARG.',1,"label","Execute"
  call SETARG 'ARG.',2,"mnemonic","X"
  call SETARG 'ARG.',3,"rightof",command
  call SETARG 'ARG.',4,"acceptFocus",FALSE
  execute = XCUA("XtCreateManagedWidget","execute","PushButton",form,'ARG.')
  call XCUA "XtAddCallback",execute,"triggerCallback","XCUAFL","EXECUTE"
  call XCUA "XcuaSetDefaultButton",form,execute
```

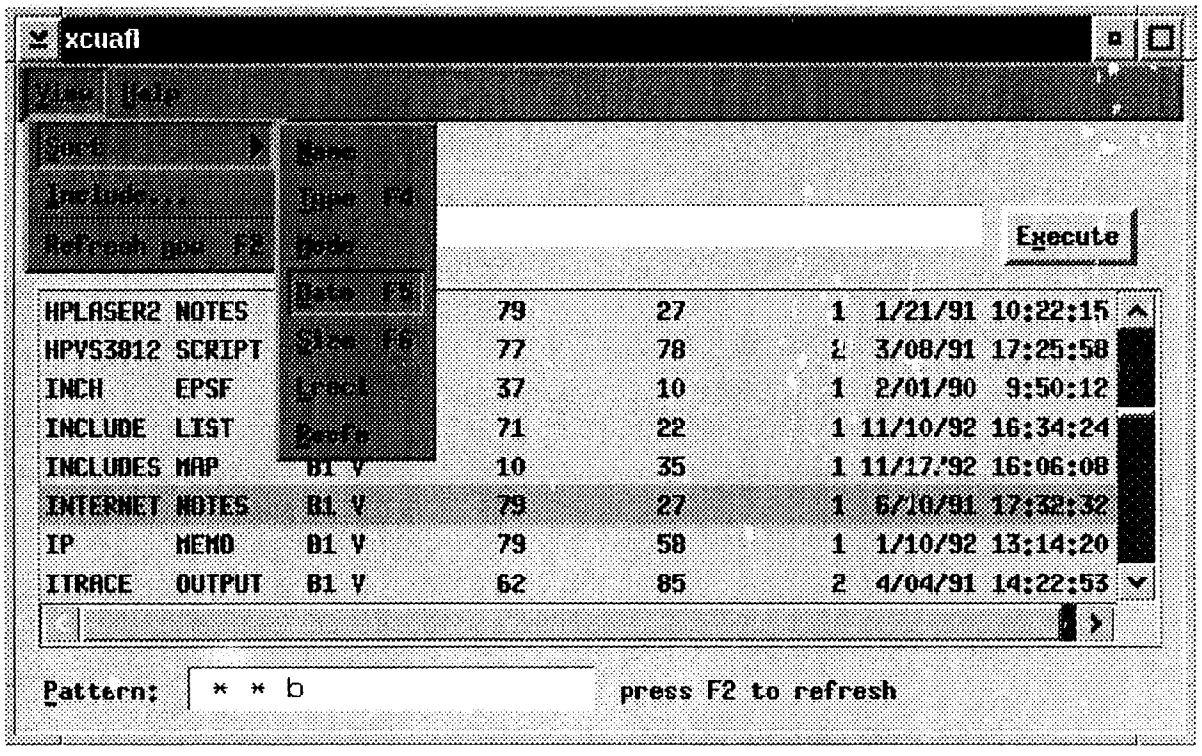
Analysis of an X-CUA application
(continued)

```
background = XCUA("XcuaPropertyOf",form,"background")
call SETARG 'ARG.',1,"below",execute
call SETARG 'ARG.',2,"rightof",form
call SETARG 'ARG.',3,"background",background
list = XCUA("XcuaCreateList","list",form,'ARG.')

call SETARG 'ARG.',1,"chars",21
call SETARG 'ARG.',2,"inset",TRUE
pattern = XCUA("XcuaCreateSingleEntry","pattern",form,'ARG.')
prompt = XCUA("XcuaAddFieldPrompt",pattern,"Pattern:","P","left")
call SETARG 'ARG.',1,"above",form
call SETARG 'ARG.',2,"rightof",form
call XCUA "XtSetValues",prompt,'ARG.'
call XCUA "XcuaAddDescriptiveText",pattern,"press F2 to refresh","right"
call SETARG 'ARG.',1,"above",prompt
call XCUA "XtSetValues",list,'ARG.'

call XCUA "XcuaSetText",pattern,filename
call XCUA "XcuaSetProperty",form,"initialFocus",command
```

Analysis of an X-CUA application
(continued)

```
/* ---------------------------------------------------------------- */
/* Create an Input dialog for the Include... action.                */
/* ---------------------------------------------------------------- */

  call SETARG 'ARG.',1,"message","Enter include file pattern:"
  include = XCUA("XcuaCreateInput","include",primary,'ARG.')
  call XCUA "XtAddCallback",include,"okCallback","XCUAFL","INCLUDE"
  call XCUA "XtAddCallback",include,"cancelCallback",,
    "XcuaCallbackCloseDialog",include
```

Analysis of an X-CUA application
(continued)



Analysis of an X-CUA application
(continued)

```
/* ---------------------------------------------------------------- */
/* Define custom menus for this application.                        */
/* ---------------------------------------------------------------- */

  ACTION = 1
  CASCADE = 2
  SEP = 3

  call SETBUTTONREC 'VIEWBUTTON.',1,CASCADE,"view","View",,"V",,"VIEW."

  call SETBUTTONREC 'VIEW.',1,CASCADE,"sort","Sort",,"S",,"SORT."
  call SETBUTTONREC 'VIEW.',2,ACTION,"include","Include...",,"I",,,
    "XcuaCallbackOpenDialog",include
  call SETBUTTONREC 'VIEW.',3,SEP
  call SETBUTTONREC 'VIEW.',4,ACTION,"refresh_now","Refresh now",,"N","F2",,
    "XCUAFL","REFRESH"

  call SETBUTTONREC 'SORT.',1,ACTION,"name","Name",,"N",,"XCUAFL","SORT NAME"
  call SETBUTTONREC 'SORT.',2,ACTION,"type","Type",,"T",'F4","XCUAFL","SORT TYPE"
  call SETBUTTONREC 'SORT.',3,ACTION,"mode","Mode",,"M",,"XCUAFL","SORT MODE"
  call SETBUTTONREC 'SORT.',4,ACTION,"date","Date",,"D","F5","XCUAFL","SORT DATE"
  call SETBUTTONREC 'SORT.',5,ACTION,"size","Size",,"S","F6","XCUAFL","SORT SIZE"
  call SETBUTTONREC 'SORT.',6,ACTION,"lrecl","Lrecl",,"L",,"XCUAFL","SORT LRECL"
  call SETBUTTONREC 'SORT.',7,ACTION,"recfm","Recfm",,"R",,"XCUAFL","SORT RECFM"
```

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------------ */
/* Set up the menu bar, create the custom menus.                      */
/* ------------------------------------------------------------------ */

  menubar = XCUA("XcuaMenuBarOf",presentation)
  call XCUA "XcuaGenerateMenu",menubar,'VIEWBUTTON.'
  call XCUA "XcuaMenuBarVisible",menubar,"file",0
  call XCUA "XcuaMenuBarVisible",menubar,"edit",0
  call XCUA "XcuaMenuBarCallback",menubar,"help","about","XcuaProductInfo",,
    "This is XCUAFL, a CUA-compliant"||'15'x||,
    "X-based FILELIST utility."||'15'x||'15'x||,
    "To contact the author, write to:"||'15'x||'15'x||,
    "     Relay Technology, Inc."||'15'x||,
    "     1604 Spring Hill Road"||'15'x||,
    "     Vienna, VA 22182"
```

Analysis of an X-CUA application
(continued)

- **Load the contents of the primary window:**

```
call XCUAFL primary,"REFRESH"
```

- **Make it visible:**

```
call XCUA "XcuaOpenPrimary",primary
```

- **Enter the main event handling loop:**

```
rc = XCUA("XcuaMainLoop",app_context)
exit rc
```

Analysis of an X-CUA application
(continued)

- **And the handler for possible syntax error:**

SYNTAX:

```
say "Syntax error" rc "occured on line" sigl":" errortext(rc)
if (symbol('APP_CONTEXT') <> "LIT") then
  call XCUA "XtDestroyApplicationContext",app_context

exit rc
```

Analysis of an X-CUA application
(continued)

- **Callbacks revisited**

  — X-CUA applications are event-driven, via callbacks
  — The application is called as a command
  — The callback handler is called as a function
  — We use PARSE SOURCE to determine calltype, handle both in the same source file

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------------ */
/* Callback()                                                         */
/*                                                                    */
/* Function: Handle callbacks                                         */
/*                                                                    */
/* ------------------------------------------------------------------ */

CALLBACK:
parse arg widget,client_data,call_data,callback_id

  valid = "EXIT EXECUTE REFRESH SORT INCLUDE"
  label = word(client_data,1)

  if (find(valid,label) == 0) then
    return
  signal value (label)

  /* ------------------------------------------------------------------ */
  /* Handle an XcNcloseCallback.                                        */
  /* ------------------------------------------------------------------ */

  EXIT:

    call XCUA "XcuaExit",0

  return
```

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------- */
/* Handle an XcNtriggerCallback for the EXECUTE button.          */
/* ------------------------------------------------------------- */

EXECUTE:

  form = XCUA("XtParent",widget)
  command = XCUA("XcuaGetChild",form,"command")
  list = XCUA("XcuaGetChild",form,"list")
  primary = XCUA("XcuaPrimaryOf",widget)
  display = XCUA("XtDisplay",widget)

  directive = XCUA("XcuaSubstr",command,0,-2)
  call XCUA "XcuaClearText",command

  if (directive = '') then
    return

  item = XCUA("XcuaListItemOf",list,0)
```

Analysis of an X-CUA application
(continued)

```
  do until (item == '00000000'x)
    next = XCUA("XcuaNextItem",item);
    if (XCUA("XcuaItemSelected",item)) then do
      label = XCUA("XcuaItemLabel",item)
      parse var label fname ftype fmode .
      call XCUA "XcuaSetWaiting",primary,TRUE
      call RunCommand directive,fname,ftype,fmode
      call XCUA "XcuaSetWaiting",primary,FALSE
      'ESTATE' fname ftype fmode
      if (rc == 28) then
        call XCUA "XcuaDeleteListItem",item
    end
    item = next
  end

return
```

Analysis of an X-CUA application
(continued)

```
/* ------------------------------------------------------------------ */
/* Handle an XcNtriggerCallback for the REFRESH menu choice.          */
/* ------------------------------------------------------------------ */

REFRESH:

  primary = XCUA("XcuaPrimaryOf",widget)
  presentation = XCUA("XcuaPropertyOf",primary,"clientArea")
  form = XCUA("XcuaScrolledOf",presentation)
  pattern = XCUA("XcuaGetChild",form,"pattern")
  list = XCUA("XcuaGetChild",form,"list")

  call XCUA "XcuaSetProperty",pattern,"error",FALSE
  directive = XCUA("XcuaSubstr",pattern,0,-2)

  parse upper var directive fname ftype fmode
  if (fmode = '') then
    fmode = 'A'
  if (ftype = '') then
    ftype = '*'
  if (fname = '') then
    fname = '*'
```

Analysis of an X-CUA application
(continued)

```
stacked = queued()
   'MAKEBUF'
   'LISTFILE' fname ftype fmode '( NOHEADER DATE FIFO )'
   if (rc ¬== 0) then do
     'DROPBUF'
     call XCUA "XcuaSetProperty",pattern,"error",TRUE
     return
   end

   stacked = queued() - stacked
   call XCUA "XcuaSetWaiting",primary,TRUE
   call XCUA "XcuaClearList",list
   do i=1 to stacked
     parse pull label
     call XCUA "XcuaAddListItem",list,label
   end
   'DROPBUF'
   call XCUA "XtCallActionProc",list,"bod"
   call XCUA "XcuaSetWaiting",primary,FALSE

return
```

Analysis of an X-CUA application
(continued)

```
/* ---------------------------------------------------------------- */
/* Handle a SORT call                                               */
/* ---------------------------------------------------------------- */

SORT:
parse var client_data . field .

  primary = XCUA("XcuaPrimaryOf",widget)
  presentation = XCUA("XcuaPropertyOf",primary,"clientArea")
  form = XCUA("XcuaScrolledOf",presentation)
  list = XCUA("XcuaGetChild",form,"list");

  ASCENDING = TRUE
  DESCENDING = FALSE
  REFRESH = TRUE
  NOREFRESH = FALSE

  select
    when (field == 'NAME') then do
      call XCUA "XcuaSortList",list,ASCENDING,19,20,NOREFRESH
      call XCUA "XcuaSortList",list,ASCENDING,10,17,NOREFRESH
      call XCUA "XcuaSortList",list,ASCENDING,1,8,REFRESH
    end
    when (field == 'TYPE') then do
      call XCUA "XcuaSortList",list,ASCENDING,19,20,NOREFRESH
      call XCUA "XcuaSortList",list,ASCENDING,1,8,NOREFRESH
      call XCUA "XcuaSortList",list,ASCENDING,10,17,REFRESH
    end
```

Analysis of an X-CUA application
(continued)

```
when (field == 'MODE') then do
    call XCUA "XcuaSortList",list,ASCENDING,10,17,NOREFRESH
    call XCUA "XcuaSortList",list,ASCENDING,1,8,NOREFRESH
    call XCUA "XcuaSortList",list,ASCENDING,19,20,REFRESH
  end
  when (field == 'DATE') then do
    call XCUA "XcuaSortList",list,DESCENDING,72,73,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,69,70,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,66,67,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,60,61,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,57,58,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,63,64,REFRESH
  end
  when (field == 'SIZE') then do
    call XCUA "XcuaSortList",list,DESCENDING,35,44,NOREFRESH
    call XCUA "XcuaSortList",list,DESCENDING,46,55,REFRESH
  end
```

Analysis of an X-CUA application
(continued)

```
      when (field == 'RECFM') then do
        call XCUA "XcuaSortList",list,ASCENDING,35,44,NOREFRESH
        call XCUA "XcuaSortList",list,ASCENDING,46,55,NOREFRESH
        call XCUA "XcuaSortList",list,ASCENDING,22,22,REFRESH
      end
      when (field == 'LRECL') then do
        call XCUA "XcuaSortList",list,ASCENDING,19,20,NOREFRESH
        call XCUA "XcuaSortList",list,ASCENDING,10,17,NOREFRESH
        call XCUA "XcuaSortList",list,ASCENDING,1,8,NOREFRESH
        call XCUA "XcuaSortList",list,ASCENDING,24,33,REFRESH
      end
    otherwise
      call XCUA "XcuaInfoMessage",widget,"sort message","Unrecognized",
        "sort type '"field"'"
      return
    end

    call XCUA "XcuaSetPosition",list,0,0

  return
```

Analysis of an X-CUA application
(continued)

```
/* ---------------------------------------------------------------- */
/* Handle an INCLUDE call                                           */
/* ---------------------------------------------------------------- */

INCLUDE:

  call GETCALLBACKEVENT 'EVENT.',call_data
  drop result
  string = GETSTRING(event.result)
  call XCUA "XcuaCloseDialog",widget

  primary = XCUA("XcuaPrimaryOf",widget)
  presentation = XCUA("XcuaPropertyOf",primary,"clientArea")
  form = XCUA("XcuaScrolledOf",presentation)
  pattern = XCUA("XcuaGetChild",form,"pattern")
  list = XCUA("XcuaGetChild",form,"list")

  parse upper var string fname ftype fmode
  if (fmode = '') then
    fmode = 'A'
  if (ftype = '') then
    ftype = '*'
  if (fname = '') then
    fname = '*'
```

Analysis of an X-CUA application
(continued)

```
    stacked = queued()
    'MAKEBUF'
    'LISTFILE' fname ftype fmode '( NOHEADER DATE FIFO )'
    if (rc ¬== 0) then do
      'DROPBUF'
      return
    end

    stacked = queued() - stacked
    call XCUA "XcuaSetWaiting",primary,TRUE
    do i=1 to stacked
      parse pull label
      call XCUA "XcuaAddListItem",list,label
    end
    'DROPBUF'
    call XCUA "XcuaSetWaiting",primary,FALSE

  return
```

Analysis of an X-CUA application
(continued)

```
/* ---------------------------------------------------------------- */
/* RunCommand()                                                     */
/*                                                                  */
/* Function: run a command on a file, with substitution.            */
/*                                                                  */
/* ---------------------------------------------------------------- */
RUNCOMMAND:
parse arg command, fn, ft, fm
  subbed = 0
  index = 1
  do until (index == 0)
    index = pos('/',command,index)
    drop sub
    if (index > 0) then do
      flag = substr(command,index+1,1)
      word = substr(command,index,2)
      upper flag
      if (flag == 'N') then sub = fn
      else if (flag == 'T') then sub = ft
      else if (flag == 'M') then sub = fm
      else if (flag == 'O') then sub = ' '
      else if (flag == ' ') then do
        sub = fn ft fm
        word = '/'
      end
    end
  end
```

Analysis of an X-CUA application
(continued)

```
 if (symbol('SUB') <> "LIT") then do
     subbed = 1
     command = delstr(command,index,length(word))
     command = insert(sub,command,index-1)
     index = index + length(sub)
   end
   else if (index > 0) then
     index = index + 1
  end

  if (¬subbed) then
    command = command fn ft fm

  address CMS command

return
```

X-CUA Requirements

- X Terminal
- VM and/or AIX (currently; other Unix to follow)
- IBM TCP/IP (for VM)
- In Alpha test, GA scheduled for Q3

Summary

- **X-CUA provides an enabling tool to move mainframe-based application user interfaces to a GUI**
  - — CUA-compliant
  - — not vendor-specific
- **Provides true, transparent, seamless interoperability**
- **Maximizes productivity through familiarity, GUI power**

Summary *(continued)*

- **CUA compliance enables standardization, reduced training costs**
- **Legacy systems may be updated with a modern look while preserving corporate investment**
- **Many legacy systems belong on the mainframe; X-CUA allows these to stay there**

DESIGN OF THE EMERGING REXX STANDARD

BRIAN MARKS
ANSI

This talk is in four sections.

The "Dull but informative" section is about the history of the committee, mandate, membership and progress so far.

The "Rexx users won't care" section is about technical aspects of writing the definition.

"What is new about errors" will explain the changes and extensions there.

"What is new about Command I/O" will explain our attempt to provide uniformity in the way data is exchanged across the interface between Rexx and system commands.

==========

At the 1990 Rexx symposium in Stanford, a panel of Rexx experts was asked "Would a standard for Rexx be a good thing, and would you contribute to creating one?". Based on the strong support expressed there, a proposal was made and the first meeting of the X3J18 committee was held in January 1991.

The proposal that X3 voted on when setting up the committee had this key paragraph:

"The scope of the standard will be the second edition of the Cowlishaw book, plus consideration of implementation experience. The scope may be altered as necessary to promote portability, reliability, maintainability and efficient execution of REXX programs on a variety of computing systems."

Note that this mandate doesn't allow the committee to add things just because users would like them. While we all know of extensions that have been frequently requested, like date conversions and new ways of using stemmed variables, the committee doesn't consider them for the first standard. (Although we do have separate discussions with a view to a subsequent standard.)

We have just had our ninth meeting. The attendance has varied, but typically is about ten.

Membership is a big investment. The membership fee is only 300 dollars a year but the cost lies in travel and in the member's time. The four meetings a year are spread over the USA (to even out costs) and each lasts several days. In theory, there is even more cost in the time member's spend on X3J18 between meetings but in practice they are all professionals with other jobs, making it difficult for them to provide that extra contribution.

The balance of the committee has more participants that are primarily implementers than participants that are solely users. Of course, the implementers also represent users, but we would like more members from the user community.

The work has progressed to the point where we have most of a draft of the standard, although there is a lot of detail work to do. There are two extremes of looking at that: "How can so little be done in two years?" (if you count elapsed time) or "A great result for a month's work" (if you only count the meetings). The truth, no doubt, is

somewhere in between.

==========

The audience for a standard comprises implementers and people who want to validate implementations. Such people understand Rexx so the standard doesn't have to be a tutorial; it does need to be rigorous and complete. The actual users of Rexx are not so interested in how the draft is written, only in its content, which will be reflected in the manuals for users.

There are some languages, like VDL and Z, which were specifically designed for writing formal definitions. One of these could have been used to write the standard, but we chose to write much of the standard in Rexx.

Superficially this is circular, using a language to define itself, but in practice it is a bootstrapping exercise. The parts not written in Rexx provide the foundation for parts which are written in Rexx.

The syntax of programs can be specified using grammars in the familiar BNF notation. We use one for the tokens and one for the higher level constructs. There is an interaction between these grammars because of the detection of keywords and implied concatenations. Detecting keywords is a good example of something that a standardizing committee has to work hard on but the user doesn't care what the answer is.

For example, the rule about 'DO' keywords is different in the "Red Book" from the "Blue Book":

"The sub-keywords WHILE and UNTIL are reserved within a DO instruction, in that they cannot be used as symbols in any of the expressions. Similarly, TO, BY, and FOR cannot be used in expri, exprt, exprb, or exprf. FOREVER is also reserved, but only if it immediately follows the keyword DO."

"The sub-keywords TO, BY, FOR, WHILE, and UNTIL are reserved within a DO instruction, in that they cannot name variables in the expression(s) but they may be used as the name of the control variable. FOREVER is similarly reserved, but only if it immediately follows the keyword DO."

It is doubtful if any Rexx programmer cares at all, but it has to be defined. To avoid special cases for individual keywords, the committee has come up with the rule "If it could be a keyword it is". That means that the BNF and the rules for detecting labels and assignments are all applied in a left-to-right way; if then a potential keyword occurs and there has been nothing to contradict the possibility of it being a keyword then it is a keyword. We don't think this rule changes the behaviour of any existing error-free programs and it guarantees the definition hasn't missed any case.

==========

The "Red Book" goes further than many language definitions by specifying the exact wording of all the error messages. However, it doesn't always say when a particular message is produced. There are some cases where the book says "It is an error..." and leaves it to the implementer to choose the message from the given set of messages. Not all implementers have made the same choice. This is another topic where the actual Rexx

user probably doesn't care much what is done; but I think the committee is right to standardize the choice.

We do this for syntax errors (that is programs that contradict the BNF) by annotating the BNF to show what message should be produced for failures at any point. We do it for execution errors by writing tests in the definition for particular numbered errors.

The number of distinct error messages defined in the "Red Book" is far fewer than the number of places where the standard will detect an error. So the simplest approach would lead to the same message being given for several places where it was detected. For example, many different things wrong with a call might be detected, but they would all lead to syntax error 40.

The committee has decided to extend Rexx with subcodes - this is within our mandate because of the portability and maintainability considerations in error detection. So there will be, for instance,

Error 40.16 Argument <number> to routine <name> must be non-null

Existing programs might be dependent on actually testing the major error number, the 40 in this case, so that part of the language isn't changed. The subcode only comes into play if the program chooses to ask for it or on termination messages.

==========

The ability to issue commands is a central pillar of Rexx strengths. The way of issuing commands is well-defined; a clause which is an expression. However, the way in which the commands access Rexx data and the way in which Rexx accesses the results of commands are not well-defined. There are mechanisms that can be used, such as streams, the stack, and the variable pool but how the commands actually do their I/O has always been left up to the implementation.

The committee has added an extension which will be available on all systems that conform to the standard, and hence provide a more portable way of using commands.

The ADDRESS instruction now has extra options:

```
ADDRESS ... WITH INPUT STREAM MyOne OUTPUT STREAM MyTwo

ADDRESS ... WITH INPUT STEM RxOut. OUTPUT STEM RxIn.
```

This way of using stemmed variables has been a popular convention in conjunction with implementers' "extras" to Rexx so the committee is not forcing some completely untested invention on users.

==========

That is the end of the presentation except to remind you that this is merely an account of a snapshot in the development process - the content of the standard when eventually approved could be entirely different.

Dr B L Marks
Room G.O.023, MP 212
IBM UK Labs Ltd, Hursley Park, Winchester, Hampshire, England
Tel 44-962-844433 Ext 6643 Internet:marks@winvmd.vnet.ibm.com

DEFECT REMOVAL TECHNIQUES FOR REXX

PAT MEEHAN AND PAUL HEANEY

Defect Removal Techniques and their Effectiveness for REXX Applications

Patrick A Meehan IBM IISL PRGS Lab, Dublin, Ireland

Paul Heaney DELPHI Software Limited, Dublin, Ireland

Abstract

Major focus has been put on the reliability of software within the last few years resulting in various attempts to improve that reliability and to produce software with close to zero-defect (six-sigma). Little effort has been expended to measure the relative effectiveness of the different techniques in a controlled fashion.

This paper focuses on the experiences of defect removal of a component of an existing REXX product and the subsequent comparison in a more controlled fashion between different methods of defect removal for a new REXX project.

The main focus of the paper will be on the measured effectiveness of different defect removal techniques and on their suitability to an application that has already been or will be developed in the REXX language with the overall objective of producing close to zero defect REXX applications.

Introduction

Many different philosophies exist as to the best way of ensuring high reliability software systems. Inspections and/or reviews of the different development phases, various forms of code testing and standards by which the development should proceed often figure among these approaches. There has however been little attempt to measure the effectiveness of the different techniques in a controlled fashion.

This paper describes work undertaken by the authors and other participants in an attempt to measure the effectiveness of different defect removal techniques during the coding phase. The incentive to carry out this research was based on our experience with the development of a Program Product component. This work involved the development of key performance changes which varied in complexity from basic performance changes to complex network changes. There were a number of key elements in this development effort.

Performance changes were prototyped at an early stage of the design process to gain some early measurements on their benefit. The resultant code was subjected to some extensive unit testing. Parallel reviews of the entire code were conducted. Results from the reviews were carefully analysed and in some instances the subject code was seeded in an attempt to measure the effectiveness of the parallel reviews. In more complex parts of the development, informal verification by the owner of the code was carried out. The performance component has handed over to formal Test phase with a defect residue of 2.6 defects per KLOC. This low residue compared very well to other components and was less than the average defect residue for projects developed with the Cleanroom techniques (1). The defects that were discovered during formal test were typically of a trivial nature and were easy to fix.

These results suggested that the techniques or at least some of the techniques practiced were very successful. However, it wasn't clear as to which was more effective and whether some combination of

the techniques might be even more effective than others. In order to determine their effectiveness, it was necessary to set up an experiment and measure their efficiencies in a more controlled fashion.

The objective of the experiment was to measure a selection of different techniques on a piece of subject code under a variety of different metrics. The selection was based on techniques typically practiced in software development and are described below.

A small REXX project to manage the reporting of PTR (Problem Tracking Reports) was designed based on well known requirements, the resultant design was reviewed and the code was developed (3K). The resultant code then became the subject of the experiment.

Different Methods

A number of different techniques were employed in order to establish their effectiveness in removing defects from the established REXX program. The exact same REXX code was the subject of all the techniques selected. The following example (please refer to Figure 1 on page 3) which is a selected piece of code from the developed REXX reporting project serves to explain the different methods used and the manner of their use from a REXX perspective.

The inputs to all the techniques were :

- Source code
- Intended function
- Design document

For the purposes of easy reference, each decision with the section of code is referenced on the right hand side of the decision (e.g. B.2.3).

Unit Testing

Unit testing can take on many different forms from the basic statement coverage to the more rigorous form of multiple-condition based unit testing and can vary significantly in their success rates (2).

Decision based Unit Testing

The purpose of this form of testing was to ensure that each decision within the code took on a true and false outcome and then checking that the result was valid. This was carried out by someone other than the code author but who was involved in the original design.

The SIGNAL ON NOVALUE and SIGNAL ON SYNTAX instructions were also added to the code in order to detect uninitialised variables and interpretation errors and NOVALUE and SYNTAX routines were inserted to trap these errors.

In general, where there are n decisions then this would mean 2*n number of test scenarios. However, the number of actual test cases is usually less than this because the different decisions are typically not all independent of each other and even where they are independent of each other they can sometimes co-exist within the same test case.

From the example in the figure (please refer to Figure 1 on page 3), there are 4 decisions. In order for each decision to take on a false and true outcome this would have required the following set of 8 potential test scenarios.

1. b.2, b.2′, b.2.1, b.2.1′, b.2.2, b.2.2′, b.2.3, b.2.3′

where the prime indicates the false outcome of the decisions. On closer examination, it becomes apparent that all of the test scenarios of the form b.2.x′ can be satisfied by the scenario b.2.y where y $\neg$ = x. In addition b.2 must co-exist with any of the list of b.2.1, b.2.2 and b.2.3 so it doesn't have to exist as a separate test case.

So we are really left with the following set of 4 test cases:

1. b.2′
2. b.2.1, b.2
3. b.2.2, b.2
4. b.2.3, b.2

This set of test cases discovered 2 defects in the selected piece of code where the keywords *SUBTRACT* and *ADD* were not included in quotes. It's of interest to note that these would equally have been found through the use of the SIGNAL ON NOVALUE instruction.

```
MANAGE_PCFRAISE:
array = 'RAISE'

 Select
  When type = 'ACTED'  then do
/* Valid Acted and either it was previously OPENed or it was ACTED on..*/
/* but the REL info is different or it was logged as Rejected.....*/
/* Decision..... B.2 */
   If rel ¬ = 'NOT',
     & ('WORD'(p.ptr_no,1) = 'OPEN',
     | ('WORD'(p.ptr_no,1) = 'ACTED' & 'WORD'(p.ptr_no,3) ¬ = rel)),
     | ('WORD'(p.ptr_no,1) = 'REJECT'),
     then do
/* Decision..... B.2.1*/
       If 'WORD'(p.ptr_no,1) = 'OPEN' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 2 'SUBTRACT' array
/* Decision..... B.2.2*/
       If 'WORD'(p.ptr_no,1) = 'REJECT' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 6 'SUBTRACT' array
/* Decision..... B.2.3*/
       If 'WORD'(p.ptr_no,1) = 'ACTED' then
          Call addsub_operator 'WORD'(p.ptr_no,4) 5 SUBTRACT array
       Call addsub_operator ymd_open 3 ADD array
       End

    Otherwise nop
    End

/*.....contd.....*/
```

Figure 1. Sample of Subject REXX Code - Input to all techniques.

Multiple Condition Based Unit Testing

Typically, the code author would unit test his/her code and for this reason the subject code was unit tested by the code author along the lines of multiple condition based unit testing.

Whereas decision based unit testing just focuses on the decision outcome, multiple condition based unit testing focuses on the actual conditions within the decision by ensuring that all possible condition combinations within a decision are exercised.

For example, decision b.2 has 5 different conditions within it and theoretically there are 2 to the power of 5 test scenarios to cover all condition combinations (32). Decisions b.2.1, b.2.2 and b.2.3 have only 1 condition within each and so are handled in the same fashion as with decision based unit testing.

On closer examination of the 5 conditions with decision b.2 it becomes apparent that only a certain subset are possible anyway. For example, the expression *WORD(p.ptr_no,1)* which we refer to as the PTR Status can have only 1 value at a time. If we name the 3 occurrences of this expression as x2, x3 and x5 then the following are the only 4 valid combinations

(x2,x3',x5'), (x2',x3,x5'), (x2',x3',x5), (x2',x3',x5')

where the prime (') indicates the false outcome of the expression. If we name the other conditions in decision b.2 as x1 and x4 then it is clear that these can take on the following 4 valid combinations

(x1,x4), (x1',x4), (x1',x4') and (x1,x4')

Therefore the total number of test cases becomes 4 times 4 or 16 valid test cases which is only half of the number of original scenarios.

Note: Decisions b.2.1 , b.2.2 and b.2.3 are automatically covered by these test cases and no further test cases are required.

This set of test cases discovered the 2 defects already mentioned under decision based unit testing (Test case x1,x2',x3,x4,x5'). In addition, they uncovered a further defect through the following test case combination of (x1',x2',x3',x4',x5) which actually resulted in condition b.2.2 being executed when in fact this particular section of code should not have been entered at all. The coding error was due to the fact that whenever x5 occurred (PTR Status = 'REJECT') regardless of the other conditions x1,x2,x3 and x4, the underlying code was executed whereas it should only have been executed when x5 *AND* x1 occurred. The error arose because the incorrect placement of parenthesis in the decision (Please refer to Figure 2 on page 6).

This defect was not detected under the decision based testing because of it doesn't embrace the different condition combinations within a decision and underlines the inadequacy of the decision based approach.

Verification of REXX Code

Another form of defect removal which has gained some prominence recently particularly since it forms a significant part of the entire Cleanroom methodology is that of verification. As part of the experiment, code verification was undertaken by the code author. This activity took place some 3 months before the multi-condition based unit testing in order to eliminate any potential bias due to the fact that the same person carried out both activities.

Verification is a means of expressing the function of a manageable section of code in an unambiguous fashion and then exercising some intellectual reasoning about the derived function and the original intended function.

The intended function was documented within the actual code when the code was originally written.

A key part of verification was that the code was not executed. Verification was conducted by establishing the derived function for each main section of code within a Procedure and then cascading towards an derived function for the entire Procedure and ultimately an overall derived function for the entire REXX program. The derived function should be sequence free and loop free because this makes it more understandable and more unlike the original code.

Some people advocate a more formal description of the program function; it is our experience that the choice of description for REXX code depends largely on the nature of the code. The authors believe that is important to describe the derived function in a more conceptual fashion and that it is important to divorce it from the actual code details as much as is possible. The use of lists, matrices, and other mathematical notation were considered invaluable.

The key to verification of developed code is a complete understanding of exactly what the code is doing. It is recommended that even where one may think that they know how a particular REXX construct or operating system command works, one should still consult the relevant documentation to verify that understanding. Once that understanding is established, then it is relatively easy to verify it against the intention.

The example (please refer to Figure 2 on page 6) shown is the derived Program Function for two sections of code in a Procedure (B.1 and B.2).

Note: Any non-obvious notation is described separately in the form of specification functions (not shown here).

Program Function B.2 represents the subject code shown in Figure 1 on page 3. The Program Functions were then analysed against the intended function; the intended function for the entire procedure is shown in Figure 3 on page 5.

In this example, the verification discovered all of the defects mentioned so far. A further defect becomes clear when B.2 and B.1 are examined together. From B.1 it is clear that when a new PTR is OPENed that it is added to the weekid corresponding to its OPEN date. However in B.2, we see that when the PTR was already ACTED on but the release information has subsequently changed, the occurrence is deleted from the weekid corresponding to the ACTED date and not from the weekid corresponding to the OPEN date.

Even though the relevant section of code would have been exercised in both types of unit testing, this defect was not found because the weekid for the test case would have been the same for both the ACTED and OPEN dates; even though this is more probable it wouldn't always be the case. This illustrates the dependence of unit testing on selecting the right data which is made much more difficult particularly (as in this case) where the data in question (ACTED date) is not part of a condition within a decision. If the data had been part of a condition then it is more probable, but not definite, that the defect would have been discovered through unit testing.

```
/*.....INTENDED FUNCTION  for MANAGE_PCFRAISE.....*/

This routine is used to manage an  array which contains all of the
information relating to PTRs raised during each week. This array is
subsequently used to fill the file PCFRAISE TABLE. The rows in the
array should be defined according to the following criteria.

ROW   Information

 2    Number of PTRs that are still OPENed during this particular weekid
 3    Number of PTRs OPENed during this weekid which are now ACTED
 4    Number of PTRs OPENed during this weekid which are now
      CLOSED but not due to an injected fix
 5    Number of PTRs OPENed during this weekid which are now
      CLOSED due to an injected fix
 6    Number of PTRs OPENed during this weekid which are now REJECTed
 7    Total number of PTRs OPENed during this weekid which is
      the same as the sum of rows 2,3,4,5 and 6

The routine should take the new information for a PTR (established
earlier) and ensure that the changes are applied to the existing array
information. This should be done by calling a separate routine,
ADDSUB_OPERATOR, with the correct parameters; these are described in
its Intended Function. The routine ADDSUB_OPERATOR makes
the actual changes. In general, new PTR information can mean that
previous information should be deleted and new information added.
```

Figure 3. Intended Function for Sample Subject REXX Code - Input to all techniques.

Parallel Reviews

A number of REXX developers (3) with a cross-section of REXX and VM experience were requested to review the subject code in parallel with the objective of detecting the maximum number of logic defects. They were provided with the design of the reporting system and would have reviewed the code subject to a set of established REXX and VM coding standards. Apart from this, the reviewers were free to use any other defect detection methods. On the example piece of code (please refer to Figure 1 on page 3), the same 2 defects that were found under decision based unit testing were also found but no other additional defects were found on this section of code.

Reviews typically suffer from a lack of structure and can be undisciplined; they are best described as a type of *black box* activity in the sense that we seldom know how they actually are conducted as this is usually left to the discretion of the reviewer.

Overall Results

The results from the experiment have been analysed on a number of different fronts. The graphs (Figure 4 on page 7) illustrate the results for the four primary metrics. Each defect was classified on

```
/* Derived program Function B1.....*/
(PTR is OPEN)
    (PTR was previously unknown) --> +2 (open date)

/* Derived program Function B2.....*/
(PTR is ACTED)
    (PTR was previously OPEN) -->
        +?3? (open date), -2 (previous open date)
    (PTR was previously ACTED & the Release info has changed) -->
        +?3? (open date), -?3? ( previous acted date )
    (PTR was previously REJECTed) -->
        +?3? (open date), 6 (previous open date)
```

Figure 2. Derived Program Function for Sample Subject REXX Code - Verification output

completion of the entire exercise according to its probability of occurrence, the severity from 1 (high) to 4 (low) and its complexity from 1 (low) to 10(high). In addition, the probability- severity metric was defined as the sum of all the probability severity ratios and the overall complexity number as the sum of all the complexity numbers.

Verification of the REXX has proven to be very successful claiming 61 defects out of a total number of 64 that were detected by all of the techniques. The code reviews were the least successful (11 defects) with the success of the unit testing varying according to the type of unit testing conducted. The probability-severity metric underlines the fact that the verification also tended to detect the more severe and the higher probability type of defects with a value of 10.6 compared to 6.1 for multi-condition unit testing. Even though the difference in the number of defects between these two techniques was 17 this deficit accounted for a probability-severity metric of 4.5. The same trends are evidenced when we look at the complexity number for each technique highlighting the fact that the verification is better at finding the more complex defects. The time taken for each technique showed little variation except for the multi-condition unit testing which took up significantly more time.

It's of some interest to look at the defect breakdown. Each defect was classified under one of 4 headings representing the effect of the defect as follows. (Please refer to the graph Figure 5 on page 8).

- A maximum of 7 defects were due to seeds placed in the code.
- Some defects were as a result of either REXX Novalue or Syntax errors
- Other defects resulted in incorrect message handling
- The remaining defects resulted in incorrect results occurring and are difficult to classify further.

All of the techniques were reasonably successful at locating the Novalue/Syntax and the messaging defects. These would typically be classified in the *easy to find* category. However when you look at the more complex defects which sometimes gave rise to subtly incorrect outcomes, the reviews and then the unit testing and finally the verification were successively more successful at detecting them. Similarly when you look at the success at removing the seeds that were placed in the code, the same trend emerges with verification finding all 7 seeds and reviews only finding one of the seeds.

Finally, the VENN diagram illustrates the uniqueness and commonality of defects across the three most successful techniques. All of the three had some uniqueness varying from 1 for each unit testing type to a significant 15 for verification. There were 21 defects which were common to all of the three techniques.

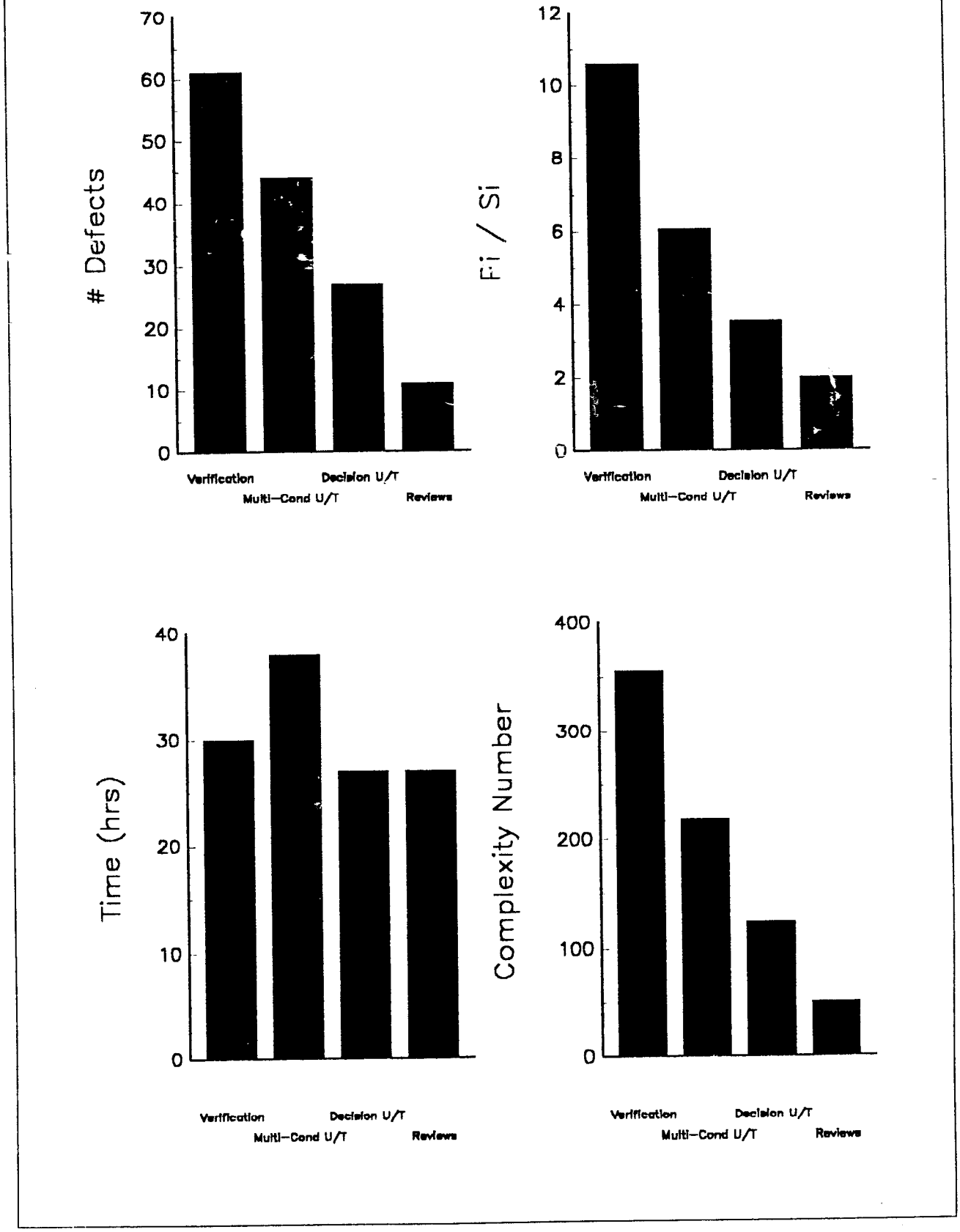
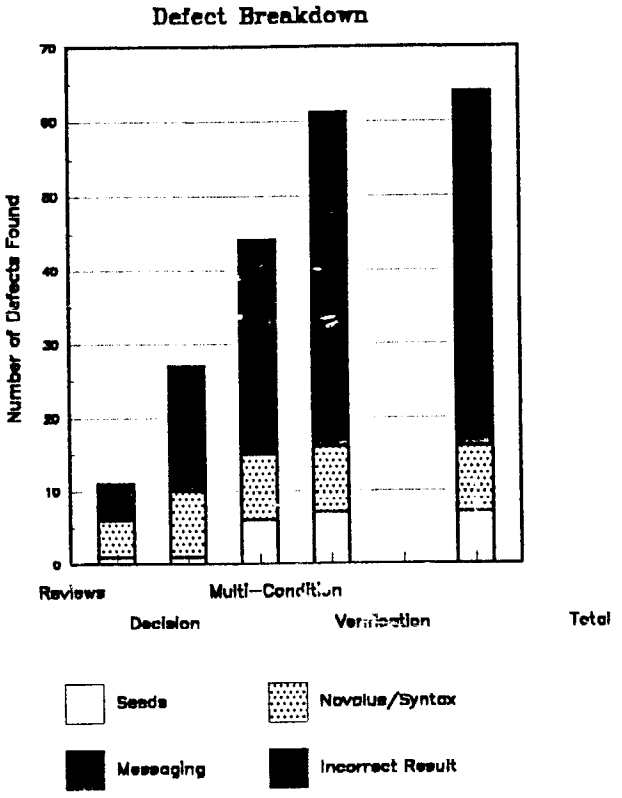


Figure 4. Results of Analysis



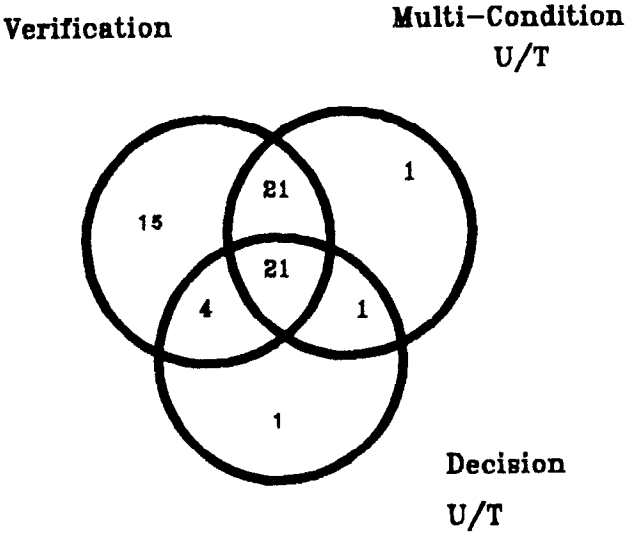


Figure 5. Results of Analysis

Concluding Remarks

Verification should be embraced as a defect removal technique as it has been conclusively shown to be very effective on the subject REXX code and is not as time consuming as expected. REXX as a language is suited to verification in the sense that it is typically easy to understand its different instructions and functions. On the other hand its loose data typing can make it difficult to fully describe the resultant state of the data. For the purposes of subsequent verification, the use of well-structured REXX code makes the task that much easier. Avoidance of REXX flow alterations like ITERATE and LEAVE makes the verification simpler. The verification exercise has also shown the importance of limiting the extent of variables to where they are needed. This can be accomplished easily with the PROCEDURE instruction which protects all existing variables and fully restores them on return from the PROCEDURE. Only those that need to be available can be done through the EXPOSE option. In fact all but one of the defects which were not found by verification were due to the fact that variables were not protected in the fashion described. If one limits the extent of variables as much as possible then the task of defining the program function for the entire program is greatly simplified and the use of the EXPOSE option on all Procedures is an easy way of knowing what variables are not protected.

Even though CMS Pipelines, which implement the pipeline concept under CMS, were not part of the subject code, their use would also appear to benefit the overall verification process in REXX. Pipelines enable complex tasks to be split into small simple robust self contained programs which would be easier to verify.

Even where one still wants to pursue the unit testing path and wants to do it in a rigorous fashion like that described for multi-condition based unit testing, it is our experience (in hindsight) that in fact the derived Program functions which were done as part of verification are an an excellent route to pursue. The program functions typically remove all redundancy, just state the code outcome, are much more understandable than the code itself and hence lend themselves to the task of defining test cases to cover the multi-condition testing rationale.

The other techniques of unit testing and reviews were successively less and less successful. The testing tends to be highly dependent on selecting the right data, cannot satisfactorily deal with missing function and lacks the intellectual control of verification. Reviews are typically black box affairs with the process of carrying out the reviews left largely up to the reviewers and if carried out should be changed to ensure that they embrace verification.

Measurement of the different techniques has provided some invaluable information and shows conclusively the effectiveness of the verification technique and not at the expense of overall productivity.

REXX FOR WINDOWS, NT, ETC.

NEIL MILSTED
iX

REXX for Windows, NT, etc.

- Windows and NT Specific
- Generic Dialogs
- Excel
- Networking

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Windows and NT Specific

- Sending keystrokes
- Clipboard
- Address DOS
- Window information and control

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Sending keystrokes with "address keys"

- Send one or more keystrokes to the active application
- Keystroke notation borrowed from Visual Basic
- Delays should be inserted between time consuming operations

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Keystroke notation

- Alt = %
- Shift = +
- Control = ^
- Esc = {esc}, Tab = {tab}, etc
- {%} = %, {{} = {, etc

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

So for example in the notepad:

- address keys "%fomyfile.txt{enter}"
- Opens the File menu with Open for the file myfile.txt

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Issuing DOS commands with "address DOS"

- DOS commands can be issued but:
  - They run asynchronously
  - There is no error or failure status returned

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Window status

- winExe() - return active window ".exe" name
- winTitle() - return active window title
- winInTaskList() - return "1" if the window is in the task list

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Window control

- winMinimize() - minimize the active window
- winMaximize() - maximize the active window
- winRestore() - restore the active window
- winClose() - close the active window
- winSwitchTo() - make the specified window active

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

```
/*
**
** Module =
**
**   ixcorp2.rex
**
** Abstract =
**
**   login to ixcorp2 with telnet from Windows
**
** History =
**
**   11-May-93 nfnm Added this comment
**
** Possible future enhancements =
**
*/
/*
 * first look for a telnet already running
 */
if winInTaskList("Telnet*") = 1 then
/*
 * if telnet's already running, make it active and exit
 */
   do
     call winSwitchTo("Telnet*")
     call winRestore
     exit
   end
/*
 * first prompt for my password
 */
password = DialogPrompt("Password?","")
/*
 * fire up telnet
 */
address dos "d:\netmanag\telnet.exe"
/*
 * wait for telnet
 */
do while winInTaskList("Telnet*") = 0
  call winsleep .1
end
call winsleep .5
```

```
/*
 * ALT-C ALT-N then the system name and wait for the connection
 */
address keys "%c%n"
address keys "ixcorp2"
address keys "{enter}"
call winsleep 1
/*
 * enter my userid
 */
address keys "nfnm"
address keys "{enter}"
call winsleep 1
/*
 * enter my password
 */
address keys password
address keys "{enter}"
```

Clipboard

- winClipboardGet() - get text contents of clipboard
- winClipboardSet() - set text contents of clipboard

Generic Dialogs

- DialogFileGet() - prompt for an input file
- DialogFileSet() - prompt for an output file
- DialogMessage() - display a message box
- DialogList() - display a list of choices
- DialogPrompt() - prompt for input

Excel

- REXX is being used with Excel on both the Macintosh and Windows
  - To extract data on the Macintosh using Apple Events
  - To extract data in Windows using the Dynamic Data Exchange

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Networking

- Data transfer
- Remote execution of REXX programs

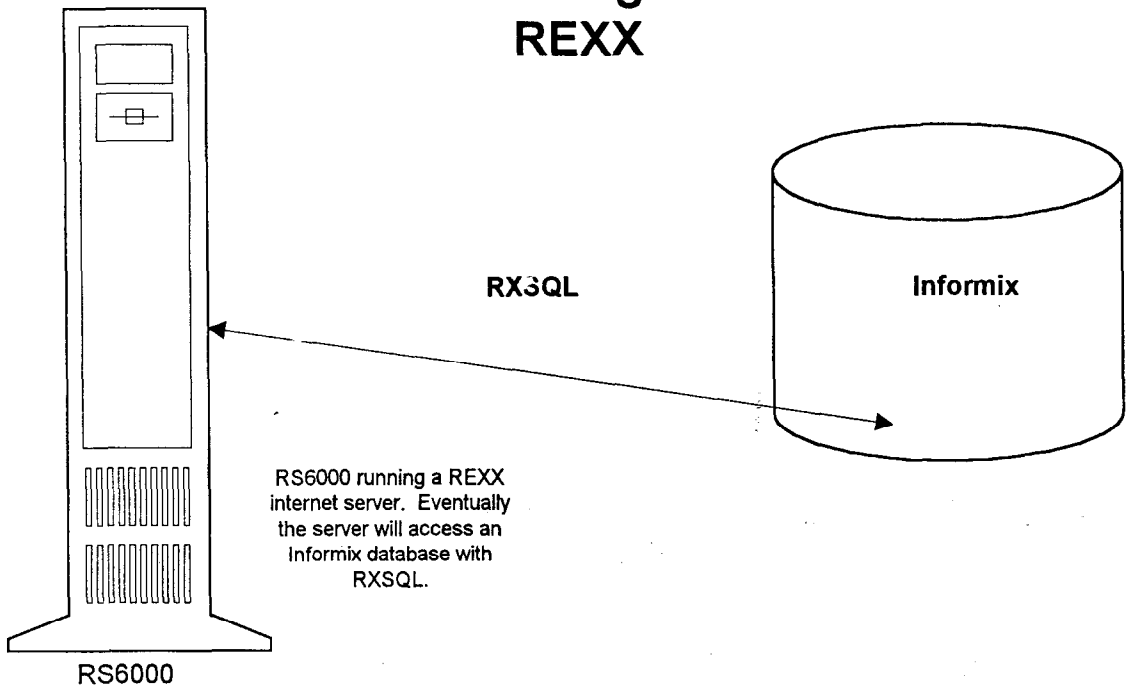
iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

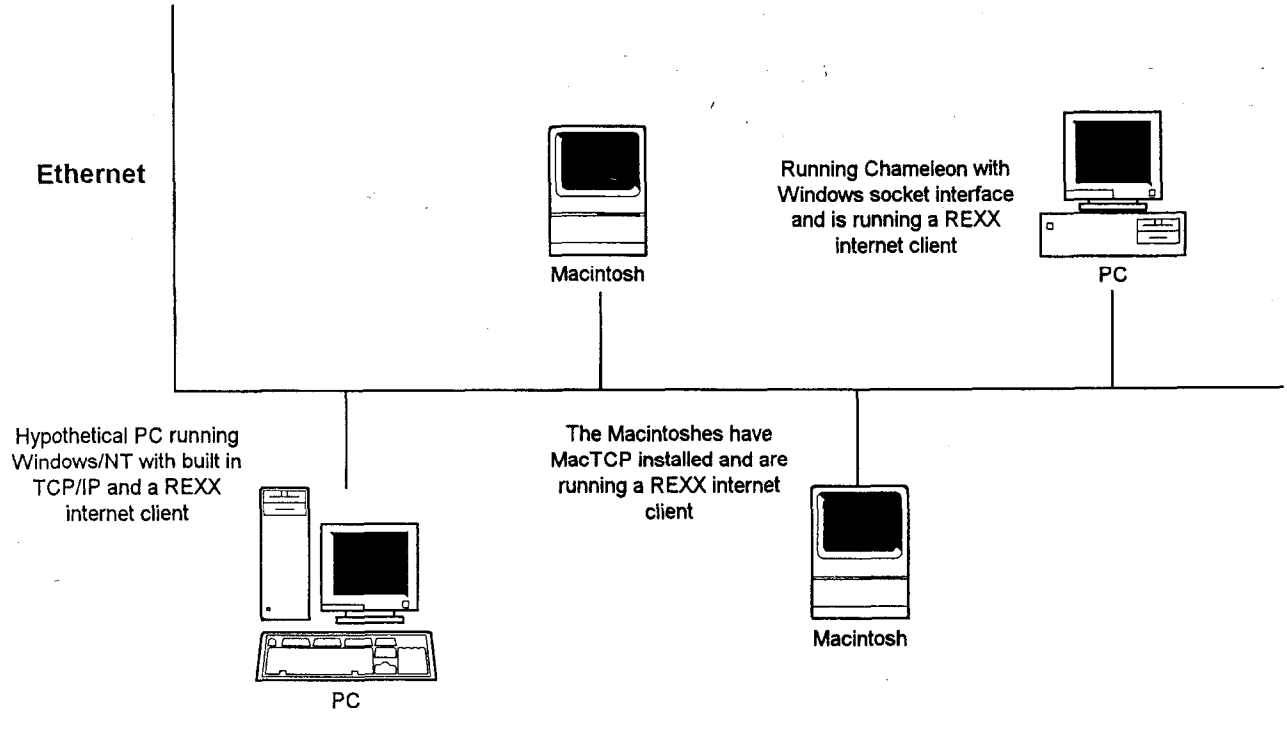
Data transfer to and from a server

- Windows and Macintosh REXX programs send and receive data through TCP/IP.
- REXX's string orientation avoids the need for an XDR

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Data Transfer Using REXX



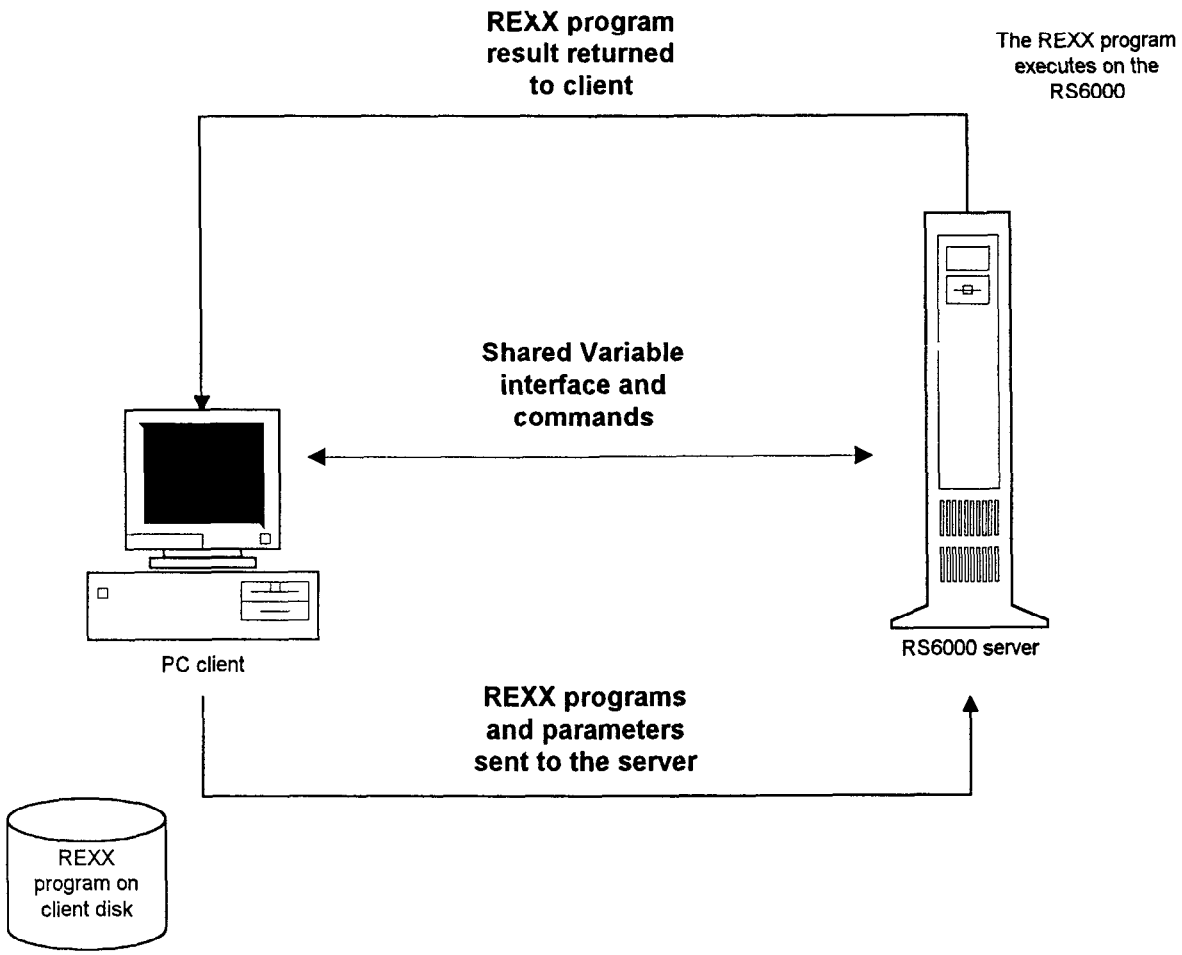


Remote execution of REXX programs on a server

- PCs and Macintoshes can send program strings to the server for remote execution
- Commands can be sent from the server back to the client
- The remotely executing program's variable pool is accessible on the client

iX Corporation, 575 W Madison #3610, Chicago IL 60661 (312)902-2149

Remote execution of REXX programs on a UNIX server



IBM Compiler and Library

Walter Pachl
IBM

IBM Compiler and Library for SAA Rexx/370

Walter Pachl

IBM VSDL Vienna
Lassallestrasse 1
A-1020 Vienna
☎(0043-1-) 211-45-4420
PACHL at VABVM1(.VNET.IBM.COM)

May 20, 1993

Chapter 1. IBM Compiler and Library for SAA Rexx/370

In a week's time, Release 2 of these products will become generally available. In the following this release's highlights and a number of related matters are described..

1.1 Highlights of Release 2

1.1.1 Support of the Interpret Instruction

When the first Rexx compiler was implemented, it was decided not to support the Interpret instruction was not to be supported. The main reasons were

- the implementation effort involved
- the relatively little use of this instruction

The compiler's Users Guide and Reference did elaborate on ways to circumvent the use of Interpret. For the most frequent use, that is assigning a value to a variable whose name is dynamically determined, a small routine was shipped with the product that could be used for that purpose:

```
varname='ABC'
Call setvar varname,expression
```

The assembler routine (RXSETVAR) used the variable pool interface to assign the second argument's value to the variable with the name passed as first argument.

This restriction was not removed with the first release of this compiler's successor product because more important user requirements had to be addressed. SETVAR was also provided for the new environment supported by that release: MVS.

However infrequent the use of Interpret may be, there is still the chance (or danger) that a package that you want to compile contains one or more programs that use Interpret and it is not always feasible to modify the programs. And there has been a steady stream of user requirements, asking for the support of Interpret. Therefore, it was finally decided to add this support to the compiler and library.

Incidentally, invocations of setvar can **and should** now be replaced by equivalent calls to the **value** built-in function which has been extended with the capability to set variables (see 1.1.3.3, "VALUE with 2 or 3 arguments" on page 3):

```
varname='ABC'
Call value varname,expression
```

(rx)setvar is still shipped with this release of the product; the chapter on how to avoid the Interpret has been removed from the User's Guide.

1.1.2 C/370 Library no longer required for compilation

The stated requirements for compiling Rexx clearly indicates that the compiler is written in the C programming language. As the cost of pre-required software must be added to that of the software a customer is interested in, this has probably kept some Rexx users from installing the compiler. Version 2 of the C/370 compiler offers the option to linkedit required library routines with the program that was implemented in C and to ship the "complete" package. After the price adjustments made with release 1 of the current product, exploiting this option is an essential step in making Rexx compilation less expensive.

In addition to reducing cost, compilation of programs became faster. (The library routines have been customized for the specific needs of the compiler.)

1.1.3 Language level 3.48

Mike Cowlishaw's "Red Book" and IBM's SAA Procedures Language Level 2 define what is called language level 4.00 of REXX. (Parse Version returns the language level as the second token.) Level 3.48 is all of 4.00 with the exception of the Rexx input/output functions. These functions have first been implemented on OS/2 and are just about to be provided on VM and OS/400. The language elements that were added for language level 3.48 are discussed in the following.

1.1.3.1 Binary strings, X2B, B2X

A literal string, immediately followed by the symbol b is interpreted as binary string. The literal string must in this case contain only the characters 0 and 1, optionally separated by blanks in certain positions.

```
x='1111 1001'b      )
x='F9'x             ) these are all the same (on EBCDIC)
x='9'               )
```

This language extension leads to a slight incompatibility. Before the introduction of binary strings, x='abc'b was the concatenation of a constant with the value of variable b. This expression will now cause an error message from the compiler (or raise the syntax condition when interpreted). The instruction x='1101'b will, unfortunately, change its semantics without being noticed. **The lesson learned**: Rexx taught me to avoid the variables I used in high school (x, y, z); now I avoid also a, b, and c.

The new built-in functions, X2B and B2X, support the conversion from hexadecimal strings to binary strings and vice versa

```
X2B('A')    --> '1010'
B2X('1111') --> 'F'
```

Conversions from character strings to binary can be achieved by a two-step process:

```
X2B(C2X('9')) --> '11111001'
```

1.1.3.2 Parsing templates +(v), -(v), =(v)

Variables could always be used for literal patterns in parsing templates. Now they can also be used as relative and absolute positional patterns.

1.1.3.3 VALUE with 2 or 3 arguments

The VALUE built-in function has been extended to allow for assigning a value to a dynamically determined variable. Additionally this function can be used to set the value of an "environment variable." (This is supported under VM beginning with CMS Release 6.) The name of the environment must then be specified as the third argument and the value of the first argument must in this case be a variable name that is valid for that environment.

1.1.3.4 Drop (ivar), Expose (ivar)

One other use of Interpret was the following **illegal** Rexx snippet:

```
a: Interpret 'Procedure Expose' v1
```

This is invalid because Procedure must be the first instruction of a subroutine, if it is used. Early CMS implementations of Rexx did not enforce this rule, an error that has since been corrected. The reason for using this construct was mainly to cast the list of variables to be exposed into a variable and to use this variable name instead of the long list. This use is now officially supported by using an indirect variable

```
a: Procedure Expose (v1)
```

For consistency, the other instruction that deals with lists of variable names, Drop, has also been extended in the same way.

1.1.4 DBCS symbols (and comments)

With the new release, pure and mixed DBCS strings can be used as symbols, that is variable names, labels, etc.

At this time the remote possibility of a bug was removed: the occurrence of '*/' or '/*' as bytes within a DBCS string used in a comment.

1.1.5 Smaller executables

The first compiler offered already significant performance improvements. However, compiled programs were, in general, larger than the source programs; sometimes significantly so. Release 1 of the IBM Compiler for Rexx/370 introduced the CONDENSE compiler option use of which results in significant disk storage and I/O savings. With release 2, another little reduction in the size of compiled programs was achieved. Compiling

REXXDX, the program that implements the CMS compiler invocation dialog, shows the following disk requirements:

```
kBytes

 160   Source program
 266   CMS REXX
 230   REXX/370 R1
 221   REXX/370 R2
  73   REXX/370 R2 (with CONDENSE)
```

1.1.6 Improved Compiler Listing

Several improvements have been made to the listing that is produced by the compiler:

- A summary of messages issued and their severity is printed at the beginning of the listing.

```
    1 message(s) reported. Highest severity code was 12 - Severe
or, the better alternative:
    Compilation successful
```

  The user can immediately check the compilation's success.

- The compiler options used are now printed in alphabetical order of their keywords proper (disregarding the NO prefix, where applicable).

  Sample Listing of Compiler Options:

```
Compiler Options

      CEXEC     (DAMEN    EXEC     A1)
    NOCOMPILE   (S)
      CONDENSE
    NODLINK
    NODUMP
      FLAG      (I)
      LINECOUNT (55)
      OBJECT    (DAMEN    TEXT     A1)
      PRINT     (DAMEN    LISTING  A1)
    NOSAA
    NOSLINE
      SOURCE
    NOTERMINAL
    NOTESTHALT
    NOXREF
```

- A list of flagged instructions is now printed at the end of the compiler listing, if applicable.

1.1.7 Support of VSE

As of this fall, Rexx will also be supported in the VSE environment. It will be possible to run Rexx programs compiled under CMS or MVS in that environment. The support for compiled Rexx will be integrated with the Rexx Interpreter on VSE.

1.2 Performance

1.2.1 Language Features

The speedup for a particular Rexx program depends on the language constructs being used in the program. The following table relates miscellaneous constructs with the performance improvement to be expected.

Programs with a lot of this ... are that much faster than the SPI	
Arithmetic operations of default precision	9.7
Constants and Variables	5.8
Ref. to built-in functions and procedures	4.9
Changes to variables' values	8.7
Assignments	25.2
Re-use of compound variables	4.4
Host commands	1.0

1.2.2 A Benchmark Program

A program that demonstrates the performance improvements is the following program that computes the number of ways you can place eight queens on a chessboard so that none interferes with the others.

```
/* Position n queens on a chess-board of n*n fields so that no queen **
** can beat any other on the board ************************************
Change Activity:
871211 PA Rexxified from the BASIC algorithm supplied by Alfred Gschwend
881104 PA give return code 0 if 92 solutions were found
910919 KH remove test code
*********************************************************************/
```

```
/*********************************************************************
 00   REM  *** Das Acht-Damen-Problem ***
 10   I = I+1
 20   D(I) = 1
 30   FOR J = 1 TO I-1
 40     IF D(I) = D(J) | ABS( D(I)-D(J) ) = I-J THEN 90
 50   NEXT J
 60   IF I<8 THEN 10
 70   R = D(1)*1E7 + D(2)*1E6 + D(3)*1E5 + D(4)*1E4 + D(5)*1E3
 80   PRINT '  ->'; R + D(6)*100 + D(7)*10 + D(8);
 90   D(I) = D(I) + 1
100   IF D(I) <= 8 THEN 30
110   I = I-1
120   IF I>0 THEN 90
130   END
*********************************************************************/
  Parse version v
  Say v

  Call time 'R'
  cs=cputime()
  nq=8                              /* set number of queens          */
  If arg(1)<>'' Then nq=arg(1)      /* allow dynamic specification   */
  n=0                               /* number of solutions           */
  x=''                              /* output buffer                 */
  i=1                               /* number of positioned queens   */
  sym='0123456789ABCDEFGHIJKLMNOPQ' /* symbols indicating row of queen*/
  ende=0                            /* end indication                */
  d.=0                              /* initialize the queen positions */
  d.1=1                             /* start at field A-1            */
  Do nn=1 By 1 While ende<>1        /* with a counter to show progress*/
    /*call out*/                    /* debugging                     */
    further=0                       /* flag indicating progress      */
    Do j=1 To i-1                   /* check if queens 1 thru i are ok*/
      If d.i=d.j | ,                /* on the same row               */
        abs(d.i-d.j)=i-j Then Leave /* or on the same diagonal is bad */
      End                           /*                               */
    If j=i Then Do                  /* queens 1 thru i are okay      */
      If i=nq Then Do               /* we have another solution      */
        n=n+1                       /* increment solution count      */
        /*call out*/                /* and show it to the user       */
        End
      Else Do                       /* not yet 8 queens              */
        i=i+1                       /* move on to next column        */
        d.i=1                       /* starting at base line         */
        further=1                   /* indicate progress             */
        End
      End
    If further=0 Then Do            /* stay on column or track back  */
      Do i=i By -1 while(d.i=nq)    /* search first column where     */
        End                         /* queen may be moved up         */
      If i<1 Then ende=1            /* all queens on row 8, so end it */
      d.i=d.i+1                     /* move up a field               */
      End                           /* of move up and/or backtrack   */
    End                             /* end of main loop              */

  Say x                             /* show buffered solutions       */
  Say n 'solutions computed'

  Say 'DAMEREXX: elapsed:' time('E') 'cpu:' cputime()-cs
Exit n<>92                          /* end of benchmark              */
```

The System Product Interpreter needs about 13 seconds to run that program (on a 9121-400). My PS/2 model 95 takes 75 seconds. The following table shows the timing of the same program with the possible combinations of compilers and run time libraries.

	COMPILE Time	EXECUTION Time		
		CMS/REXX	REXX/370 R1	REXX/370 R2
CMS REXX	0.28	1.14	1.16	1.17
	0.29	1.12	1.15	1.15
		1.13	1.16	1.15
REXX/370 R1	0.24		1.09	1.08
	0.24		1.08	1.08
	0.24		1.09	1.07
REXX/370 R2	0.21			1.11
	0.21			1.10
	0.21			1.11

1.2.3 Compiler Options

Some compiler options affect the runtime performance of the compiled programs. These are discussed in the following.

1.2.3.1 CONDENSE

The CONDENSE option causes the compiled program to be stored in a condensed format. This has the following advantages:

1. The compiled program uses less disk space.
2. Preloaded compiled program use less virtual storage.
3. Loading the program requires less I/O activity.
4. Literals in a program become illegible (and un-"ZAP"-able).

On the other hand there are a number of little disadvantages:

- There is a minimum overhead for unpacking the program at execution time.
- The virtual storage required while the program is being executed is larger.
- It takes some time to do the packing at compile time
- The CONDENSE option is mutually exclusive with the DLINK option.

1.2.3.2 TESTHALT

Compiling with the TESTHALT option causes tests to be included in the executable code that determine whether the user has attempted to interrupt the program's execution (by entering the immediate command HI, for example, under CMS). The cost of these tests at execution time is negligible.

1.2.3.3 DLINK

This option results in the most spectacular performance improvement if a large number of external function and subroutine calls are made during a program's execution. Using this option, a program and its external subroutines can be packaged into a module that uses branch-and-link instructions to invoke external subroutines. Avoiding the CMS (or MVS) search order for external routines is the reason for the dramatic performance improvement. A fringe benefit of using this technique is that changes in the program's environment (name clashes with invoked external routines) do not have any effect on the packaged program.

1.3 Testing the Rexx Compiler

Beginning with the first Rexx compiler, the CMS Rexx Compiler, a test project was set up to develop a suite of function test cases to test the language implementation as extensively as possible. Rexx was used to implement a highly automated test environment and to minimize the effort of test case writing.

1.3.1 Original Test Ideas

As any other test, the test cases for Rexx must compare the results from a language construct with the expected results. Results include

- the values of variables after executing the construct to be tested
- flow of control
- error messages
  - at compile time (for errors that are detected by the compiler)
  - at run time
- the contents and layout of compiler listings
- compiler and runtime performance
- etc. etc.

The test project was given significant lead time and could use the existing implementation, the System Product Interpreter, for testing the test cases and for constructing the test environment.

An ideal test case would consist simply of the construct to be tested, for example:

```
(5.6+1.0000000000002)
```

The expected result was either that produced by the Interpreter or that from a "pseudo-implementation" of Rexx (very much like the approach now being taken by the Rexx standardization committee).

Most of the test cases have been constructed to be self-checking. Techniques were developed to automatically handle error situations like Syntax and Novalue conditions being raised. The test environment performs the bookkeeping of successful test runs and the notification about failing test cases. With the completed test suite, human involvement is only required

- to request the execution of the test suite for a particular implementation
- to run those test cases for which human action or attention is required
- to verify, on a glance, on the morning after that no errors occurred
- or to report errors to the developers
- to extend the test suite when an error is discovered or reported or when a new test idea crosses the mind.
- and, of course, to rework the test cases for new implementations or new environments.

1.3.2 Reuse of Test Cases

The test suite has been kept alive over the past years and was extended to test all releases of the compiler in all supported environments (currently CMS and MVS, with VSE to come soon) and other Rexx implementations (such as the interpreters on most IBM platforms). This approach has not only resulted in a very high quality of the compiler products but has weeded out some errors in the other implementations.

A significant effort is, however, involved in keeping the test suite up to date for all implementations it is used for and one has to take care that the number of "generation directives" does not become excessive. Variations to be catered for include

- changes of the language

```
x='123'b  /* changes meaning in 3.38                              */
```

- implementation improvements

```
x='a'
1: x=x+1  /* is now a compiler detected error if 1 is not used  */
```

- character set

```
x='F1'x   /* is two things on ASCII and EBCDIC                   */
```

- extensions of the language

```
x=value('x',123) /* new second (and third) parameter            */
```

The forthcoming Rexx standard will, of course, be considered for further extensions and customization of the test suite.

1.3.3 Sachertorte

As it is typical for this product, the current release was "finished" quite some weeks before the committed end date. This situation was (again) exploited to our customers' advantage by exposing the compiler to a large number of IBM internal users. This time the development team had to motivate their users to try hard in finding problems, that is bugs, in this very well tested product. A contest was put in place that every person finding one or more reasonably severe errors was to be awarded with a Sachertorte[1]. The person who found the most problems is to collect the cake in Vienna where he can meet the developers and testers and can enjoy a few days in not too bad a town. Rewarding customers for problems they find is a process yet to be explored and defined; for the time being we try to deprive them the "pleasure" to find problems.

Meanwhile here is the recipe for the Sachertorte that my wife is using:

```
Sachertorte       allegedly the original recipe from Sacher.
***********       translated By Walter Pachl  900531
                  (with BJ's help - on the English side).

Ingredients

140 g butter
160 g sugar
180 g ground chocolate
  8   eggs
 30 g powdered sugar
140 g wheat flour
  1 level tea spoon baking powder
200 g apricot jam
200 g icing
----------------------------

Stir butter and sugar to get a foamy cream.

Melt the ground chocolate OVER (not in) hot
water, stir well until cooled down

Add the chocolate to the butter/sugar mixture.

Keep stirring until the mixture is thickly foamy

Slowly, by and by add egg yolks, beat heavily
until you have a chocolate cream.

Beat whites of the eggs and powdered sugar until
stiff and put this on top of chocolate cream.

Mix flour and baking powder, add on top
of all the above.

Mix it all cautiously (slowly, carefully).

Fill the dough into a cylindric cake-form
that you have coated (on the inner side :-
with aluminum foil or baking paper
(ours is about 12 inch in diameter).

Bake (use a knitting needle to check
if done - it'll come out dry then)

Let the cake cool down, take it out
of the form. Heat the apricot jam
and smear it on top and on the side
and let it soak a little into the cake.

Heat the icing in hot water and cover
the cake with it.

Note: Almonds, nuts, cream are NOT
to be used in Sachertorte.
Whipped cream is recommended with it.
```

Bon appetit.

---

[1] A famous chocolate cake produced (not only) by Hotel Sacher in Vienna.

REXXSHIP FOR OS/2

TIMOTHY SIPPLES
Center for Population Economics

Timothy F. Sipples
Center for Population Economics
University of Chicago

REXXShip for OS/2

THE REXX FOR UNIX DISCUSSION PANEL

MODERATOR: ED SPIRE (TWG)
SPEAKERS: IAN COLLIER
MARK HESSLING
NEIL MILSTED (IX)
RICK MCGUIRE (IBM)
STEVE BACHER (DRAPER LABS)
"TRANSCRIBER:" F. SCOTT OPHOF (CONSULTANT)

The REXX for Unix discussion panel
4th annual REXX Symposium
La Jolla, California
May 18, 1993

Moderator: Ed Spire (TWG)
Speakers: Ian Collier
Mark Hessling
Neil Milsted (iX)
Rick McGuire (IBM)
Steve Bacher (Draper Labs)

"Transcriber": F. Scott Ophof (cmsultant)

Note from the "transcriber":

Due to an unfortunate circumstance, there was no recording of the discussion on which to base a transcription for the Proceedings. The following thus consists of a reconstruction pieced together from memories and comments.
Appended you will find the list of subjects planned for discussion, depending on the time available and the time used by the actually discussed subjects.

Thus, though the accuracy in questionable, the intent is present. And we (speakers, moderator and "transcriber") hope the subjects specified in the "List of Subjects" will receive more attention.

Introduction:

Moderator Ed Spire introduced the speakers, probably saying something close to:

Steve Bacher, commonly known as "Batchman", is a Technical Staff person in the Computer Support group at Draper Laboratory in Cambridge, Massachusetts. He has moved over the years from straight MVS system programming to supporting both mainframe users and Unix workstation users. Steve is a very enthusiastic user of REXX and a contributor of the odd TSO/REXX utility. The Draper Consulting "CLOG" help desk facility, which Steve implemented, is written entirely in REXX. Steve is the author of the MVS NNTP and Gopher clients (not written in REXX). and was the creator of the "Zil" Lisp system for MVS (also not written in REXX).

Ian Collier worked for a year at IBM Hursley (near Winchester in the South of England), which is where his interest in Rexx originates. Ian then went to Oxford University and took a Ba (first class) from Oxford University in 1990 in "Mathematics and Computation". He is presently in the third year of a D.Phil. at Oxford in the field of "correctness-preserving transformations in action systems", which is a branch of parallel computing. Irritated for the last two years by the lack of a Rexx interpreter (or indeed any "decent" interpreter) on their SunOS system, Ian decided to write one. Unfortunately, academic work has often for long periods prevented him from working on it. So although the interpreter has been working for some time it was summer 1992 before Ian added the file I/O functions and finally released REXX/imc to the public.

Mark Hessling, currently working as Oracle DBA for Griffith University in Brisbane in Australia, started on ICLs, moved to DEC-10s and then to VM/CMS. From there he went to Unix via VMS. Mark had about 4 years REXX experience on VM/CMS including contract work in the UK. He has been working on SunOS for about the past 3 years. Mark is here at the REXX Symposium to present "The Hessling Editor", which is based on Kedit and CMS-XEDIT.

Neil Milsted, of iX Corporation, implemented a REXX for UNIX known as uni-REXX. Neil is highly active as Vice Chairman in the X3J18 REXX Standards Committee, where the effort of standardizing REXX is

taking place.

Rick McGuire is well-known as lead developer/designer of the REXX products (like the OS/2 interpreter) from IBM Endicott in New York. Rick, due to time pressures, will lead off the discussion.

Discussion:

Rick McGuire led off with an entertaining speech about the history of REXX's introduction to new platforms (starting with VM). He also spoke on common misconceptions about why Rexx is not useful on Unix. The consensus is that Rick had enough good material for a complete session of his own.

On the question of what kind of people will be using REXX in Unix, Ian Collier said:

I think that one thing we should do is give the Rexx programmer access to basic Unix system calls, such as those for dealing with sockets. If Unix hackers see that Rexx is a powerful tool for controlling Unix then they might start to pay some attention. It would be quite neat if we could implement an NNTP newsreader for Unix in Rexx – especially if it is better than "rn", or whatever. That way people will start to see how useful Rexx is.

Ed Spire asked:

Should we really give the user all the Unix system calls, including fork()?

Steve Bacher replied:

This might not be a good idea, especially for non-hackers since once you call fork() you become two people and you've got to figure out who you really are.

[Steve might also have said: "Right now, only ex-mainframers use REXX on Unix. The RS/6000 market is like that. This isn't good enough. We MUST reach the native Unix hackers."]

[Note: The next day however Ed showed that he had already given the user a fork() call in Uni-Rexx].

Mark Hessling commented:

There are 3 kinds of potential REXX users on Unix – ex-mainframers and hackers, plus folks told to use it with no significant computer training, a la PC/DOS/Windows users. This might be the largest potential upcoming market.

The stack and GUI issues were discussed in some detail. Someone seems to have said that it would be too much work to implement the Rexx stack with sockets.

Ian Collier replied to this:

I did implement the stack in REXX/imc with sockets; it allows the programmer to type "ls -al | rxstack" (where "rxstack" is a program which communicates with the stack via the socket) and have the output stacked. This mimics OS/2 behaviour, and is also vaguely similar to the CMS method. This has another benefit: you can start off a stack before calling Rexx, and then any data on the stack will persist

between invocations of Rexx.

Charles Daney asked a question about whether there was interest in REXX as an embedded macro language under Unix, and Neil said it had become easier to do. But the impression seemed to be that there wasn't a lot of interest.

Even though the panel was accorded extra time, there just wasn't enough for the issues planned. A number of points on the prepared list thus didn't get a mention, including the questions for Mike Cowlishaw... The above is all that we could piece together.

List of Subjects for Discussion:

Included are some things various speakers wanted to say on specific subjects, and indeed may even have said if that subject actually was discussed at the Symposium.

1: REXX still seems to be associated with CMS.

2 : Old world vs. New world (text mode vs. graphical interfaces)

3: A natural universal notation for capturing the output of a command.

4: Extensions to Rexx unique to Unix (regexps, RXSOCKET, . ..).
Ian Collier: It might be useful to discuss whether these extensions are to be keyword instructions, built-in functions, or library functions. I propose the latter. In this case you might not call them "extensions" at all. However, it is clearly necessary to make sure that the library for each interpreter contains the same functions.

5: Redirect/pipe into the REXX environment (vars, stack, etc.).

6: Redirect/pipe into another command (ADDRESS SHELL/EXEC/PERL/SH/...).

7: How to best integrate Rexx into the Unix environment.
Steve Bacher might have said:
We have to find those things that keep people from trying REXX (like the IBM association) and also those things that keep people from continuing to use it once they've tried it (lack of features, performance, etc.).

8: Are the (free) REXXes robust enough?

9: Way(s) of interfacing to Unix.
Steve Bacher: "We need to discuss the passing of arguments to REXX from a typical Unix shell environment. This is critical."

10: Is there any persistence?
To this Ian Collier might have said "Not in Unix there isn't. End of story...".
Steve Bacher: "Imagine providing Unix users a way to write a script that will "cd" or set environmental variables in the main shell. Of course, this requires people with smarts about

Unix internals to get involved in the REXX implementation process."

11: Macros and scripting.

12: Have any issues regarding integrating Rexx into the shell been addressed?
Steve BAcher: "Mainly the issue of argument passing, which has been ignored for far too long. (No, PARSE QUOTED won't help.) Other thoughts of a "rexxsh" are probably not viable now - REXX may not be the most pleasant INTERACTIVE environment for people."

13: What direction would the user community have for us on these issues?
Steve Bacher: "Ask them! But more to the point, have something to sell them Show them what REXX can give them that perl can't. Also show them that anything they can do in the other shell scripting languages can be done in REXX - if it can't it's probably too arcane to be bothered with. If there is anything that can't be done in REXX that is important to Unix users (like regexps), then by all means start working on integrating it into REXX."

14: Is the ANSI effort holding REXX useage back, and could this also be relevant to other platforms?

15: Value of conversion utilities (sh2rexx, csh2rexx, perl2rexx...)?

UNI-REXX

ED SPIRE
Workstation Group

uni-REXX
Rexx for Unix

Rexx Symposium

May, 1993

La Jolla, California

The Workstation Group Rosemont, Illinois

iX Corporation Chicago, Illinois

Recent uni-REXX support

TRL-II Issues

- stdin:, stdout:, stderr: named streams
- variable **sublists** in DROP and EXPOSE
- 3rd parm of **value()**
- "**b**" date format
- several obscure conformance issues

Recent uni-REXX support

irxstk, irxexcom via IPC, not linking.

EXECIO

execio {lines I *} {DISKR I DISKW} file (linenum) (((options))
– DISKR options: avoid, find, locate, fifo, finis, lifo, margins, notype, stem, strip, var, zone
– DISKW options: case, finis, margins, notype, string, strip, stem, var

GLOBALV

globalv select group
(select group) {set I sets I setp} name1 value1 . . . nameN valueN
{setl I setls I setsl I setlp I setpl} name value
{put I puts I putp I list I get I stack} name1 . . . nameN
select group purge
purge
grplist
grpstac k

Recent uni-REXX support

procedure expose for external procedures

Exposure of standard Unix system programming interfaces to Rexx:

-accept	-bind	_close
_closedir	_connect	_errno
-exit	-fork	_geteuid
_gethostbyname	_gethostid	_gethostname
_getpid	_getppid	_getservbyname
_getuid	-kill	-listen-opendir
_readdir	_recv	_regex
-send	_setsid	_sleep
-socket	-stat	_sys_errlist
-truncate	_umask	-wait
_waitpid		

```
/*
** Copyright (C) iX Corporation 1993. All rights reserved.
**
** Module =
**
**  syserr.rex
**
** Abstract =
**
**  System error routine. Display the error number and message for
** a system error.
**
** History =
**
** 07-May-93 Added this comment
**
** Possible future enhancements =
**
*/
procedure expose sial
/*
 * display error messages
 */
say "Error in" arg(2) "line" sigl
say arg(1) "error" _errno()":" _sys_errlist(_errno())
/*
 * exit for good
 */
call _exit(1)
```

```
/*
** Copyright (C) iX Corporation 1993. All rights reserved.
**
** Module =
**
**  sendbuf.rex
**
** Abstract =
**
**  Send a buffer in a length prefixed packet
**
** History =
**
** 07-May-93 Added this comment
**
** Possible future enhancements =
**
*/
sendbuf:procedure
parse arg socket, buffer
buff erlength = right(length(buffer), 4, '0')
bufferlength = right(bufferlength + length(bufferlength), 4, '0')
call send socket, bufferlenath i I buffer,,""
if sendrc c 0 then call sockerr "send"
return
```

```
/*
** Copyright (C) iX Corporation 1993. All rights reserved.
**
** Module =
**
**  recvbuf.rex
**
** Abstract =
**
**   Receive a buffer in a length prefixed packet
**
** History =
**
** 07-May-93 Added this comment
**
** Possible future enhancements =
**
*/
recvbuf:procedure
 parse arg socket
 recvrc =  recv(socket, "bufferlenath", 4, MSG PEEK)
 if recvrc < 0 then call sockerr "recv"
 recvrc =  recv(socket, "buffer", bufferlength, "")
 if recvrc < 0 then call sockerr "recv"
 return substr(buffer, 5)
```

Interprocess Communication in uni-REXX

architecture

- Before issuing a command to the OS, the language processor opens a socket and places the address of the socket in an environment variable.

- The Rexx API library used for external commands contains routines that communicate with the language processor via that socket.

- Original Rexx API library remains available for embedded applications, allowing direct access to the language processor APIs.

Interprocess Communication in uni-REXX

Applications to date:

- globalv
- execio
- rxsql (oracle)

Performance Considerations

C library bif's (sockets, multitasking, etc.) allow for similar coding in Rexx rather than C.

```
/*
** Copyright (C) iX Corporation 1993. All rights reserved.
**
** Module =
**
**   iserver.rex
**
** Abstract =
**
**   Demonstrate Open-REXX UNIX system interfaces with a client
**   server implementation. This routine is the server. It must be run
**   as root. Only one copy should run at once, or unusual things
**   may happen. To stop it, simply kill the PID displayed at start-up.
**
**   Note: The service name rexxinet must be in /etc/services.
**
**   This server simply accepts a file mask and returns all the files
**   that match it in the current directory. The server signals it's
**   done by sending "<end>".
**
**   The client and server may be anywhere on the same network.
**
** History =
**
** 07-May-93 Added this comment
**
** Possible future enhancements =
**
*/
```

```
/*
 * note the program name
 */
parse source. . pgmname .
/*
 * create the client process
 */
forkrc = _fork()
if forkrc < 0 then call syseri- "fork", pgrname
/*
 * the parent now exits, leaving only the child
 */
if forkrc <> 0 then
  do
    say "REXX daemon started: PID =" forkrc
    exit
  end
/*
 *
 */
```

```
setidrc = _setsid()
if setidrc < 0 then call syserr "setid", pgmname
call _umask(0)
/*
 * open the server internet socket
 */
socket0 = _socket(AF_INET, SOCK_STREAM, 0)
if socket0 < 0 then call syserr "socket", pgmname
/*
 * get the server structure
 */
call _getservbyname("rexxinet", "tcp", "server.")
/*
 * initialize the internet socket address structure
 */
inetsocket.sa_family = AF_INET
inetsocket.sin_addr.s_addr = INADDR_ANY
inetsocket.sin_port = server.s_port
/*
 * bind the socket to the port
 */
bindrc = _bind(socket0, "inetsocket.")
if bindrc < 0 then call syserr "bind", pgmname
/*
 * listen for connections
 */
listenrc = _listen(socket0, 5)
if listenrc < 0 then call syserr "listen", pgmname
```

```
/*
 * process client connections as they appear
 */
do forever
        /*
         * accept the client connection
         */
          socket1 = _accept(socket0, "inetsocket.", 0)
          if socket1 < 0 then call syserr "accept", pgmname
        /*
         * get the client file mask
         */
          mask = recvbuf(socket1)
        /*
         * convert the file mask to a regular expression
         */
          regex = sh2reg(mask)
        /*
         * open the current directory
         */
          dir = _opendir(".")
          if dir < 0 then call syserr "opendir", pgmname
```

```
      /*
       * send each matching file name back to the client
       */
        do forever
            /*
             * get the next file
             */
               currentfile = _readdir(dir)
            /*
             * if it's the last file, we're done
             */
               if currentfile == "" then
                 leave
            /*
             * if the file matches the pattern, send it to the client
             */
               if _regex(regex, currentfile) = 1 then
                 call sendbuf socket 1, currentfile
              end
      /*
       * send the end‘of transaction indicator
       */
        call sendbuf socket 1, "<end>"
      /*
       * close the client connection
       */
        closerc = _close(socket1)
        if closerc < 0 then call syserr “close”, pgmname
      end
/*
 * close the accepting connection
 */
cioserc = _close(socket0)
if closerc < 0 then call syserr “close”, pgmname
```

```
/*
** Copyright (C) iX Corporation 1993. All rights reserved.
**
** Module =
**
**  iclient.rex
**
** Abstract =
**
**  Demonstrate Open-REXX UNIX system interfaces with a client
** server implementation. This routine is a client. It connects with
**  an internet service named "rexxinet" and sends it a file mask.
**  The server should then respond with each file in its current dir,
**  that it matches. A buffer containing "<end>" signals completion.
**
**  Note: The service name rexxinet must be in/etc /services.
**
**  The client and server may be anywhere on the same network,
**
** History =
**
** 07-May-93 Added this comment
**
** Possible future enhancements =
**
*/
```

```
mask = "*.rex"
/*
 * note the program name
 */
parse source. . pgmname .
/*
 * get the host structure
 */
call_gethostbyname(_gethostname(),“ph.“)
/*
 * get the server structure
 */
call _getservbyname("rexxinet", "tcp", "ps. ")
/*
 * initialize the internet socket address structure
 */
sin.sa_family = ph.h_addrtype
sin.sin_addr = ph.h_addr
sin.sin_port = ps.s_port;
/*
 * create the internet socket
 */
socket = _socket(AF_INET, SOCK-STREAM, 0)
if socket < 0 then call syserr “socket”, pgmname
/*
 * connect to the server
 */
connectrc = _connect(socket,“sin.“)
if connectrc c 0 then call syserr “connect”, pgmname
/*
 * send the file mask
 */
call sendbuf socket, mask
```

```
/*
 * get the directory line
 */
say "Response from server for" mask
do forever
        /*
         * get the buffer from the client
         */
          buffer = recvbuf(socket)
        /*
         * if it's the end of transaction indicator, we're done
         */
          if buffer == "<end>" then
            leave
        /*
         * display the file
         */
         say buffer
        end
/*
 * close the internet socket
 */
closerc = _close(socket)
if closerc < 0 then call syserr "close", pgmname
```

Market Acceptance

Last year, many research purchases, few pilot projects

This year, many pilot projects, some production implementations

Unbundling Rexx from it's embedded applications has helped cost justify it's acquisition by commercial accounts

AN INTRODUCTION TO VREXX

CRAIG SWANSON
UCSD

An Introduction to VREXX

Craig Swanson
San Diego OS/2 User Group

REXX symposium
La Jolla, California
May 18, 1993

VREXX - Gateway to Graphical REXX for OS/2

is REXX-aware applications for OS/2 2.0 and 2.1 come to market, the system scripting abilities of the language will allow OS/2 users to write a REXX programs to tie together multiple applications to perform complex actions. For example, a REXX script for OS/2 might allow a user to double-click an icon in the Workplace Shell to start a telecommunications program, dial up a remote service such as CompuServe, retrieve stock prices-and news regarding a stock portfolio, and then take that information and send it to a spreadsheet to create new stock trend graphs and update the current value of the portfolio. But even without using such REXX-aware programs as *Borland ObjectVision for OS/2*, REXX programs for OS/2 can have a graphical user interface. VREXX, short for Visual REXX for Presentation Manager, was written by Richard B. Lam of the IBM T.J. Watson Research Center to allow REXX for OS/2 to have a Presentation Manager user interface complete with windows, dialog boxes, text (even in varied fonts and colors) and graphics without the programmer having to learn the intricacies of writing PM programs in C or C++ programming languages. VREXX can be found in the archive called `VREXX2.Z IP` which is available on *OS/2 Connection* bulletiu board in La Jolla (619-558-9475) and many other bulletin boards and FTP sites. It is distributed under the IBM Employee Written Software plan that permits pmgrams to be released free of charge but without any guarantee of product support from IBM.

Simple VREXX Calculator Example

We'll examine a short VREXX program to show the essentials of using the package. Take a look at the program listing labelled VCALC. `CMD`. Please note that the line numbers are not really part of the program but are simply there to make it easier to point out the interesting parts of the program. The first six lines of the program are comments. As you know, every REXX program must start with a comment. I added a few others to note what the program is supposed to do and when it was written. Line 7 is the first `that does any real work. The CALL` instruction transfers control oi the program to a subroutine provided by REXX for OS/2. This subroutine is named `RxFuncAdd` aud will add a new function to the REXX environment called `VInit`. The `VInit` function is found in the `VREXX.DLL` file andiusidethat file has the name `VINIT`. Then online 8, the `VInit` function is called to add all the other VREXX functions to the REXX environment. If it fails, the value `"ERROR" is stored in` the variable "initcode" and the `SIGNAL VREXXCleanup` instructionis run, thus transferring control of the program to code that will shut down VREXX and terminate the program.

Normally the `VInit` call should not fail, so in this case lines 10 and 11 tell the program to jump to the `VREXX-Cleanup` label if the program fails or is asked to end for some reason. Through experimentation, I found that line 15 is required to handle cases where the user types in a bad expression like `"5 / 0"` which causes a divide by zero error. REXX considers this a syntax error. When an error `likethis` happens, `VCALC.CMD assumes it` isbecausethe user made a mistake and then jumps to a block of code that will tell the user a bad expression was entered.

So far the program has set up the REXX environment to permit the use of VREXX. Lines 21 to 23 specify the title for the input window, its width in characters, and the type of buttons it should have. For some reason numbers must be used for button types and the numbers are not very well documented, possibly because VREXX is freeware. I figured out which number to use by examining the sample programs that came with `VREXX2.ZIP`. Lines 28 and 29 set up the set of strings that will be used to prompt the user for input Stem variables are used for this and the variable ending in `".0"` tells VREXX how many prompt strings to expect starting with the one ending in ".1". The variable ending in `".vstring"` is used to specify the initial string `displayed` in the input box entry field. For this program, I didn't want there to be any text in the entry field at first, so the two adjacent double quote marks are used to indicate an empty string. Line 33 finally displays the input dialog box and waits for the user to press the OK or CANCEL buttons. The name of the button that was pressed is stored in a variable `named "but ton"` and the user's input is stored back into `"prompt.vstring"` which on line 35 is then copied into the variable `expr`.

Line 37 checks to see if the OK button was pressed. If it was, then lines 38 to 47 evaluate the expression usiug one `Of the more unique features` of REXX, the `INTERPRET` instruction. The answer is stored in the variable named `"result"` and finally displayed on the screen in a message box that will be displayed until the user clicks on the OK button. Then the program jumps `to the InputLoop`

label to get the next expression from the user.

If line 37 decided that the OK button had not been pressed, the THEN clause would not have been run and instead the next instruction run would have been on line 53. The "CALL VExit" instruction tells the VREXX code to shut itself down. Finally, line 54 terminates the REXX program If you do not do a "CALL VExit" before ending a VREXX program, there is a program file named VREXX.EXE that is left running. Until that program is terminated, other VREXX programs will not be runnable from the session where you started VCALC.CMD.

You may be wondering that if line 54 terminated the REXX program, why are there lines after it? I decided to put the block of code to handle expression errors after the EXIT instruction. Since this block of code is jumped to because of the SIGNAL ON instruction on line 15, it is OK for it to be after the EXIT instruction. Lines 57 to 69 merely display a message box telling the user that the expression typed was bad. After the uses clicks on the OK button in the message box, then the SIGNAL Inpu tLoop instruction causes the program to loop back to get more input.

VREXX has a lot of other abilities that I haven't cov ered, but this program illustrates the basics of calling VREXX functions that you'll need to do anything more complicated. VCALC.CMD may not very useful as a tool, but it was a helpful exercise for me to leam the basics of VREXX by writing a program that accomplished something.. If you run OS/2, type in the program and try it out. If you don't want to retype if you can get a copy of VCALC.CMD in the electronic version of the March 1993 issue of the San *Diego* OS/2 Newsletter which is available as SDIN9303 .ZIP on *OS/2 Connection*. VCALC.CMD is included inside the ZIP archive file.

How VREXX Works

If you are already familiar with OS/2 programming, you might know that REXX programs are usually run by the CMD.EXE command line interpreter using various DLL files stored in the \OS2\DLL directory such as REXX.DLL and REXXAPI.DLL. You might be wondering how a text mode program like CMD.EXE can display PM windows and dialog boxes. The answer is it can't, at least not on its own.

By using the PSTAT, PSPM2, and OS20MEMU tools while running a VREXX program, I've been able to determine that the VREXX program is actually functioning as a client of a PM program that it has spawned to manage the display. When the VInit() function is executed in the REXX program, it appears that a shared memory region named "\MEM\VREXX\V#" (where # is a number representing the particular VREXX program running) is created. Then a PM program named VREXX.EXE is spawned. The CMD.EXE and VREXX.EXE programs communicate via this shared memory region. This allows the client REXX script being run in the CMD.EXE process to request PM services to be provided by the VREXX.EXE process it is using as a server.

VREXX.EXE has two threads. I'd speculate that one of these threads contains the main PM message loop and that the other communicates with the REXX program. It uses the services of two DLL files supplied with VREXX which are DEVBASE.DLL and VREXX.DLL. DEVBASE.DLL appears to be more than just a supporting library for VREXX as inside it has text strings such as "OS/2-AIX Development Base" and what look to be Adobe PostScript commands. What else it might do it unclear to me.

VREXX.DLL appears to be code used by both the CMD.EXE and VREXX.EXE processes. If you kill one of these processes without killing the other, the remaining process appears to be destabilized so it crashes with a protection fault. Also if you do not do a "CALL VExit" in your REXX program, the CMD.EXE process cannot run additional VREXX programs and in fact may disappear entirely in what also appears to be the result of a protection fauii Lastly, it appears that there is a limif on the number of VREXX programs that can be run at one time. I was not able to run more than two at once. Tryiug to start additional VREXX programs resulted in the command line sessions disappearing, probably due to a protection fault while running in the VREXX.DLL code. I do not see any reason why such a low limit is required by the approach that appears to be used to make VREXX function, so perhaps this was an oversight in the original code. After all, it is a 1.0 release. Or maybe something is not being cleaned up properly due to the way VREXX is architected using DLL's and shared memory. While experimenting with VREXX, I've noticed symptoms such as the second of two concurrently executing VREXX scripts not starting up consistently which indicate that the latter might be what is really happening.

Helpful Tools for VREXX Programmers

Since sometimes things go wrong when writing a VREXX program (after all, programmers do make mistakes), it is possible that you- will leave VREXX.EXE processes running when a VREXX program stops with an error before executing "CALL VExit" to terminate the VREXX environment nicely. Therefore I'd recommend that you download a pair of files from *OS/2 Connection* called PROCS21.ZIP and KILLEM21.ZIP. These programs will let you list running processes to find the process ID number of VREXX using the "procs" program and then let you kill the VREXX program using "killem" followed by the process ID number of VREXX. The archive PSPM2.ZIP contains a single PM program to perform the same functions if you prefer graphical user interfaces.

I hope this introduction to VREXX has given you a starting point to experimenting with graphical REXX programs. If you have questions or feedback for me, you can send them to "Craig_Swanson@f354.n202.z1.fidonet.org" on Internet. Please include a reply-to address in your message in case your address is stripped by any mail gateways.

VCALC.CMD

```
 1: /* VREXX simple calculator program ● /
 2: /* San Diego OS/2 Newsletter        */
 3: /* March 1993 edition              */
 4:
 5: /* Program Initialization */
 6:
 7:   CALL RxFuncAdd "VInit", "VREXX", "VINIT"          /* Add VInit function to attach to VREXX */
 8:   initcode = VInit()                                /* Initialize VREXX */
 9:   IF initcode = "ERROR" THEN SIGNAL VREXXCleanup    /* Exit program if VInit() failed ● /
10:
11:   SIGNAL ON FAILURE NAME VREXXCleanup               /* If the program fails or stops for any    */
12:   SIGNAL ON HALT NAME VREXXCleanup                  /* reason, the VREXX cleanup must be done   */
13:                                                     /* in order to leave VREXX in a known state */
14:
15:   SIGNAL ON SYNTAX NAME SyntaxError     /* Syntax errors should only be triggered by bad */
16:                                         /* user input, so when one happens, tell the user ● /
17:                                         /* the math expression was bad.                   */
18:
19:/* Main Program ● /
20:
21:   windowTitle = "VREXX Calculator 1.0"  /* Title of input window ● /
22:   dialogWidth = 50                      /* Input dialog should be 50 characters wide ● /
23:   buttonType = 3                        /* type 3 means use OK and CANCEL buttons */
24:
25:
26: InputLoop:                              /* Label used for looping back to get more input ● /
27:
28:   prompt.0 = 1                                                  /* Only one prompt string */
29:   prompt.1 = CENTER( "Enter a math expression:", dialogWidth )  /* This is the prompt string. ● /
30:   prompt.vstring = ""                                          /* No default expression */
31:
32:   /* Get input from user ● /
33:   button = VInputBox( windowTitle, prompt, dialogWidth, buttonType )
34:
35:  expr  = prompt.vstring                 /* Store the expression the user typed */
36:
37:   IF button = "OK" THEN DO              /* If the OK button was pressed */
38:       INTERPRET "result =" || expr    /* evaluate the expression */
39:
40:       text.0 = 1                        /* and then show a one-line result ● /
41:       text.1 = result                   /* in a message box on the screen */
42:
43:       /* Show the message box */
44:       CALL VMsgBox "Result of <" || expr || ● >", text, 1
45:
46:       SIGNAL InputLoop                  /* Go get the next expression */
47:   END
48:
49:   /* The OK button wasn't pressed so exit the program. ● /
50:
51: /* Program Exit */
52: VREXXCleanup:
53:    CALL VExit                           /* Clean up the VREXX resources */
54:    EXIT                                 /* Terminate the program        */
55:
56:
57:/***** ERROR HANDLER ● ****/
58:
59:/* Display an error message ● /
60: SyntaxError:
61:   SIGNAL ON SYNTAX NAME SyntaxError     /* Reinstall error handler ● /
62:
63:   text.0 = 2                           /* Show a two line display */
64:   text.1 = "Bad expression:"           /* of the mistake          */
65:   text.2 = "  " expr
66:
67:   CALL VMsgBox "Error", text, 1        /* Show the message box with just an OK button */
68:
69:   SIGNAL InputLoop                     /* Go back and get more input */
```

THE CONTROL & ACCOUNTING SYSTEM FOR THE COMPUTER CENTER

V. O. KROUGLOV AND S. A. GOLOVKO

The Control & Accounting system for the Computer Center

(the REXX language and the management of host computers)

V.O.Krouglov, S.A.Golovko

Computer Center of the Institute for Low Temperature

Physics & Engineering of the Ukrainian Academy of Science,

47 Lenin Avenue, 310164 Kharkov, Ukraine

In 1985 our Computer Center has received - at last - a normal computer - ES-1045 (a Soviet analog of IBM/370 Model 155: 8 MB RAM, 800 MB HD). After overcoming the long resistance of the people used to work under OS/360, it has been decided to perform all the future tasks only under the VM/370 environment. The majority of the problems being solved at our institution are purely computational, for most of which there has been some source code already, which simplifies considerably their migration to the new operating system. For the rest, by luck, substitutions were found to work under CMS. Almost immediately thereafter, we got VM/SP Release 3.x accompanied by REXX, which soon became (along with XEDIT) our main tool of system support.

One of the main problems we had to deal with, was the absence (at our disposal) of satisfactory systems for batch processing and systems of accounting (and limitation) of accessible computing resources. Soon enough it became clear that it was reasonable to restrict the usage of dialog resources of CMS to problems of editing and debugging of programs, concentrating the handling of resource-consuming tasks in several batch virtual machines.

The main principles on which the new system was based, were as follows:

1. All the accessible time was split into 3 approximately equal parts:

1.1. Dialog tasks (editing and the debugging of programs); they require personal work of the user at the terminal, mainly in the daytime.
1.2. Small tasks (up to approximately 20-30 minutes of CPU time).
1.3. All the rest.

According to this each user had monthly limits for paper usage, terminal and processor time (1.1). He had also the possibilities to send own tasks into a batch, for the processing in one of 2 queues: priority (we could guarantee the consuming time for each; this time was accounted weekly) and usual (the rest of tasks or tasks with unpredictable time for processing by system).

2. All the users were divided into 3 categories:

2.1. Ordinary users (usual limits on all resources);
2.2. Privileged users (twice the usual limits);
2.3. Super users (unlimited usage of resources).

The privileges were established by the administration, either on the constant basis or temporarily: an ordinary user could always get arbitrary resources but had to apply for them again in a month. The number of category 2.2 and 2.3 users was under strict control.

The batch processing was performed by several constantly functioning batch virtual machines automatically started (and stopped) by the described system according to “scenarios” depending on

- requested resources (estimated time, RAM, disk storage size);
- time for start and/or stop (weekdays and weekends differed).

The task became known to the system only after having been sent to iit by the user. The latter could send in, return back and delete (his) processed task, move it from one category to another, change the order of or establish links between his tasks. The order of sending the tasks to a queue did not determine the order of their processing - the system intended for processing the

task of the user possessing the minimal value of

already-used-time + estimated-time.

Usually during the daytime the tasks requiring not more than 20 minutes were served; longer tasks were postponed till the nighttime and the weekends. In the daytime shorter tasks were served first, in the nighttime - longer ones (everything was determined by an easily modifiable scenario). (Such a rule is excused certainly by non-sufficient reliability of the Soviet computing equipment).

All the batch virtual machines had (VM) priority higher than those dialogue VMs. Each user having already exhausted his limit obtained priority 99 upon entry. Such a system of controlling the priorities excluded monopolization of the computing resources by single persons.

Besides all above mentioned, the system automatically compiled daily, weekly, monthly and yearly reports on usage of machine resources, producing the bills for the payment.

The system is practically completely written using the mixture REXX + XEDIT and consists of several interacting (by exchanging the messages) virtual machines:

- PROGOPER (programmable operator). From VM/SP point of view, this is the main system operator. Here, filtration of messages is performed (to the usual operator only messages requiring human intervention are handed).

- BATCHx (batch virtual machines). These are usual VMs possessing in the system directory the CMSBATCH parameter. They receive subsequent tasks from the manager of batch processing, pass the results to the user and inform the system about termination of the task and the time used. To achieve the reliable work of these VMs, it was necessary to modify the DMSBTP module (by luck, VM/SP was supplied with source code).

- MANAGER (manager of batch processing). This VM receives tasks from the users, controls the business of the batch VMs, entry and exit of the users from the system, performs current accounting of the usage of computer's resources. This VM controls the order of work of the rest of service VMs (in case of hangs of certain VMs, it performs their reboot; if this reboot is unsuccessful, the ordinary operator is informed and the whole system is halted). This VM controls the activity status of DISKACNT.

- DISKACNT (accounting subsystem). This VM manages all the statistical information, automatically producing reports and answering the queries of the the rest of service VMs. Also makes cleaning of the out-of-date spool files and converts huge volume of accounting records into intermediate data, well-suitable for cumulative processing and report generating.

Besides the above, to the system are linked also VMs DIRMAINT and DATAMOVR. Their functions are standard, but the system controls their activity.

The above described system almost completely frees the operator from the routine tasks, is able to function without anyone's interference for months and utilizes about 20-30 minutes of processor time per day. After the appearance in VM/SP Release 5 of a feature of protected execution (CONCEAL) the consummation of the system functioning were reduced in three times.

As it seems to us (especially when reading complaints of the users on RAM deficit of 128 MB size and more) the absence of a feature for batch processing

in the systems like OS/2 2.x and AIX is a mistake of the manufacturers, and the facilities of system control depend, of course, on it itself as well as on the level of its integration with a REXX-type language. VM/SP is a beautiful, well-balanced system, and we believe in the appearance of analogous features in our next favourite - OS/2 2.x.

Jose Aguirre
3704 Edgar Park
El Paso, TX 79904
71407,441@Compuserve

Steve Bacher
Draper Laboratory MS 33
555 Technology Square
Cambridge, MA 02139
617-0258-1525
seb@draper.com

Mark Baker
12200 Montecito, Apt. K-205
Seal Beach, CA 90740
310-593-8838

Gary Brodock G09/16-4
P. 0. Box 8009
Endicott, Ny 13760
607-752-5134
brodockg@gdlvm7.vnet.ibm.com

Martin Brown
195 Bender Circle
Morgan Hill, CA 95037-3535
73537,2143@compuserve.com

Merrill Callaway
Whitestone
511-A Girard SE
Albuquerque, NM 87106
505-268-0678

Robert Clay
1071 6th Ave. #183
San Diego, CA 92101
Robert_Clay@f1808.n202.z1.fidonet.org

Ian Collier
The Queen's College
High Street, Oxford
0X1 4AW, England
+44-865-727940
imc@prg.ox.ac.uk

M.F. Cowlishaw
IBM UK Laboratories
Hursley Park
Winchester, SO21 3AL England
mfc@vnet.ibm.com

Barbara Cunningham
8 Country Club Drive
Carmel Valley, CA 93924
3404p%navpgs.bitnet@princeton.pucc.edu

Cathie Burke Dager
Stanford Linear Accelerator Center
P.O. Box 4349, MS 97
Stanford, CA 94309
cathie@slac.stanford.edu

Charles Daney
Quercus Systems
P.O. Box 2157
Saratoga, CA 95070
408-867-7399 (-REXX)
75300.2450@compuserve.com

Chip Davis
Amdahl Corp.
10420 Little Patuxent Parkway
Columbia, MD 21044-3598
410-992-0090
chip.davis@amail.amdahl.com

Daniel Duffin
Suitable Alternatives
4655 Old Ironside Drive
408-727-3142
Santa Clara, CA 95054

Forrest Garnett
2500 Huston Court
Morgan Hill, CA 95037
408-284-0295
garnett@vnet.ibm.com

Paul Giangarra
IBM, MS 1510
1000 NW 51st Street
Boca Raton, FL 33429
ppgx@bcrvmpc1.vnet.ibm.com

Gabriel Goldberg
Computers and Publishing, Inc.
13382 Brookfield Court
Chantilly, VA 22021
703-506-1125 x237 703-506-0936 (FAX)
gabriel.goldberg@permanet.org

Michael Golding
IBM Santa Teresa Lab (L74/F247)
555 Bailey Ave.
San Jose, CA 95141
415-463-3569
golding@stlvm22.vnet.ibm.com

Linda Suskind Green
IBM (G98l6C12)
P. 0. Box G
Endicott, NY 13760
607-752-1172
greenls@[illegible]dlvm7.vnet.ibm.com

Eric Giguere
Watcom Inc.
415 Phillip St.
Waterloo, Ont.
Canada N2L 3X2
519-886-3700
giguere@csg.uwaterloo.ca

John W. Hambidge
MAC 2001-028
3440 Flair Drive
El Monte, CA 91731

Paul Heaney
Consultant, Deiphi Software Ltd.
Flemming Place
Dublin 4, Ireland 602877

Mark Hessling
ITS, Division of Information Services
Griffith University
Nathan QLD 4111 Australia
M.Hessling@gu.edu.au

Richard Hoffman
11400 Burnet Road
Austin, TX 78746
ubiquity@ausvm1.vnet.ibm.com

David Hock
Hockware
104C Fountain Brook Circle
Cary, NC 27511

Paul Holbrook
IBM, Skill Dynamics
1503 LBJ Highway
Dallas, TX 75234
pholbrook@vnet.ibm.com

Tahereh Jafari
Univ. of Houston Computing Center
4213 Elgin St.
Houston, TX 77204-1961
713-743-1532
jafari@uh.edu

Michael Johnson
Relay Technology, Inc.
1604 Spring Hill Road
703-506-0500 703-506-0510 (FAX)
Vienna, VA 22182

Amir Kolsky
IBM, T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
amir@vnet.ibm.com

Luc Lafrance, Simware
2 Gurdwara Road
Ottawa, Ontario
613-727-1779
Canada K2E 1A2

Linda Littleton
Pennsylvania State University
214 Computer Building
University Park, PA 16802
lrl@psuvm.psu.edu

Dr. Brian L. Marks
354 Hursley Road
Chandlers Ford
Hampshire
Engiand SO5 1PL
44-703-253709
100010.664@compuserve.com

Todd McDaniel
165 Perry St. Suite 1A
New York, NY 10014
70012.2003@compuserve.com

Rick McGuire
RR. 1, Box 164P
Brackney, PA 18812
rick mcguire@vnet.ibm.com

Pat Meehan
IBM Ireland Limited,
2 Burlington Road
Dublin 4
Ireland
meehanp@gfdvm2.vnet.ibm.com

Neil Milsted
iX Corporation
575 W. Madison, Suite 3610
Chicago, IL 60661
76050.3673@compuserve.com

Bill Mueller
Source Line Software
7770 Regents Road, #113-502
San Diego, CA 92122

Elliot Nadel
Multiversal Enterprises
738 Laguna Seca Ct.
San Jose, CA 95123
408-363-8888

Simon Nash
IBM U.K. Lab Ltd.
Hursley Park
Winchester, England S021-2JN
nash@vnet.ibm.com

Robert O'Hara
Microsoft Corp.
One Microsoft Way
Redmond, WA 98052
206-936-2159
rohara@microsoft.com

Lincoln Ong
Stanford University
Pine Hall
Stanford, CA 94305-4122
415-723-9112
lso@jessica.stanford.edu

Walter Pachl
Lassallestrasse 1
A-1020 Vienna
+43-121145-4420
pachl@vabvm1.vnet.ibm.com

Robert Page
IBM, Usine de Fabrication Bromont
23 Boulevard de L'Aeroport
Bromont, Que. JOE 1LO Canada

David Robin
Hewitt Associates
100 Half Day Road
Lincolnshire, IL 60069
708-295-5000
derobin@halinva1.ibmmail.com

Frank Rothacker
Stanford Linear Accelerator Center
P.O. Box 4349, MS 88
Stanford, CA 94309
frank@slacvm.stanford.edu

Pat Ryall
1124 Amur Creek Ct.
San Jose, CA 95120
408-974-7354
ryall@applelink.apple.com

Jukka Saekkinen
IBM Education Center
P.O. Box 265
Fin-001001 Helsinki, Finland

Timothy F. Sipples
Center for Population Economics
1101 E. 58th Street
Chicago, IL 60637
sipl@kimbark.uchicago.edu

Ed Spire
The Workstation Group
6300 N. River Road
Rosemont, IL 60018
ets@wrkgrp.com

Craig Swanson
9260 Regents Road
Apt. K
La Jolla, CA 92037
craig_swanson@f354.n202.z1.fidonet.org

Anh Te
Towers Perrin
1500 Market St.
29th Floor
Philadelphia, PA 19102
215-246-7147

Jay Tunkel
IBM Corp-1512
1003 NW 51st Street
Boca Raton, FL 33429-1329
407-443-5955
tunkel@bcrvmpc1.vnet.ibm.com

Melinda Varian
Princeton University
Computing & Info. Technology
87 Prospect Ave.
Princeton, NJ 08544
609-258-6016
maint@pucc.princeton.edu

Larry Wall
4920 El Camino Real
Los Altos, CA 94022
415-961-9500
lwall@netlabs.com

Zvi Weiss
IBM, T.J. Watson Research Center
30 Saw Mill River Road
Hawthorne, NY 10532
914-784-6269
zvweiss@ibm.watson.com

James Weissman
Failure Analysis Associates
149 Commonwealth Drive
P.O. Box 3015
Menlo Park, CA 94025
415-688-6737
jwh@cup.portal.com

Bebo White
Stanford Linear Accelerator Center
P.O. Box 4349, MS 97
Stanford, CA 94309
bebo@slac.stanford.edu

ANNOUNCING

The REXX Symposium for Developers and Users

Boston, MA
April 25 - 27, 1994

- **Meet the developers of the REXX implementations currently available on a wide variety of computing platforms and operating systems.**
- **Learn about current research and development projects in REXX.**
- **Be among the first to learn of new products developed for and in REXX.**
- **Learn the latest programming tips and techniques from the REXX pioneers and an international body of REXX enthusiasts.**

For further information, or to participate as a speaker or panelist, contact:

Cathie Burke Dager
(415) 926-2904
cathie@slac.stanford.edu
FAX: (415) 926-3329

Forrest Garnett
(408) 996-4089
garnett@vnet.ibm.com
FAX: (408) 997-4538

Bebo White
(415) 926-2907
bebo@slac.stanford.edu
FAX: (415) 926-3329