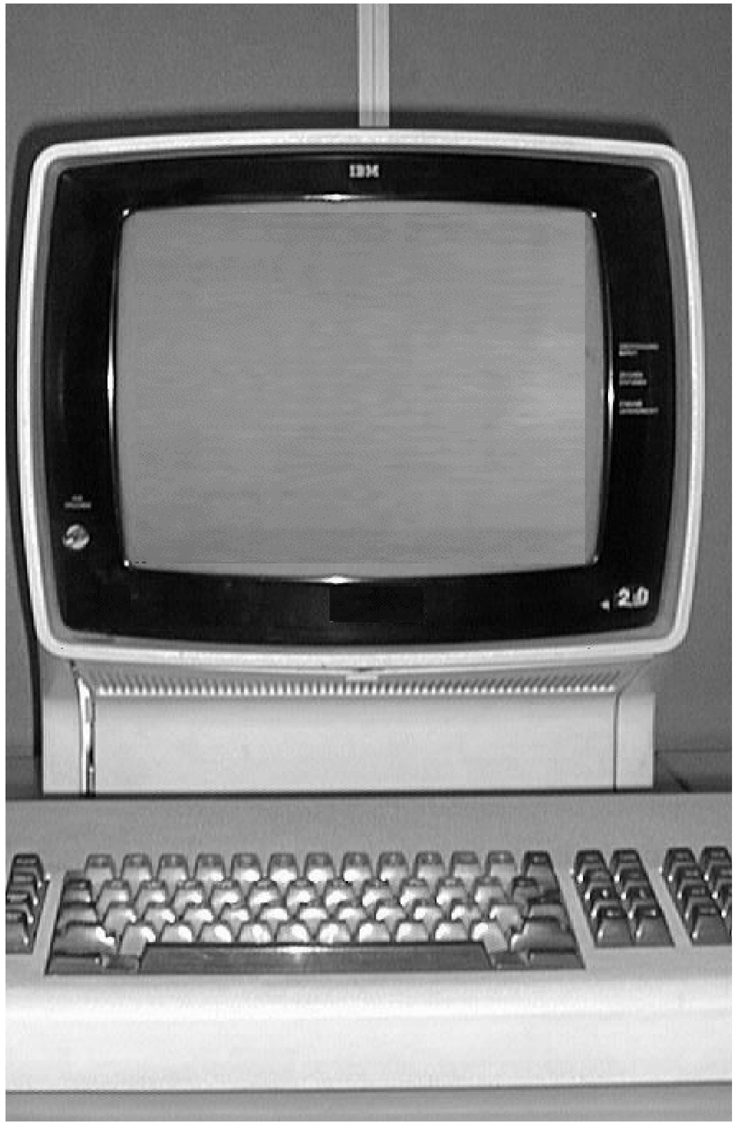
# VM/370 Interfaces for REXX



## REXX and VM started as partners from the beginning

REXX provided VM with a structured programming language for

- Command procedures
- Editor subcommands
- Prototyping
- Personal computing

IBM had to use a mouthful of a name for some reason

- The Restructured Extended Executor Language

It almost immediately replaced the older EXEC and EXEC2 in the VM community

I loved this quote: IBM said "EXEC2 is intended for manipulating English-like words as they appear in computer command languages".

REXX was a complete and fully capable programming language able to both replace EXEC2 capabilities and to promote writing complex applications

REXX and VM started as partners from the beginning

## REXX and VM started as partners from the beginning

The REXX language needed VM services to connect with the environment. Without these services, REXX connected directly only to a person or to the stack.

- Console input
- Console output
- The program stack which was accessed with console interfaces

VM provided all the interfaces I am going to talk about today in the same VM release that introduced REXX

VM and REXX complimented each other to produce a facility that was very much more than the sum of its parts.

## VM Versions

- VM/370
  - In use from 1972 to 1979. Free from IBM.
- VM/SP
  - In use from 1980 to around 1985 (depending on when customers changed over to 31 bit XA architecture)
  - EXEC2 processor introduced in Release 1
  - REXX introduced in VM/SP Release 3 in 1982 (or 3?)

## Why Use VM/370?

- VM/370 is the last free version of VM
  - IBM initially charged for their computers but provided system software at no cost
  - VM/SP was available in source code but … It had to be purchased from IBM and access was controlled by contracts.
  - VM/SP and later VM versions are unavailable to hobbyists
- VM/370 has been enhanced by Hercules emulator users
  - The community has gathered a large collection of software and packaged it with a VM system.
  - REXX first became available on VM/370 over a decade ago.
- All the source code is there
  - If it doesn't do what we need, we can change it
  - Simplest version of VM. Easiest to modify.

## Initial BREXX Implementation on VM/370

- A small team of experts built the VM/370 Sixpack Version 1.2 and released it in 2010. Bob O'Hara, Paul Edwards, and Dave Wade enhanced the earlier version developed by Andy Norrie and Paul Gorlinsky
- Many popular utilities and language compilers were included
- The Gnu C Compiler was added with a traditional runtime library.
- The team built a CMS native runtime library. This overcame memory management problems in the original library.

## Initial BREXX Implementation on VM/370

- A memory resident BREXX compiler was integrated into CMS.
- BREXX was selected after initial attempts at porting Regina to CMS proved unsuccessful.
- BREXX was able to reduce the resident memory required by Regina by almost 80%
- The team built a CMS native runtime library. This overcame memory management problems in the original library.
- Still (according to Bob) "The BREXX Interpreter is somewhat flakey"
- Resident BREXX and GCC library routines consumed about 256K

## Why modify the VM/370 Sixpack?

- BREXX  had problems. I found it so restricted it was unusable.
- Adrian started the effort to improve things and discussions ramped up on the h390-vm.groups.io list
- I became involved because I wanted to add compatibility so we can run more old programs
  - Programs from personal libraries or community archives were built on VM/SP and used its capabilities
- Original REXX was released along with new VM capabilities
  - Early REXX programs were built by programmers who expected a certain set of VM system interfaces and utilities.
- I also wanted to fix things that just did not work.
- It is a fun project!

## New (but Old) Facilities for REXX on VM

- Trace control interface
- Stack control interface
- BREXX program call interface
- BREXX function call interface
- Access to REXX variables from programs
- REXX function package support

## CMS Immediate commands for Halt Execution and Trace control

- CMS immediate commands provide a mechanism to enter a command request while another program is running. Older immediate commands stopped program execution or runaway output.
  - HX – Halt Execution
  - HT – Halt Typing
- CMS Immediate commands for REXX
  - HI – Halt Interpretation
  - TS – Trace Start
  - TE – Trace End

## CMS Immediate commands for Halt Execution and Trace control

- CMS immediate commands basically set flags. It is up to REXX to check the flags and take appropriate action.
- The CMS implementation copied the data structures defined in later VM releases. (It's basically trivial.)
- Adrian added BREXX changes to test for a request to end interpretation, or to turn statement tracing on or off.
- The CMS command operands for SET EXECTRAC ON/OFF and QUERY EXECTRAC were added to setup tracing before a REXX EXEC starts, and to manage the settings. This is a cleaner mechanism than having to add trace statements to the EXEC file.

## CMS Stack

- <-ATTN LIFO (push)
  - Line 1 (oldest line) → RDTERM (pull)
  - Line 2
  - Line 3
  - Line 4
  - Line 5

  - Line n (newest line)
- <- ATTN FIFO (queue)

## CMS Stack
### Multiple Stacks

- Stack 1
  - Line 1 (oldest line)
  - Line 2
  - Line 3
  - Line 4 (newest line)
- Stack 2
  - Line 5
  - Line 6
- Stack 3
      - ← ATTN LIFO
  - Line 7
  - Line 8
  - Line 9 → RDTERM
      - <- ATTN FIFO

## CMS Stack
API

- MAKEBUF
  - Create a new stack level
- DROPBUF
  - Remove one or more stack levels
- SENTRIES
  - Return the number of lines in the stack

## CMS Stack
### Example 1

```
/* Copy columns one through 10 from one file to a new file */
address COMMAND
Parse upper arg fname .

'MAKEBUF'
bno = rc
Push 1-10   1

'COPYFILE' fname 'OLD A = NEW A ( SPECS NOPROMPT'
errorcode = rc
If errorcode <> 0 Then
  Say 'ERROR copying file'

/* Remove anything left in the stack */
'DROPBUF' bno

Exit errorcode
```

## CMS Stack
Example 2

```
/* Process an entire file using the stack */
address COMMAND
Parse upper arg fname .

'MAKEBUF'
bno = rc
'SENTRIES'  /* or queued() which calls it */
startCount = rc

'EXECIO * DISKR'  fname 'INPUT A' /* Read file onto the stack */
errorcode = rc
If errorcode = 0 Then
   Do
      'SENTRIES'
      endCount = rc
      lines  = endCount – startCount
      Do lines
         Parse pull nextline
         ret = process(nextline)
         If  ret <>0 Then Leave
       End
   End
/* Remove anything left in the stack */
'DROPBUF' bno
Exit errorcode
```

## Calling REXX Procedures in CMS

- Loading BREXX in memory
  - BREXX originally built as a CMS MODULE file. This makes REXX load in a fixed storage location. This would interfere with BREXX calling other CMS programs in MODULE form.
  - The Sixpack developers brought in a program called RESLIB,  from the Waterloo tapes of the era. It allowed objects to be loaded high in memory and retained between invocations. That design idea was implemented in VM/SP as Nucleus Extensions.
  - RESLIB allows BREXX and the GCCLIB routines it used to be loaded in high memory. This removed the restriction on calling MODULE files from BREXX.

## Calling BREXX Procedures in CMS

- Calling BREXX
  - The Sixpack developers copied the implementation from VM/SP into VM/370 by modifying the EXEC command.
  - That modification changed the EXEC command so that after it located the target EXEC file, it read the first line of the file.
    - If a REXX comment starting sequence of '/*' was found, then BREXX is called.
    - Otherwise the original EXEC processor is called.
  - IBM code does exactly that in VM today.

## Calling REXX Procedures in CMS

- Calling BREXX - problems
  - The Sixpack method required creating a new parameter list and calling DMSREX (the pseudo module loaded in RESLIB storage). This resulted in incorrect processing for certain parameter list types.
  - CMS limits the call depth to 20 levels. BREXX consumed 2 levels on each call.
  - BREXX assumed that calling an EXEC meant calling BREXX, so REXX calling and original CMS EXEC program failed.
  - BREXX functions could not be called by other programs.

## Calling REXX Procedures in CMS

- Calling BREXX - changes
  - Changes in CMS and in BREXX mimicked later IBM code by directly branching into BREXX from the EXEC processor and eliminating the parameter list copy.
  - BREXX now isolates each REXX program into its own environment to eliminate crosstalk errors. Calling EXECs from REXX now always uses the CMS calling method to force a new runtime environment to be built.

## Calling REXX Functions

- REXX Functions are different than REXX procedures in 2 ways
  - They accept multiple argument character strings
  - They may return a character string result
- BREXX did not support the calling functions in a general way
  - Non-REXX programs could not call REXX functions
  - REXX could not call functions written in another language

## Calling REXX Functions

- CMS defined a new calling sequence when REXX was introduced
  - The caller could provide multiple argument character strings
  - The caller could provide a buffer to accept a character string result
  - The Sixpack developers did not support this calling parameter list in their implementation
- Significant changes were required to BREXX to use the CMS calling interface
  - The changes separated each REXX procedure and external function into separate invocations of the interpreter. Crosstalk problems were eliminated.
  - REXX no longer depended on an external function being written in REXX. One impediment to the implementation of REXX function packages in assembler was removed.
  - The CMS EXEC processor no longer had to read the EXEC file. Function calls to an EXEC were always processed in REXX.

## Calling REXX Functions

## Parameter list

SIX WORD PLIST

→Command name (EXEC)

→EXEC filename

→Past last byte of EXEC filename

→Argument list

→Pointer to result

→Result pointer

EVALBLOCK

Reserved Word

Size in doublewords

Exact length of result

Result character string

-----------------------------------------

ARGUMENT LIST

→First argument , Length of first argument

→Second argument, length of second argument

FFFFFFFF FFFFFFFF

## Infrastructure Requirements
## -
## Most REXX programs in the early 1980s depended on 3 CMS facilities

- The ability to package external function, written in assembler, and load them in high memory. These functions required a global memory capability to share information. For example, a file OPEN, READ, WRITE, CLOSE function set had to communicate with each other.
- The ability to issue subcommands to be processed by an active program. For example, the XEDIT editor could support macro programs, written in REXX, that could directly issue multiple XEDIT commands selected under program control.
- The ability for external programs to access the values of REXX variables and assign new values to those variables.

## Nucleus Extension Infrastructure

- CMS storage management for loading programs was simple and restrictive.
  - User programs generally ran one at a time and loaded in a fixed storage location called the "user area".
  - Small programs (usually assembler) smaller than 8K could be generated to load in the "transient area". Transient area programs could be called by user area programs.

## Nucleus Extension Infrastructure

- ------------------ End of virtual machine
- Free
-   Storage
- ------------------ Variable (lowered when high memory allocated)
- User
-   Area
- ------------------ 20000
- CMS
-   System code
- ------------------ 10000
- Transient area
- ------------------ E000
- CMS Nucleus
- ------------------ Location

## Nucleus Extension Infrastructure

- Loading CMS programs into high memory presented some problems. VM/370 and early releases of VM/SP did not have a relocating loader. Programs loaded as Nucleus extensions had to be:
  - 1. Completely relocatable – no program address constants. Unfortunately, CMS Application Programming Interface (API) macros had many address constants.
  - 2. Processed by the OS Linkage Editor into a member of a CMS Load Module Library (LOADLIB).
- In spite of these restrictions, a Nucleus Extension capability was available in CMS before REXX was released.

## Subcommand Interface Infrastructure

- The ability for programs like the original EXEC processor to function as a macro processor for the EDIT program was not available in VM/370.
  - During an EDIT session, the user could run an EXEC by entering the name on the command line.
  - The EXEC could place a series of EDIT commands onto the program stack.
  - When the EXEC returned control to the EDIT command, the stacked lines were executed as EDIT command.
  - Obviously, functionality was limited, because there was no ability for an external program to issue an EDIT command directly and check to return code for success.

## Subcommand Interface Infrastructure

- The EDGAR editor was an IBM editor available for an extra charge in VM/370. It used a full screen architecture and had a product specific calling sequence to allow other programs to submit EDGAR subcommands.
- VM/SP added a subcommand interface for use primarily by the XEDIT editor. The EXEC2 processor added an &SUBCOMMAND directive so that EXEC2 could become a macro processor for XEDIT. A MACRO, implemented in EXEC2, could be called by XEDIT, then use program logic to issue the appropriate XEDIT commands.

## Access to EXEC Processor Variables

- VM/SP added the EXECCOMM capability for the EXEC2 processor.
- A program called by EXEC2 could retrieve or assign the values of EXEC2 variables.
  - The program created an SHVBLOCK control block which specified the name of one EXEC2 variable. Multiple SHVBLOCKS could be chained in one request to process multiple variables.
  - For variable retrieval, the program supplied a buffer address and length. If the retrieval buffer was too small, the required length was passed back to the program so that the request could be re-tried.
  - For variable assignment, the value buffer and length are passed to EXEC2.
  - EXEC2 variable values are limited to 255 characters.

## Access to EXEC Processor Variables

- The EXEC2 variable interface was first used by the new EXECIO program. It provided:
  - I/O to CMS files and Unit Record Devices (Virtua Card Reader, Printer, and Punch)
  - The ability to issue CP commands and retrieve the response
  - Data transfer using the stack or direct variable access
  - Filtering and selection of input data

## Infrastructure Requirements - CMS was ready

- The ability to package external function, written in assembler, and load them in high memory. These functions required a global memory capability to share information. For example, a file OPEN, READ, WRITE, CLOSE function set had to communicate.
- The ability to issue subcommands to be processed by an active program. For example, the XEDIT editor could support macro programs, written in REXX, that could directly issue multiple XEDIT commands selected under program control.
- The ability for external programs to access the values of REXX variables and assign new values to those variables.
- VMSHARE posts from that era mention that Mike Cowlishah worked closely with the EXEC2 team.

## REXX Function package support

- REXX used the Nucleus Extension facility to allow an EXEC to call conforming assembler programs using the External Function interface. Either a CALL statement or a function call sequence was used.
- At the time, CMS provided services were generally available only to assembler programs, so this gave REXX the potential for complete access to VM services.
- Many VM installations created function packages and distributed them.

## REXX Function package support

- REXX dynamically loaded these assembler routines on first use.
  - Functions were grouped into one of three packages:
    - RXSYSFN – IBM provided
    - RXLOCFM – Installation provided
    - RXUSERFN – End user provided
  - REXX first called an external function, and if it was not found, it tried to load the function with a call to each package (in the order of User, Installation, and IBM package) and then re-tried the call to the function.
- Many REXX packages are available on VM collections of the time from Waterloo and VMSHARE

## Deliverables for VM/370

- H390-vm@groups.io is the discussion group for Hercules-390 and VM/370. VM modification files for the CMS functions to support REXX are in the 'Files' area in the directory 'CMS Extensions for BREXX'
- These modifications are written to be applied to the VM/370 Sixpack Version 1.3
  - Support use of REXX as filetype EXEC
  - Expand NUCON to 4K
  - NUCEXT and SUBCOM support
  - MAKEBUF,DROPBUF, and SENTRIES
  - Query and SET EXECTRAC
  - CMS Support for REXX External function calls

## Deliverables for VM/370

- We hope to establish a new VM/370 deliverable in the near future which will have these VM modifications already applied. The latest GCCLIB and BREXX updates will be included in that.
- Adrian has a BREXX deliverable on his github development environment, which I don't completely understand ☺
- An implementation of EXECIO is in final testing.
- I plan on making available a BREXX and GCCLIB deliverable available within a week or two,  for direct installation on a Hercules hosted VM/370 system.

## Demo

- This demonstration exploits all the VM changes and BREXX/GCCLIB updates we have done.
- It is unchanged from the source file on my VM history disk.
  - Except for one thing☺ - I remember writing functions to communicate with the Message System Service with IUCV. That is not available in VM/370. I did have to apply an update to use VMCF instead of IUCV to respond to an SMSG command to my server.

## Demo

- The demo is for a “WAKEUP” server. Many people wrote code for this kind of server virtual machine. A VM/370 system, and the users on that system, produce items like spool files, monitor data, accounting data, etc. that gets left on the spool system or on disk that needs to be selectively cleaned up.
- A WAKEUP server has the job of responding to a schedule or other external events to perform cleanup activities.

## Demo

```
/* Entries in the WAKEUP TIMES file are as follows:                *
/*                                                                 *
/* 1         10        19        28       <-- COLUMNS              *
/*                                                                 *
/* MM/DD/YY  HH:MM:SS  DATESTAMP USER-TEXT  (Done once)            *
/* ==/DD/YY  HH:MM:SS  DATESTAMP USER-TEXT  (Once a month)         *
/* ==/==/==  HH:MM:SS  DATESTAMP USER-TEXT  (Once a day)           *
/* ==/01/==  HH:MM:SS  DATESTAMP USER-TEXT  (On the 1st)           *
/*                                                                 *
/* ALL       HH:MM:SS  DATESTAMP USER-TEXT  (Once a day)           *
/* DAYOFWEEK HH:MM:SS  DATESTAMP USER-TEXT  (Once a week)          *
/* WEEKEND   HH:MM:SS  DATESTAMP USER-TEXT  (On weekends)          *
/* S-S       HH:MM:SS  DATESTAMP USER-TEXT  (Same as above)        *
/* WEEKDAY   HH:MM:SS  DATESTAMP USER-TEXT  (On weekdays)          *
/* M-F       HH:MM:SS  DATESTAMP USER-TEXT  (Same as above)        *
```

## Demo

```
/* ==/==/==  +05       TIMESTAMP USER-TEXT  (Every 5 minutes)       */
/* WEEKEND   +10:30    TIMESTAMP USER-TEXT  (Every 10:30 Weekends) */
/* WEEKDAY   +20       TIMESTAMP USER-TEXT  (Every 20:00 Weekdays) */
/* DAYOFWEEK +05       TIMESTAMP USER-TEXT  (Every 5 MIN On day    */
/*                                                                  */
/* Column  1: Day/Date to wakeup for this event                     */
/*        10: Time of day interval to wakeup (FILETIME)             */
/*        19: Date/Time stamp when completed (FILELAST)             */
/*        28: User test for this event       (FILEDATA)             */
```