SLAC–401
CONF–9205149
UC–405
(M)

PROCEEDINGS OF THE REXX SYMPOSIUM FOR DEVELOPERS AND USERS

May 3–5, 1992
Annapolis, Maryland

Sponsored 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.

PROCEEDINGS OF THE REXX SYMPOSIUM
FOR DEVELOPERS AND USERS

TABLE OF CONTENTS

SUMMARY

The third annual REXX Symposium for Developers and Users was held on May 3–5, 1992 in Annapolis, Maryland. Ninety-one people attended, representing eight countries and nineteen American States.

There was a great deal of interest regarding REXX in the UNIX world. Alberto Villarica and Anders Christensen announced two free versions of REXX for UNIX. Also, the Workstation Group announced a free run-time version of their Uni-REXX available to any educational institution.

Two flavors of REXX under MS/Windows were presented by Eric Giguere and Charles Daney. Both implementations demonstrated the ease in which REXX was able to create GUI applications. This prompted some discussion of REXX under the Macintosh, probably the last frontier for REXX.

Along with his usual informed opinions, Mike Cowlishaw described some of his current research efforts. He also wowed us with some statistics demonstrating the incredible penetration REXX has made internationally as measured by the number of books published about it and the millions of users with access to it.

Prompted, in part, by Lotus' announcement of a REXX interface to 1-2-3, there was much discussion of ways that REXX could be promoted as a universal scripting and macro language. In this context, REXX was compared both to PERL and to Visual BASIC, which appears to be Microsoft's attempt to cover some of the deficiencies of BASIC.

Next year's symposium will be held in San Diego, California on May 18–20. Because of the great success of this year's symposium, we are expanding to three days next year and look to you, the REXX community, to help us fill these days with interesting and useful presentations.

Signed,

1992 Program Committee:

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

Announcement of the Regina REXX Interpreter

Anders Christensen, University of Trondheim

Announcement of the Regina REXX Interpreter

Anders Christensen <anders@solan.unit.no>

Annapolis, May 5, 1992

Summary

Regina is a REXX interpreter for Unix systems, written in ANSI C, lex and yacc. The source code for Regina is available by anonymous ftp on Internet. Regina is "free" software, meaning that you don't have to pay for it.

Platforms

Regina has been built on several systems, under the following environments:

- GCC v2.1, flex and bison, on several OS architectures.
- c89 (unbundled) on Decstation 5000, under Ultrix 4.2
- cc on Irix Indigo, under IRIX 4.0
- acc 1.1 (unbundled) on Sun Sparc, under Sunos 4.2
- cc on Decstation 3100, under OSF/1
- c89 on HP 9000, under HP/UX 8.05

The lex and yacc code included in Regina is fairly standard, and can easily be processed by the standard yacc and lex utilities under all the systems named above.

The C code is ANSI C and uses POSIX, when interfacing to the operating system. On several machines where the standard setup of the C compiler is not completely ANSI C and POSIX, you might have to set compiler options to force the compiler to use these standards.

At a few places in the source, where POSIX is not powerful enough, non-POSIX code has been included. Alternative POSIX-compliant source is also present, and may be chosen instead through the setting of C preprocessor flags.

What is Included in Regina

Regina follows the 3.50 version of REXX, as described by the first edition of "The REXX Language" by Mike Cowlishaw. The areas where it is not according to that description are:

- The `SIGNAL ON` command is missing
- Arithmetics are done using C-functions, so anything related to the `NUMERIC` command will not work. In fact, conversion of numbers might even be dependent on the C-compiler you are using. This also effects the results of the `FORMAT()` builtin function.
- There are some problems connected with tracing.
- For (external) commands, there is not a persistent shell in the background to which commands are sent. Instead, a shell is started up each time (`ADDRESS SYSTEM`), or the command is run directly (`ADDRESS PATH` and `ADDRESS COMMAND`).

Other Parts of the Regina Package

Regina comes with more than just the source code for the interpreter itself. A set of documents that describe the functionality of the interpreter, both the standard REXX functionality, and the extra functionality of Regina, in particular the parts interfacing to the Unix system. The documentation is located in the "`doc`" subdirectory of the Regina distribution.

Also included is a test consisting of a set of REXX programs that check various parts of the REXX language, in particular the more obscure features, border-conditions and limits. Both this "trip-test" and the documentation are under construction, and are far from complete in the current version.

In the "`code`" subdirectory are various small REXX programs included, that demonstrate features and programming techniques in REXX. I have no intention of writing all these myself. I hope to be able to include small REXX-programs written by other people, in this demo-directory, in order to gather a nice collection of instructive programming pearls.

In the Future ...

I intend to continue developing Regina, improvements and new features include:

- The remaining parts of the 3.50 REXX standard will be implemented. In particular, true string arithmetics will be added.
- The interpreter will be made compatible to the 4.00 version of REXX, as defined in the second edition of "The REXX Language".
- A mechanism for dynamically adding external function packages, during execution time. These packages may be written in compiled languages (e.g. C). This will allow Regina to use numerous functions as if they were builtin, without having to link in the code for these functions into the executable of the interpreter at compile time.
- Using this library mechanism, some libraries will be added to Regina, including a wrapper library to curses (for fullscreen manipulation of ASCII graphics), a math library, an interface to Unix system services and an interface to TCP/IP.
- I will port Regina other Unix systems, and to some non-Unix systems, in particular MS-DOS and VAX/VMS. Support for other systems will depend on what access I have to those systems.
- Tools for program development will be added, such as syntax-checking, pretty-printing, crossreferences etc.

Still in Beta Version

Please note that Regina is still in beta-version. The code will be released as version 1.00 when full REXX 4.00 functionality has been implemented, and most of the known bugs have been removed.

Where to Get Regina

If you have access to anonymous ftp on Internet, you can get it from the server:

flipper.pvv.unit.no (129.241.36.200)

Use ftp to log in to the account "`ftp`", and use you electronic mail address as password (that is the normal etiquette of the Net.) If you do not have access to the anonymous ftp service, you will have to get Regina from another source. Hopefully other people will redistribute the code to places to which you have access.

Note that flipper is located in Norway, so if you are located outside Europe and can get the Regina from a site closer to you, please try to do so.

Copyright and Distribution

As long as the code is released as beta-version, the copyright for the interpreter generally says: "You can use it for whatever you want, as long as you don't use it for commercial purposes.". More details on these is included in the "README" file that accompanies the source distribution.

When Regina is released as version 1.00, it will most likely have the copyright notice generally known as the "Gnu General Public License" (GPL). It (generally) says: "You can (re)distribute the program as you wish, including selling it, but you have to provide the full source for it when you distribute it. Including the source for any modifications you might have done to the program." More information about this is available in files contained in the distribution.

Bugreports

If you use Regina, and you find a bug in it, I would be very glad to hear about it. Although I do not guarantee that I will fix anything that is broken (after all it is free software), I generally fix anything that I too consider broken, and which is within my capasity to fix.

The interpreter in its current version has bugs and missing features, some of them are listed in the accompanying documentation. If you report a bug, the easiest method (at least for me) is to use electronic mail with a description of the bug. Preferably, such a bugreport should contain:

- Description of what equipment you used, i.e. hardware-platform, operating system, compiler version, compiler options used, version of Regina etc.
- A description of the buggy behavior that you saw (and the behavior that you expected to see.)
- Preferably a piece of REXX code that demonstrates the behavior, it should be as small as possible, preferably not more than about 10 lines.
- If you have already fixed the bug, please append a context diff of the changes you made to the source, then I won't need to redo the same work to track down the bug.

Please make sure that the bug is really a bug in the interpreter, not a bug in your program or a peculiar behavior of your machine. If possible, run your program on other REXX interpreters to see how they behave, and check with a REXX manual if you have access to one.

Where to Send Electronic Mail

If you have questions concerning Regina, feel free to contact me at my electronic mail address listed below. I will gladly accept comments, bugreports, wishes or cries for help. But since I do this on my spare time, and since I don't charge any money for it, I can't guarantee bugfixes and help in advance.

Anders Christensen
Norwegian Institute of Technology
University of Trondheim

email: anders@solan.unit.no
or: anders@pvv.unit.no

snail: Stud.post. 31
N-7034 Trondheim-NTH
Norway

REXX—THE FUTURE

MIKE COWLISHAW
IBM

REXX—The Future

Mike Cowlishaw

IBM UK Laboratories
Hursley

29 Apr 1992

The Future of REXX

- Where are we now?
- REXX assets
- Trends and directions
- Discussion

29 Apr 1992

Mike Cowlishaw

Where are we now?

- 17 implementations, on most significant platforms
- 35 published books and manuals

  (Over 50, if service guides, second editions, and translations are included.)
- Accessible to over ten million users
- Widely used, with an international following
- ANSI standard work well under way.

Which assets are most important for the future?

- Simplicity:
  - A small, readable, language
  - Just one data type—the string
  - Decimal arithmetic
  - Few limits
- Flexible and extendible
  - Existing and future system interfaces

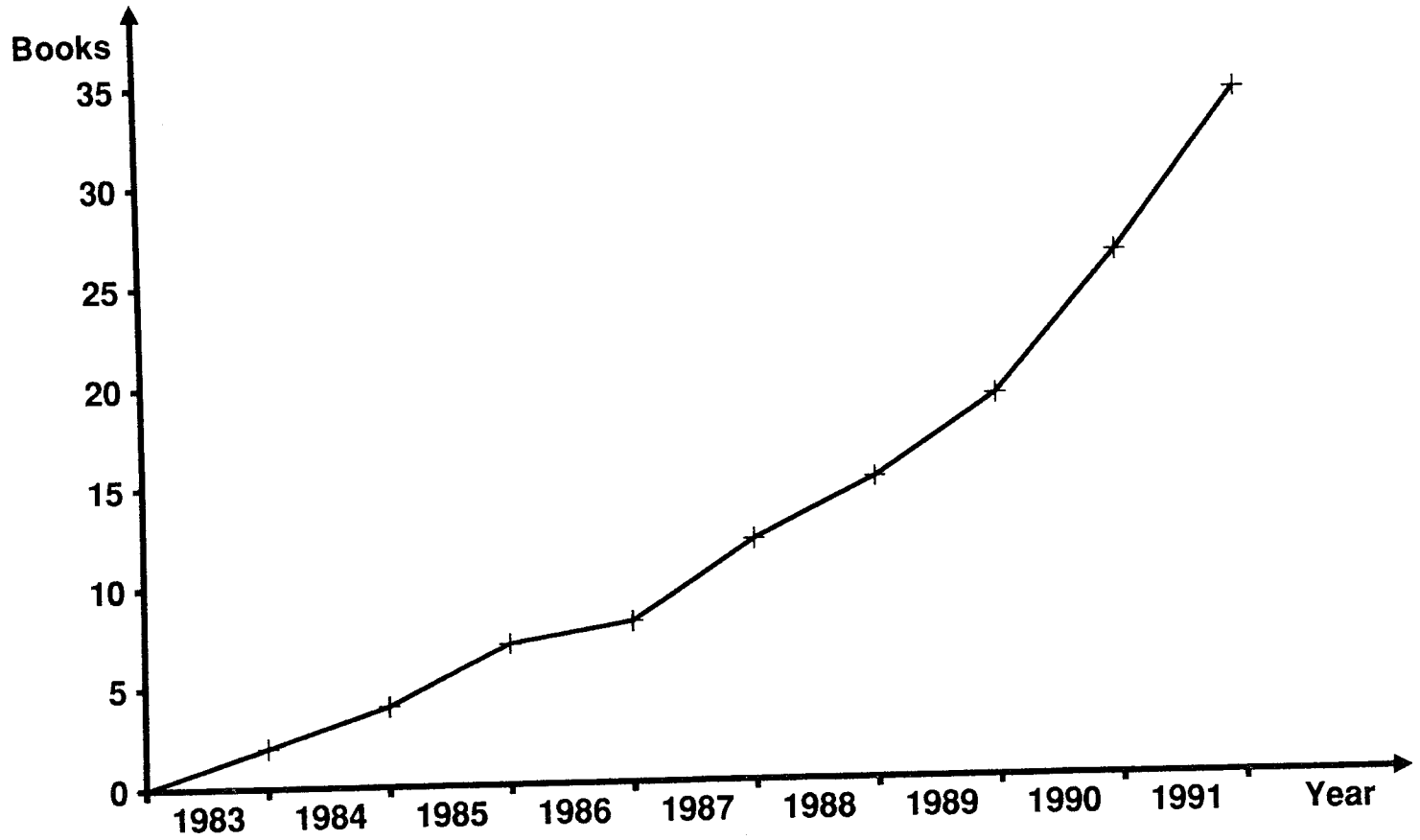
More assets...

- *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.

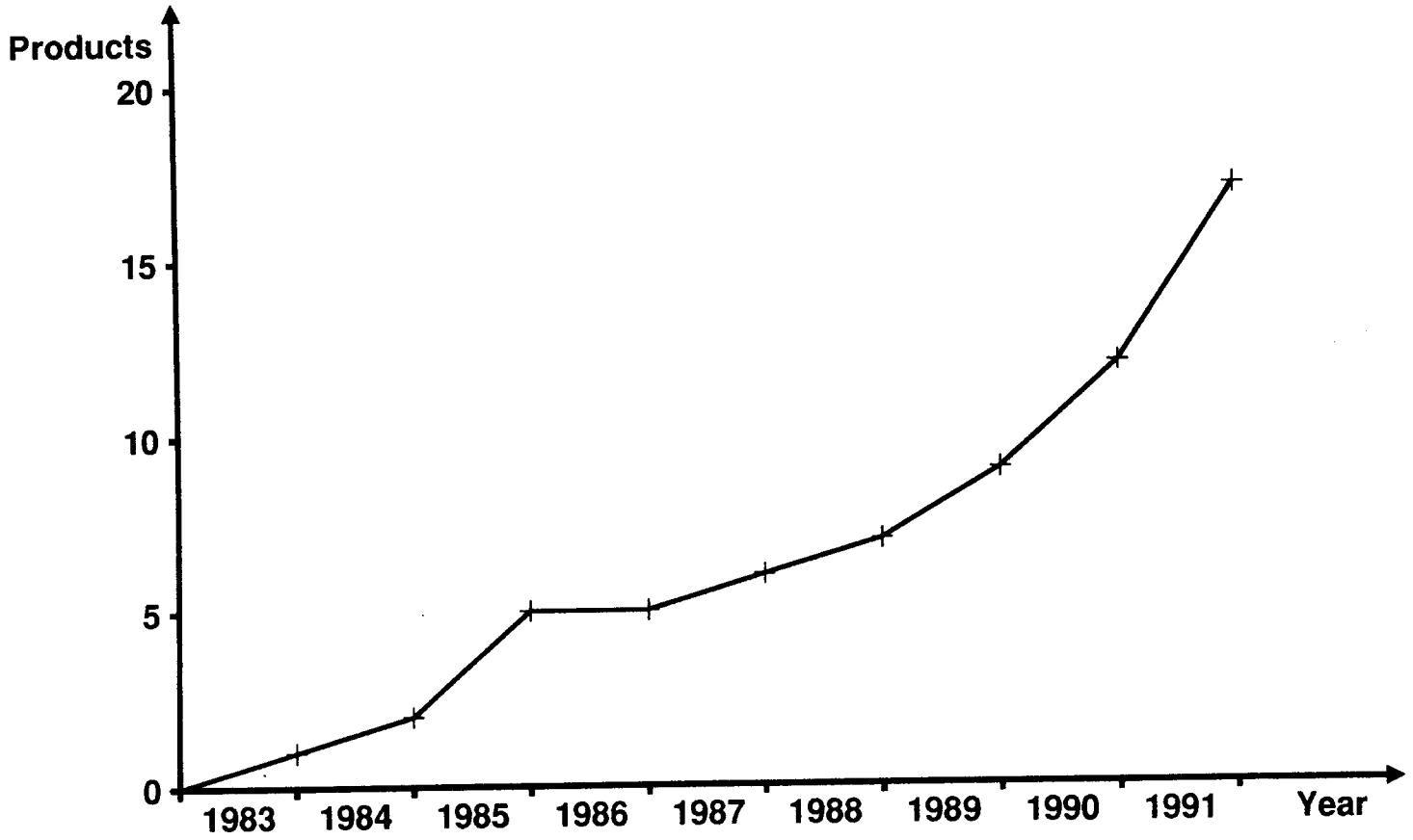
Trends and directions

- Mainframe interactive applications continue to move to workstations and PCs
- Networking of workstations and PCs encourages standardization of applications and languages
- Increasing complexity of applications, and sophistication of users, demands extensive subsetting and customization.

REXX Books and Manuals



REXX Language Products Available



29 Apr 1992

Mike Cowlishaw

WinREXX: Personal REXX for Windows

Charles Daney and Stan Murawski
Quercus Systems

WinREXX
Personal REXX for Windows

- Background
- Current Product and Capabilities
- Market Needs and Opportunities

By Stan Murawski
(408) 288-6759, or CIS Mail 70444,55

The REXX Language

Created by Michael Cowlishaw.

Originally for IBM mainframe timesharing users (VM/CMS replacement for EXEC II).

Conceived as a "Command Language" running in a hosted environment (embedded).

16

***Not* an IBM product development - grass roots.**

Easy to learn and use, designed to be *extensible*.

Keep it Small and Simple philosophy.

Becoming a Standard Language

IBM Systems *had* disparate Command Languages:
- **Mainframe - MVS/TSO and VM/CMS.**
- **Mini - OS/400.**
- **Desktop - OS/2 (CMD/BAT).**

Quercus (Mansfield) delivers Personal REXX.

IBM Chose REXX for *all* their platforms.

ANSI chose REXX as its *newest* standard language.

Commodore embedded AREXX into AMIGA DOS.

The Workstation Group publishes UNIREXX.

Quercus publishes WinREXX.

Personal REXX (DOS and OS/2)

Originally developed as a macro language for KEDIT, an IBM VM XEDIT compatible editor.

A REXX 4.0 compliant language for DOS & OS/2 - *the* REXX for PC's.

A powerful alternative to DOS BAT files and OS/2 CMD files.

A good language for developing simple utility applications - some useful extensions.

Comprehensive extensions for DOS disk and file manipulations, e.g. file attributes, free space.

WinREXX 1.x

A REXX 4.0 compliant language processor in a Windows DLL.

A comprehensive API - (compatible with the IBM OS/2 REXX API).

A window for editing and running REXX programs.

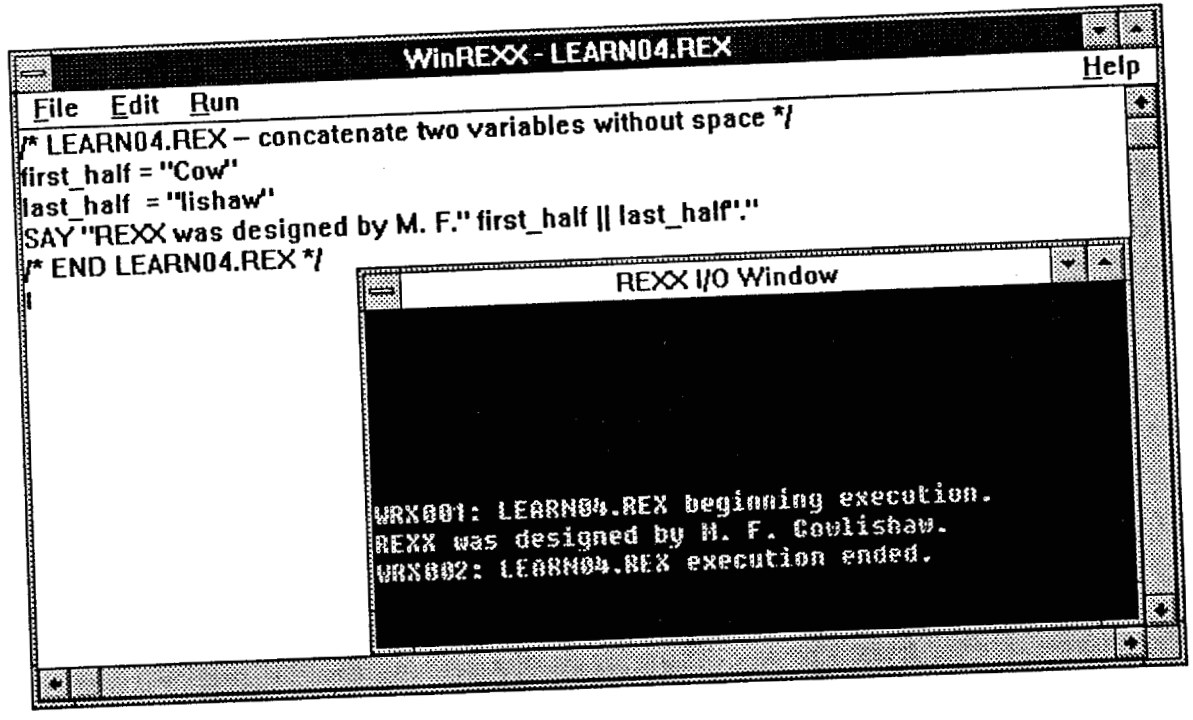
A window for REXX program Input and Output.

The DOS REXX extensions, e.g. file management.

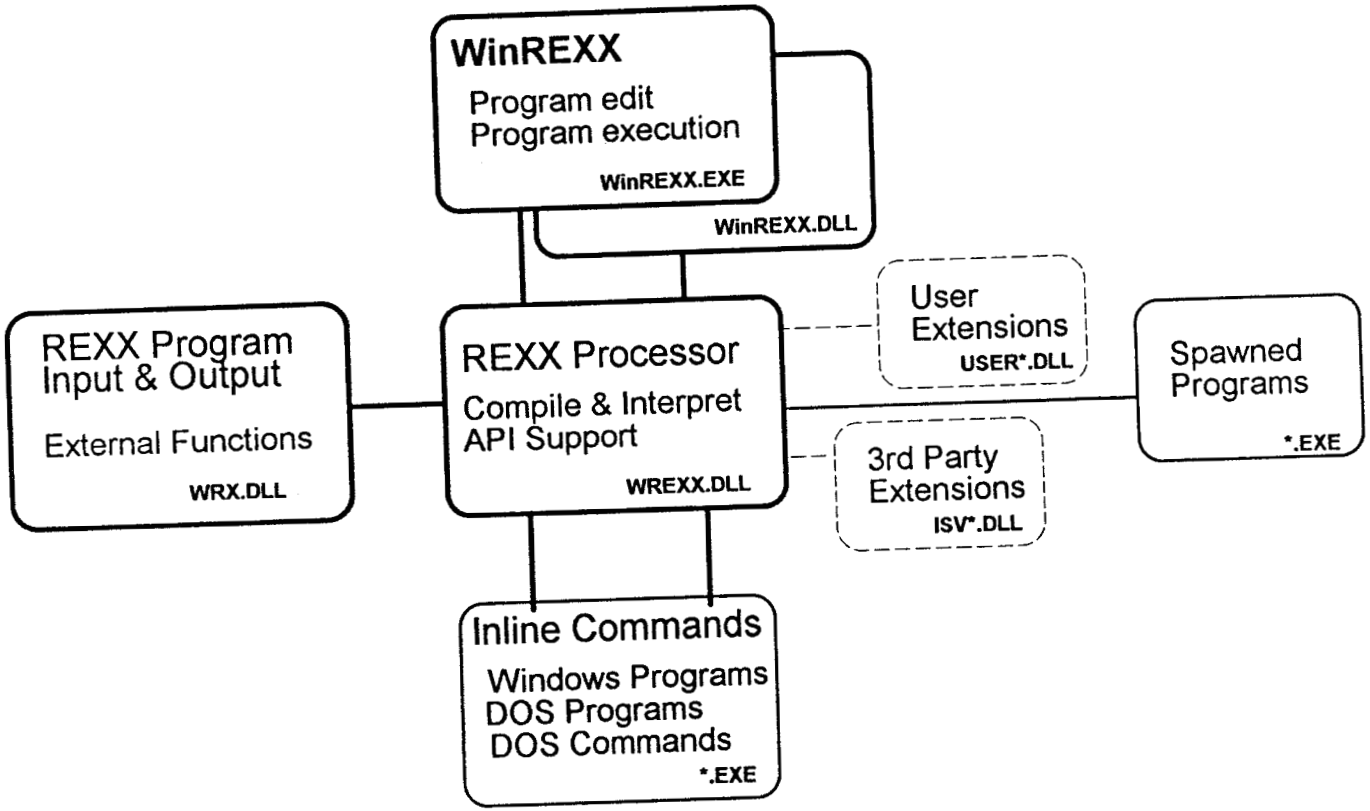
Windows REXX extensions, e.g. MessageBox.

The heart of a command language for Windows.

The WinREXX Windows



WinREXX's Executable Structure



Only WREXX.DLL is *needed* for embedded REXX.

Some WinREXX Features

- Several sets of Windows HELP
  - Using WinREXX
  - Learning REXX
  - Messages Explanations
  - The WinREXX API
- User Input Alternatives to SAY/PULL
  - Message/Question/CancelBox
  - PromptBox
  - ChoiceBox
- Automatic PARSE ARG prompting
- Get/Set "INI" file strings

WinREXX API Capabilities

Run REXX programs.

Create Subcommand environments.

Add External Functions.

Interrogate & Change REXX variables.

REXX Processing Exits, e.g. I/O or Trace.

Support *multiple concurrent threads* of execution
(from multiple windows tasks)!

A Command Language for Windows Applications

Windows does not have *any* command language.

Each vendor must write its *own* command language processor.

Each Windows product has its own *proprietary* command language.

The user must learn *many* languages, each different in structure, syntax and commands.

User still can't do what can easily be done in DOS, e.g. directory traversal and file manipulation.

The User Market Need

Users need a single consistent command language:

- **Macros.**
- **Scripting.**
- **DDE control.**
- **OLE control.**

Vendors need a standard embedded language - without writing their own language processor.

The market needs a central point of control - not a macro in *every* applications.

There needs to be one powerful "glue" to bind applications together!

REXX Based Possibilities

REXX extensions for generic DDE and OLE.

A window that "knows" DDE and OLE to popular Windows applications, so users can "link" applications from one "REXX" place.

REXX based control of multi-application OLE - a "cross-application" command language.

WinREXX as a generic OLE Server, with REXX program created Objects (embedded output).

Cross-network support for applications on other platforms, e.g. UNIX, MAC?

Command Language Competition

Microsoft is variously promoting "System Basic"
- **Will this be part of NT?**
- **Is Word Basic a prototype for all MS apps.?**
- **(BASIC has a special place in the heart of Bill Gates).**

REXX vs. BASIC was part of the IBM - Microsoft "divorce".

IBM could be an ally, and may be important depending on the success of OS/2 version 2.

Bridge Batch is a probable competitor
- **Softbridge promoting as a "common language".**

SoftSCRIPT?, WinBatch?

WinREXX

Positioned to become *the* industry standard embedded command language for Windows!

RELATIONAL ARCHITECTS PRODUCT FAMILY

CARL FEINBERG
RELATIONAL ARCHITECTS

Relational Architects Product Family

Presented at

REXX Symposium

Annapolis, MD -- May 1992

by

Carl Feinberg

Director of Development
Relational Architects Intl

30

Some of our clients

American Hospital Association

American President Lines

Australian Telecom

Bank of Liechtenstein

Blue Cross Blue Shield

British Columbia Telephone

British Telecom

CalFarm Insurance

Ciba - Geigy

Credit Suisse (Switzerland)

Daimler Benz (Germany)

Depository Trust Company (NY)

Dow Corning

Dresdner Bank (Germany)

Ericsson (Sweden)

The Equitable Life Assurance Soc.

Federal Government Agencies

Fireman's Fund Insurance Co.

The Franklin Mint

G E Information Services

Glaxo, Inc.

The Home Depot

I B M Corporation

Iowa Public Service

Los Alamos National Labs

Los Angeles Water and Power

MCI / Telecom*USA

Mead Corporation

National Westminster Bancorp

Norwegian Telecom

NYNEX

PARS

The Pillsbury Company

State Bank of Sweden

Tandy Corporation

United States Fidelity & Guaranty

University of California

U S A A

VISA International

Product groups

DB2 Productivity Series

- RLX/REXX
- RLX/ISPF
- RLX/CLIST
- RLX/Compile
- RLX/Net
- AcceleREXX

Smart Jobstream Series

- Smart/CAF
- Smart/Restart
- Smart/QBF
- Multi/CAF

RLX Product Family

- ☐ *Extends embedded SQL support to REXX EXECs and TSO CLISTs*
- ☐ *Exactly the same embedded SQL used with COBOL and PL/1*
- ☐ *Fully supports IBM's RXSQL syntax*
- ☐ *Full host and indicator variable support*
- ☐ *RLX parser validates SQL statements, assigns "best fit" data types*
- ☐ *Full support for SQLCA and SQLDA fields*
- ☐ *Extensive full screen diagnostics*
- ☐ *Multi/CAF supports concurrently active DB2 plans*
- ☐ *Quasi-static SQL*

RLX/ISPF

- ☐ *Extends SQL's set processing facilities*
- ☐ *Powerful composite functions*
- ☐ *Load SQL query results directly into ISPF tables*
- ☐ *Display and process those results on scrollable ISPF panels*
- ☐ *Creates an ISPF table containing columns selected from a DB2 table*
- ☐ *Encapsulated object to manipulate table*

How can we use RLX?

- ☐ *Build DB2/ISPF based tools quickly and easily (DBA utilities and developer workbenches)*
- ☐ *Prototype high volume applications for CICS, IMS and batch*
- ☐ *Develop decision support, individual and departmental applications*
- ☐ *Develop production applications for the DB2/ISPF environment*
- ☐ *Testing tool for performance analysis and problem resolution (one time fixes)*
- ☐ *Teaching tool - Learn SQL with immediate feedback, extensive diagnostics and context sensitive help*
- ☐ *NetView automation procedures*
- ☐ *Automated console operations for system administration*

Why Interpretive?

- ☐ *Quick and easy development*
- ☐ *'glue' to integrate diverse components like DB2, ISPF REXX and Netview into cohesive application solutions. Combines SQL, ISPF dialog services and procedural logic into a functional unit*
- ☐ *Edit and test RLX dialogs directly within PDF/Edit*
- ☐ *No preprocess, compile, link edit or bind steps are required*
- ☐ *Reduce application size and complexity by at least 50%*
- ☐ *Reduce development, maintenance and enhancement time by at least 50%*
- ☐ *Increase application functionality by 100%*

Developers can

- ☐ *Apply their SQL skills immediately*
- ☐ *Quickly prototype applications*
- ☐ *Copy RLX SQL statements directly into their COBOL and PL/1 programs*
- ☐ *Ignore data declarations and data conversion and concentrate on algorithms*
- ☐ *Skip Preprocess, Compile, Link Edit and Bind steps entirely*
- ☐ *Quick trial and error development approach*

Sample RLX/REXX dialog

RLXS TOWNER SYSIBM

```
--------------------------- Tables Created by SYSIBM -------------  ROW 1 OF 30
Command ===>                                                    Scroll ===> HALF

Table Name          Owner     Type   DB Name   TS Name      DB ID      Colcount

SYSCOPY             SYSIBM      T    DSNDB06   SYSCOPY          6            14
SYSFIELDS           SYSIBM      T    DSNDB06   SYSDBAS          6            13
SYSTABLESPACE       SYSIBM      T    DSNDB06   SYSDBAS          6            23
SYSTABLES           SYSIBM      T    DSNDB06   SYSDBAS          6            31
SYSTABLEPART        SYSIBM      T    DSNDB06   SYSDBAS          6            21
SYSTABAUTH          SYSIBM      T    DSNDB06   SYSDBAS          6            21
SYSSYNONYMS         SYSIBM      T    DSNDB06   SYSDBAS          6             6
SYSRELS             SYSIBM      T    DSNDB06   SYSDBAS          6            11
SYSLINKS            SYSIBM      T    DSNDB06   SYSDBAS          6            12
SYSKEYS             SYSIBM      T    DSNDB06   SYSDBAS          6             7
SYSINDEXPART        SYSIBM      T    DSNDB06   SYSDBAS          6            16
SYSINDEXES          SYSIBM      T    DSNDB06   SYSDBAS          6            26
SYSFOREIGNKEYS      SYSIBM      T    DSNDB06   SYSDBAS          6             7
SYSCOLUMNS          SYSIBM      T    DSNDB06   SYSDBAS          6            19
SYSCOLAUTH          SYSIBM      T    DSNDB06   SYSDBAS          6            10
```

Implementation of TOWNER Using RLX/REXX

```
/*  RLX REXX EXEC TOWNER -- using embedded SQL and ISPF services  */
arg createdby    /* Obtain the creator's name as a parameter       */
address RLX      /* Route host commands to RLX for execution       */

/*  You denote REXX host variables with the standard colon prefix.*/
"rlx declare tblnames cursor for
     select name, creator, type, dbname, tsname, dbid, colcount
         from sysibm.systables
         where createdby = :createdby"

/*  Address RLX recognizes all ISPF dialog service names          */
"TBCREATE TBLNAMES
    NAMES(NAME, CREATOR, TYPE, DBNAME, TSNAME, DBID, COLCOUNT)
    NOWRITE"

"rlx open tblnames"           /*  Produce SQL query result        */

/* RLX FETCHes values directly thru memory into 'host'            */
/* variables which RLX, ISPF and the REXX interpreter share.      */
"rlx fetch tblnames into
name, :creator, :type, :dbname, :tsname, :dbid, :colcount"

/*  RLX updates all the host variables comprising the SQLCA       */
Do while sqlcode = 0      /*  While FETCHes are successful       */
   "TBADD TBLNAMES"

   /*  RLX recognizes statements it's already processed to further*/
   /*  improve performance.  A reexecuted RLX SQL statement runs  */
   /*  at 'static' SQL speed.                                     */

   "rlx fetch tblnames into
        name, :creator, :type, :dbname, :tsname, :dbid, :colcount"
END
"rlx close tblnames"          /*  close the cursor                */

"TBTOP TBLNAMES"
Do while rc = 0               /*  until user signals end or return */
   "TBDISPL TBLNAMES PANEL(TBLNAMES)"
End
"TBEND TBLNAMES"
exit rc
```

Implementation of TOWNER Using RLX/ISPF

```
arg createdby    /* Obtain the creator's name as a parameter        */
address RLX      /* Route host commands to RLX for execution        */

/*  Flow the SQL query result into an ISPF table -- with a single */
/*  statement -- using the DECLARE ISPFTABLE service               */

"rlx declare tblnames ispftable for
select name, creator, type, dbname, tsname, dbid, colcount
   from sysibm.systables
     where createdby = :createdby"

Do while rc = 0
   "rlx tbdispl tblnames panel(rlx)"
end

Exit rc
```

RLX Feature Summary

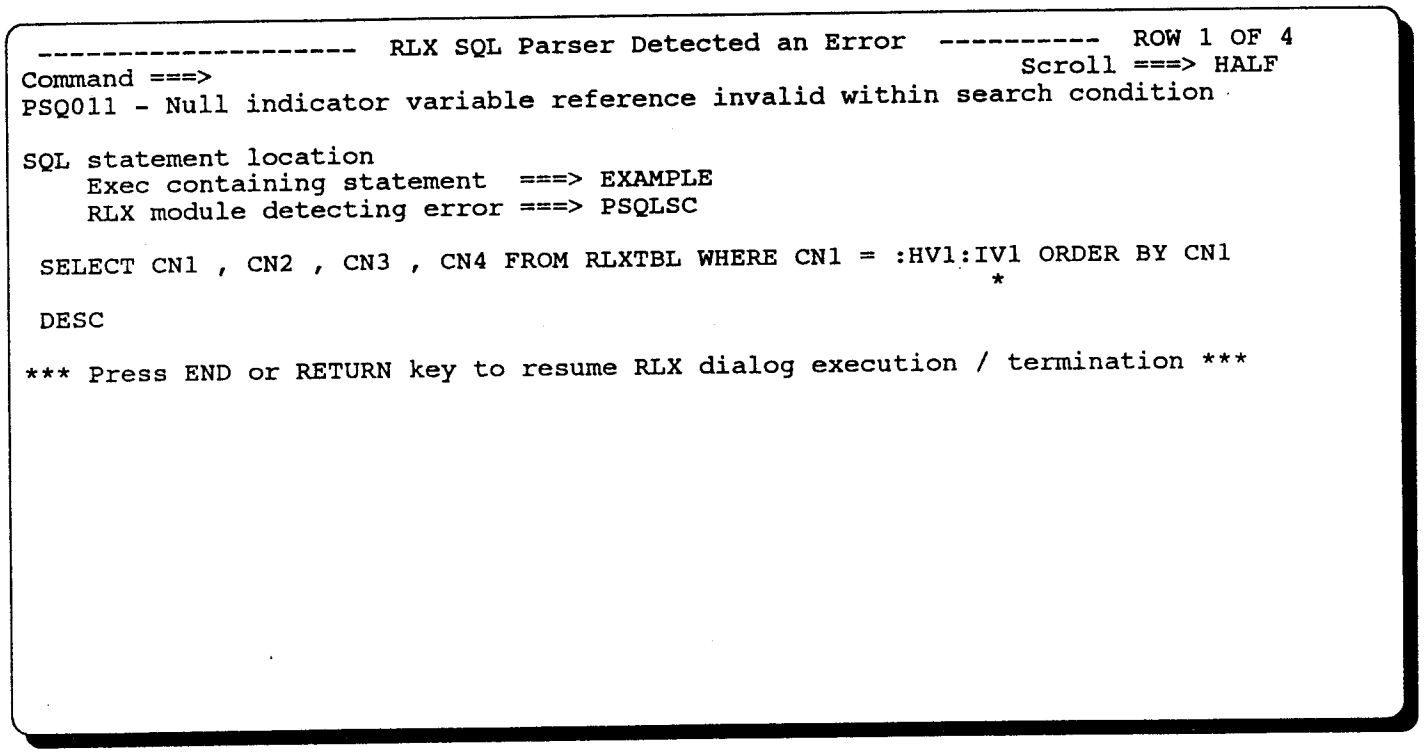
- ☐ NULLs and host variables fully supported
- ☐ Automatic conversion between internal and external data formats
- ☐ SET and ROW oriented processing
- ☐ SQL Communications Area feedback after each RLX SQL statement

41

Human engineered

- ☐ Interactive diagnostic facilities pinpoint errors and speed their correction
- ☐ Profile facilities customize RLX operation
- ☐ ISPF split screen is fully supported
- ☐ RLX can run concurrently in both screens

Syntax errors detected by the RLX semantic parser:

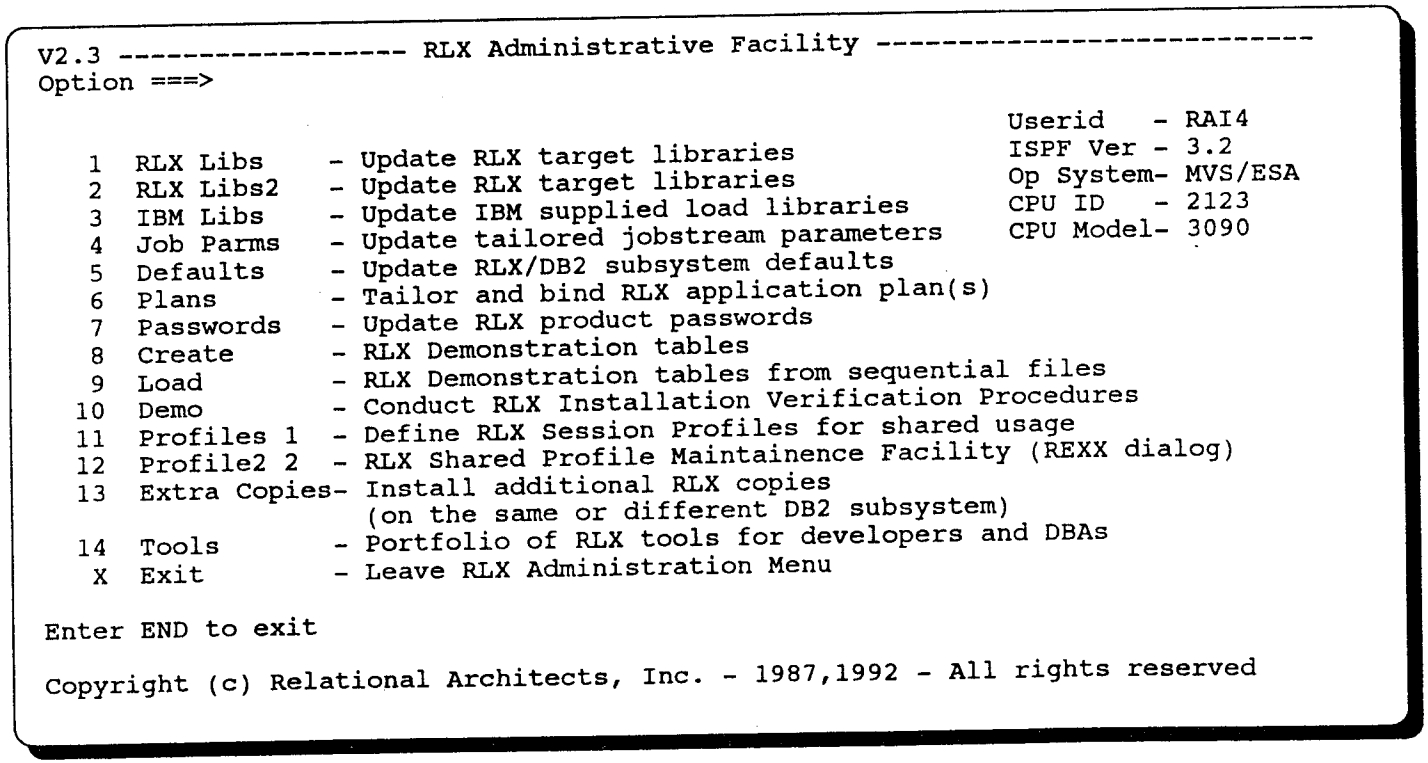


Data errors:

For example: Date value inconsistent with specified date format

```
------------------------- Data Error Recognized --------------- ROW 1 OF 6
Command ===>                                                  Scroll ===> HALF
PSR032 - Expected dash - between ISO/JIS date components

SQL statement location
   Exec containing statement  ===> RLXSINS3
   RLX module detecting error ===> PSQFDTC

Host Variable / SQL Variable Profile
   Host Variable Name         ===> DATE
   Host Variable Value        ===> 12/04/1991
   Host datatype origin       ===> DATE
   SQL Data Type              ===> DATE                  Nulls ===> Y
   SQL Data Length            ===> 10          (Precision when Decimal)
   SQL Data Scale             ===> 0           (0 when not Decimal)
   Date or Time format name   ===> ISO         (Blank when not date or time)
   Date or Time format        ===> YYYY-MM-DD (Blank when not date or time)

Statement executing when the Data Error was detected

INSERT INTO RLXREL3 (INTEGER , DATE , TIME , TIMESTAMP , FLOAT VALUES (:DV1
:IV1 , :DV2 :IV2 , :DV3 :IV3 , :DV4 :IV4 , :DV5 :IV5)

PSQ105 - No row inserted because column value was invalid for its datatype
```


RLX Administration



RLX Profile Defaults

```
 ------------------------ RLX User Profile Facility 2 -------------------------
Command ===>

RLX068 - Your RLX Session Profile was updated successfully
CONTROL service settings
   When RLX Error  ===> F          (C - Cancel   F - Filter   R - Return)
   Error Panel     ===> D          (D - Display  N - No Display)
   When ISPF Error ===> C          (C - Cancel   R - Return)
   Cursor Scope    ===> L          (L - Local    G - Global)
   Variable Scope  ===> L          (L - Local    G - Global)
   Statement Scope ===> L          (L - Local    G - Global)
   Maximum Digits  ===> 9          (Before using scientific notation)
   Tracing Option  ===> 0          (Integer value between 0 and 255)
   RC/LASTCC value ===> S          (S - SQLCODE, N - Nonzero, Z - Zero)
```

```
 ------------------------ RLX User Profile Facility 1 -------------------------
Command ===>

RLX068 - Your RLX Session Profile was updated successfully

Environmental Parameters
   DB2 Subsystem   ===> DSN        (DB2 subsystem with which to connect)
   Retry Count     ===> 0          (Connection retries if DB2 is not active)
   Max CPU Time    ===> 0          (in seconds before work is suspended)
   Max Idle Time   ===> 0          (in minutes before thread is terminated)

Application Plan Selection
   Max Cursors     ===> 50         (Maximum number of cursors referenced)
   CSRs WITH HOLD  ===> 00         (Max cursors maintained across COMMITs)
   Max Update      ===> 50         (Max DELETE, INSERT, and UPDATE statements)
   Isolation Lvl   ===> C          (C - Cursor Stability  R - Repeatable Read)

Data Format Preferences
   Numeric Format  ===> E          (I - Integer  E - Edited Decimal)
   Date format     ===> I          (I - ISO,U - USA,E - EUR,J - JIS,L - Local)
   Time format     ===> I          (I - ISO,U - USA,E - EUR,J - JIS,L - Local)
```

45

PROGRAMMING WITH OBJECTS: A REXX-BASED APPROACH

ERIC GIGUERE AND ROB VEITCH
UNIVERSITY OF WATERLOO

Programming With Objects: A REXX-Based Approach

Eric Giguère
Rob Veitch

Computer Systems Group
University of Waterloo
Waterloo, Ontario, Canada
N2L 3G1

giguere@csg.uwaterloo.ca
rgv@csg.uwaterloo.ca

Introduction

The emergence of graphically-oriented user interfaces (GUIs) on a variety of multitasking platforms gives rise to a whole new set of problems for REXX language implementors. What do you do when a console-oriented language like REXX is to be ported to an environment like Microsoft Windows that lacks any kind of command-line environment? How does a user access the GUI from REXX to create dialogs? What changes are required to a REXX interpreter for it to function in a multitasking environment?

These are some of the issues we tackled in implementing a REXX interpreter, WRexx, for use in the Microsoft Windows environment. This paper discusses our approaches to solving these problems, concentrating for the most part on the REXX-to-GUI interface, where we feel the interesting and original work of this implementation lies. (Readers with no Windows programming experience may wish to read the appendix for a quick overview of Windows.)

Note: Throughout this paper, *Windows* refers to the Microsoft Windows environment, *X11* refers to the base X Window System, *Xt* refers to the X Toolkit and *DOS* refers to MS-DOS/PC-DOS.

1. Adapting The REXX Console Model

The REXX language assumes the existence of a *console* through which it can interact with a user. The `SAY` instruction is the most obvious example:

```
say "Please enter your name:"
pull name
```

This model works well on systems like DOS, CMS, Unix (text mode) and OS/2 (text mode), where a console is the normal mode of operation. It also works well on hybrid systems like X11 and the Amiga, where virtual consoles coexist within the GUI environment. Systems like the Macintosh and Windows, however, do not provide operating system support for consoles. Consoles become the responsibility of the REXX environment.

WRexx uses a virtual console to handle user interaction and tracing, and a separate virtual console for displaying error messages. The consoles are windows that can be moved and resized like any conventional window. Users can also scroll through the console's contents using the cursor keys or the scrollbars. Neither console is displayed until input or output occurs, and once visible remains onscreen until explicitly closed.

WRexx also adds a virtual console stream type to the REXX I/O model:

```
call lineout 'con:My Window', 'Hello, world'
```

The consoles can be used with any of the stream-based functions.

2. UI Options for REXX

While virtual console support allows a REXX interpreter to *function* in a GUI environment, the interpreter will be more useful if it can also *use* the environment. Instead of consoles, REXX programs can use windows, buttons, edit fields and other *user interface objects* to interact with the user.

When designing WRexx we considered three options for adding GUI access to REXX:

1. **Language extensions.** Extending the REXX language to include new instructions and programming structures for building dialogs, menus and so on.
2. **UI-oriented functions.** Adding functions like `CreateMenu()`, `CreatePushButton()`, `ShowWindow()`, etc., as BIFs or through an external function library.
3. **Object-oriented functions.** Adding functions like `UICreate()`, `UISet()`, `UIGet()`, etc. These functions work on generic user interface objects.

There are advantages and disadvantages to each approach. Language extensions make it easy to connect individual objects and events with REXX code:

```
menu "File"
    item "Open..."
        call OpenFile
    item "Exit"
        exit
endmenu
```

But such extensions are also completely non-portable and may require other changes to the REXX language. We rejected this approach because we wanted to remain faithful to the language as defined by Cowlishaw's book [Cowlishaw 90].

Once the function-based approach was chosen, it became a matter of choosing between the two kinds of function libraries: very specific, UI-oriented functions, or more generic, object-oriented functions. We eventually settled on the object-oriented approach (described in the next section) because we felt it would be a more consistent and extensible interface, even though UI-oriented functions are the more traditional approach for REXX extensions.

3. The OOUI Library

The WRexx GUI library is known simply as the "OOUI" (object-oriented user interface, pronouced *oo-ee*) library. It is implemented as a Windows dynamic link library (DLL) and is only needed by REXX programs that wish to access the Windows GUI.

3.1 Objects and Classes

The OOUI library implements a hierarchical class structure of *window objects* such as edit fields, buttons and various containers. Each object has a set of *properties* that determines its current state and behaviour, as well as a set of *methods* to alter that state. The properties, methods and behaviour of an object are defined by its *class*. The library is hierarchical in the sense that each class *inherits* properties, methods and behaviour from a parent class or *superclass*. The *subclass* usually adds new properties or methods to those of the superclass. The current OOUI class hierarchy is shown in Figure 1. It is based for the most part on the window types defined by Microsoft Windows.

C programmers can also use the facilities provided by the OOUI DLL to write their own DLLs to implement new classes and subclasses.

3.2 Object Manipulation

Objects are manipulated from within WRexx using five functions. `UICreate()` creates an object of a given class and `UIDestroy()` destroys an object. `UISet()` and `UIGet()` are used to set and retrieve property values, while `UIMethod()` invokes a method. Objects are identified by *handle* (returned by `UICreate()`) or by *name* (assigned by the user).

Objects are also created hierarchically. Except for objects called *Forms*, each object has a parent object on the screen which affects the child's positioning and other properties. Each object tree is rooted on a Form, which is a top-level (application or dialog) window.

For example, the following code creates a blank Form on the screen and immediately centers it:

```
f = UICreate( '', 'Form', 'visible', 'true', ,
              'height', 100, 'width', 200 )
call UIMethod f, 'centerwindow'
```

This example attaches some text and a button to the Form:

```
f = UICreate( '', 'Form', 'visible', 'false' )
```

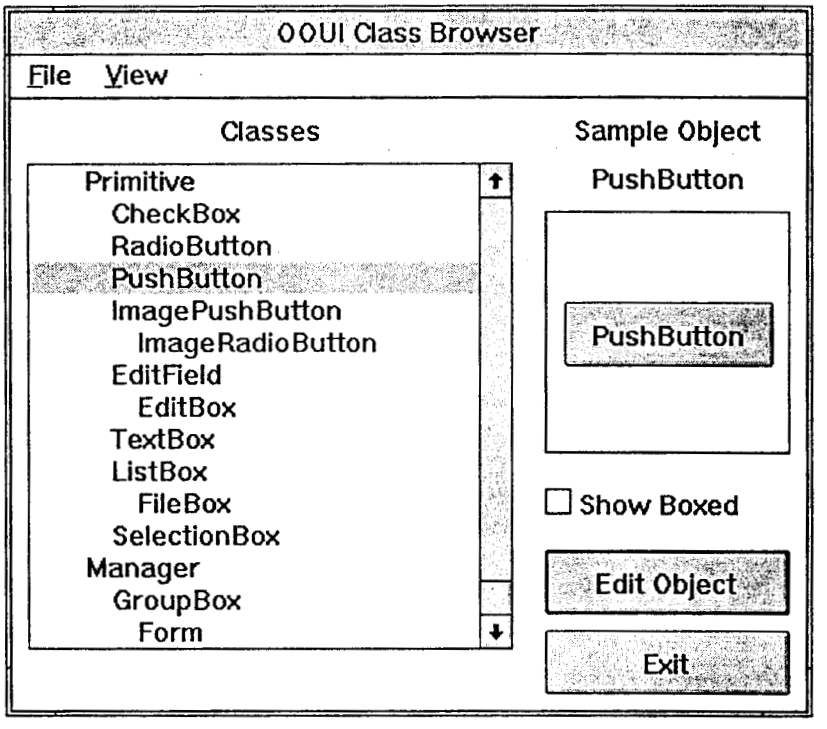


Figure 1: Viewing the OOUI Class Hierarchy

```
t = UICreate( f, 'TextBox', 'caption', 'This is some text' )
p = UICreate( f, 'PushButton', 'caption', 'Press Me!' )
call UISet f, 'visible', 'true'
```

Because the Form is the parent object for both the TextBox and the PushButton, neither child object will be shown until the Form itself is made visible.

Note: Form and GroupBox objects include behaviour (which may be turned off) for automatically resizing and positioning their children, thus freeing the programmer from having to specify absolute coordinates when positioning objects.

When finished with an object, a call to `UIDestroy()` recursively destroys an object and all of its children.

3.3 Events and REXX

Objects will generate events whenever something interesting occurs; for example, when a pushbutton is clicked. These events must be passed to the REXX program that created the objects so that the program can respond to the user. This is done using *event strings* for each object's events. The event string is merely a string that is associated with a specific event. The string will be returned to the REXX program whenever that event occurs. The REXX program checks for pending events by calling the `UIEvent()` function, which will return the next event string. For example, the PushButton object has a "click"

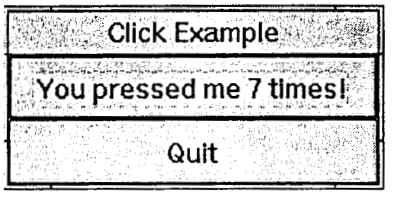


Figure 2: Running 'click.rex'

event signifying that the user has clicked on the button. The following program demonstrates the use of event strings:

```
/* click.rex */

f = UICreate( '', 'Form', 'caption', 'Click Example' )
p = UICreate( f, 'PushButton', 'caption', "You haven't pressed me!", ,
              'click', 'call FirstPress' )
e = UICreate( f, 'PushButton', 'caption', 'Quit', ,
              'click', 'exit 0' )

do forever
    interpret UIEvent()
end

FirstPress:
    call UISet p, 'caption', 'You pressed me once!'
    count = 1
    call UISet p, 'click', 'call NextPress'
    return

NextPress:
    count = count + 1
    call UISet p, 'caption', 'You pressed me' count 'times!'
    return
```

The program creates a form with two pushbuttons and then enters an *event loop*, waiting for user events to occur. When the user presses a button, an event string is returned to the program and the program executes it using the `interpret` statement.

> Notice that **no language modifications or extensions** were necessary to add GUI support to REXX, only clever use of the `interpret` instruction.

In some cases it may not be obvious to which object an event belongs. The `UIInfo()` function can be used to obtain this and other information on the string most recently returned by `UIEvent()`.

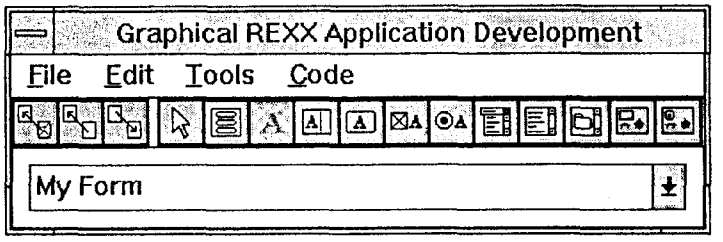


Figure 3: The GRAD Tool

4. Programming With OOUI

After using the OOUI library and REXX, three things become apparent:

1. **The traditional REXX program structure is no longer suitable.** REXX programs typically consist of a single file, augmented with external (and independent) functions. However, even the simplest REXX application under Windows may display several Forms with numerous objects on each form. The single-file approach in this case leads to monolithic programs that take longer to load and are harder to debug. Performance is improved and debugging made simpler (and code reuse encouraged) if an OOUI-based program is split across multiple small REXX files.

2. **Exposing variables across files is extremely useful.** Splitting a program into several files is much more tolerable if variables can be exposed across files. WRexx has been extended so that `procedure expose` will expose variables across file boundaries. (This feature becomes invaluable to the programmer in a very short time.)

3. **OOUI programming is ugly, so automated tools are needed.** Adding object-oriented concepts to a procedural language almost always seems to lead to ugly code, and REXX is no exception to the rule. Writing the REXX programs to display complicated dialogs is itself a complicated process if all the programmer has is a text editor to work with. Tools such as the class browser (Figure 1) and GRAD[1] (Graphical REXX Application Development, Figure 3) can be of immense help.

An issue that also comes up when using the OOUI library is that of multiple independent (i.e., modeless) Forms. There is only one call to `UIEvent()` active at any time (because there is only a single thread of execution within a REXX program), and it may be in a different file or procedure. Problems can then arise due to scoping issues. Luckily, there are few situations where modeless Forms are required. (Problems do not arise with modal Forms because the previously active Form is always disabled before the new one is made active.)

[1] The reader may find it interesting to note that both the browser and the GRAD tool are themselves written in REXX.

5. Conclusions

Virtual consoles and the OOUI function library allow WRexx to thrive in the Windows environment. With them, REXX can be used both as a general-purpose scripting language (which Windows lacks) or for implementing real applications.

Appendix A. A Crash Course on Windows Programming

Readers with no GUI programming experience will discover that there is a substantial learning curve involved in developing for systems such as Microsoft Windows. This section is intended to provide you with enough information to understand the rest of the paper, but for more complete treatments of GUI programming models please refer to the bibliography. (**Note:** The Windows programming model is almost identical to the model used by the OS/2 Presentation Manager. Readers with PM experience should have little trouble understanding the terminology used throughout this paper.)

What is Microsoft Windows?

Windows is a multitasking environment built on top of DOS. It provides a windowing environment, device-independent graphics and inter-application communication (IAC) facilities. Windows applications will not run under DOS, as they use a completely different application programming interface (API) and a different programming model. Windows can emulate a DOS environment (the so-called "DOS box") in which to run DOS programs, but such programs cannot take advantage of Windows' features.

The multitasking model used by Windows is often termed *cooperative multitasking*: each Windows application will run until it voluntarily releases control of the CPU, at which time Windows will switch control to another application. Well-behaved applications must ensure that they give up the CPU at small time intervals. Unlike OS/2, Windows is not a preemptive system, nor does it support threads (lightweight processes). Because of this there are no semaphores or other means of task synchronization.

Programs and User Interaction

Like other GUI platforms, Windows uses an *event-driven* programming model. Applications create one or more windows, to which are attached user interface objects such as buttons and menus. The programs then wait for user events (such as clicking on a button or pressing a key) to occur. When an event occurs, Windows sends a *message* to the application that "owns" the event. The message is added to the end of a queue which the application continually checks for new messages. Each Windows application has a loop in it to do this (in pseudo-code):

```
do forever
    get next event
    process event
end
```

The same type of loop is used in Macintosh, Amiga and X11 applications. In Windows (and PM) the loop serves mainly to demultiplex the application message queue, dispatching messages to the appropriate *window procedure*. When you create a window (or more accurately, a *window class*) you register a window procedure to handle that window's events, including those that bypass the application message queue.

```
do forever
    get next event
    dispatch event
end

window procedure:
    case message is BUTTONDOWN
        ....
    etc.
end procedure
```

Note that Xt applications (this includes Motif applications) take this demultiplexing one step further by registering *callback routines* for each *event* of interest.

Dynamic Link Libraries

The Dynamic Link Library (DLL) is a method for sharing code and resources between Windows applications. (Windows itself is implemented as a set of DLLs.) A DLL is a run-time library that is loaded into memory on demand and dynamically linked to an application. Applications can call DLL routines just like normal (statically-linked) library routines.

One important feature of a DLL is that it has its own dataspace, shared by all tasks using the DLL. (Note: OS/2 has DLLs as well, but OS/2 DLLs have a separate dataspace for each process.)

Dynamic Data Exchange

Dynamic Data Exchange (DDE) is a form of inter-application communication. Applications communicate by setting up DDE "conversations" using invisible windows and a well-defined protocol. Communication is done by sending messages to these windows. The DDE protocol includes facilities for sending commands and for maintaining data links.

References

[Cowlishaw 90] M. F. Cowlishaw. *The REXX Language: A Practical Approach to Programming, 2nd edition*, Prentice-Hall, 1990.

REXXBITS

LINDA SUSKIND GREEN
IBM

REXXbits

Linda Suskind Green
SAA Procedures Language Interface Owner

IBM
Endicott Programming Lab
G93/6C12
PO Box 6
Endicott, NY 13760

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

May, 1992

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:

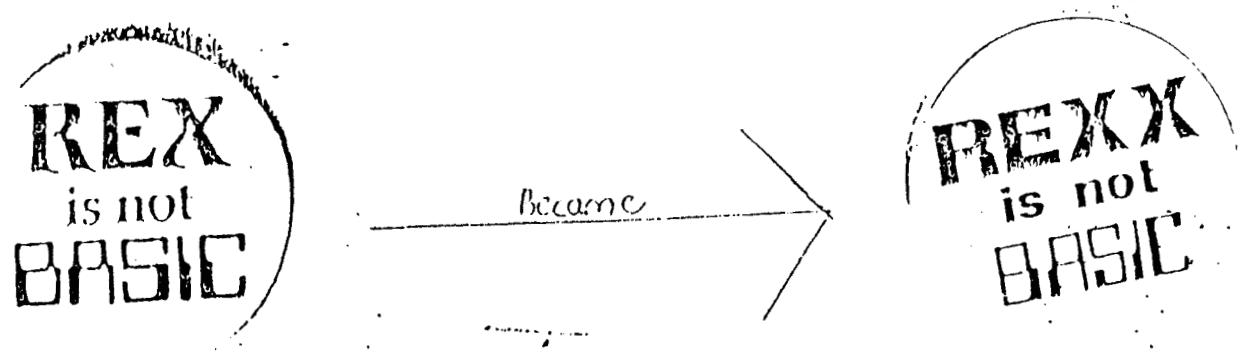
over 6,000,000

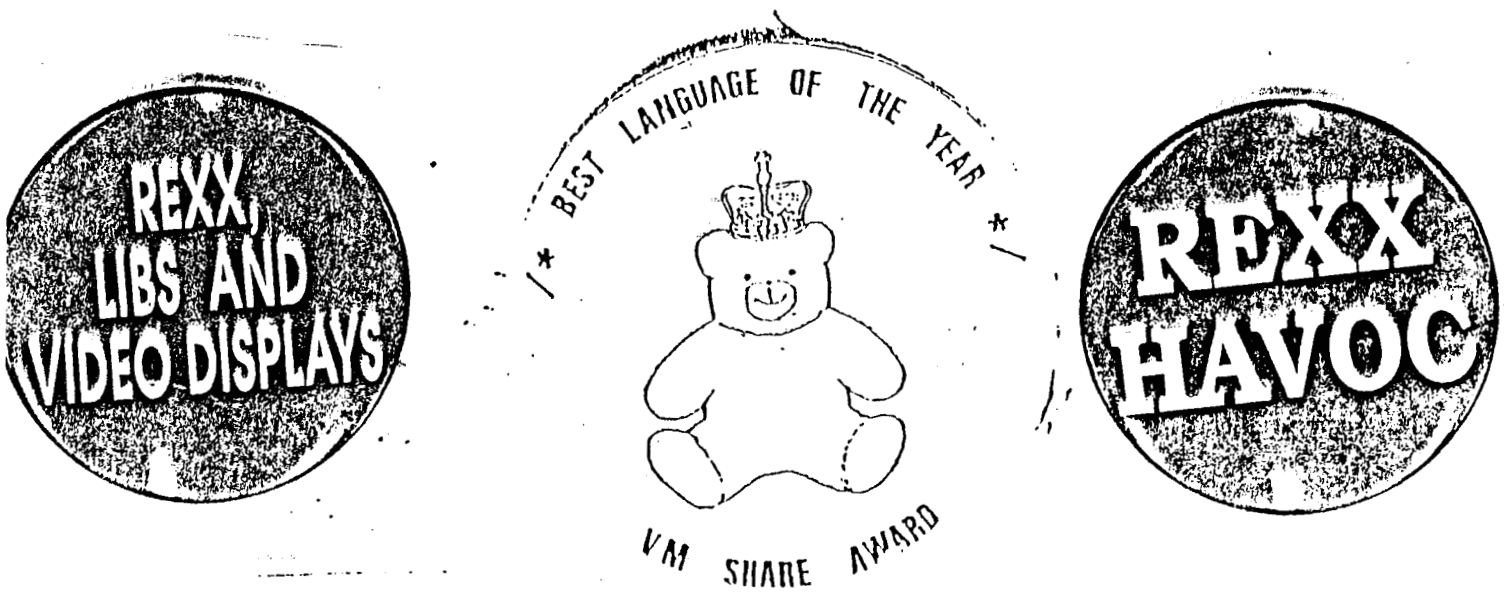
Question is:

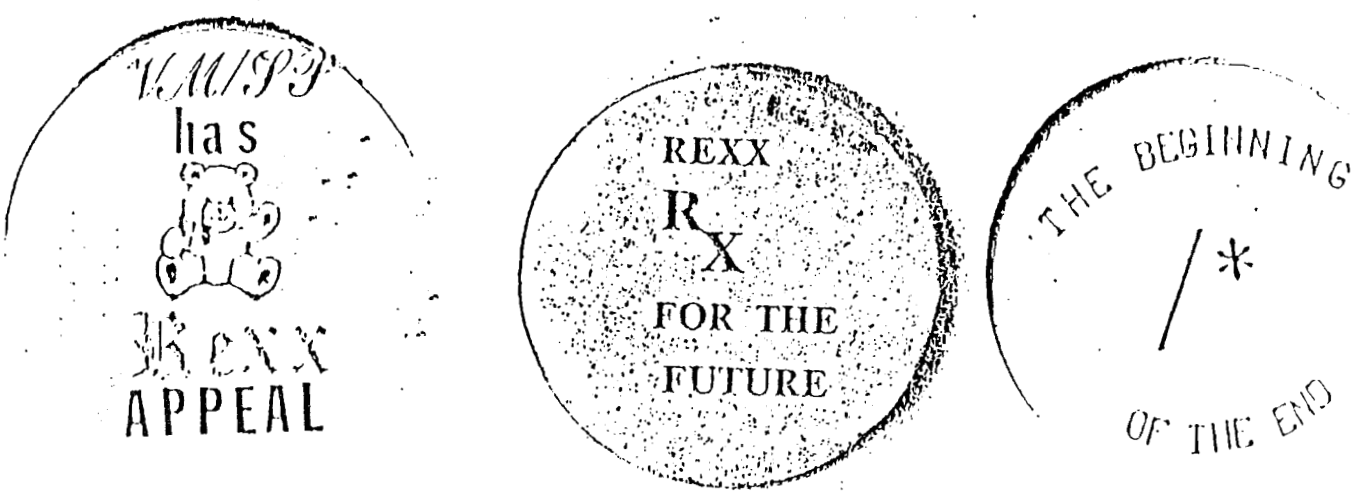
What is the largest known total number of lines of REXX code used in any one company?

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

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
1992: planned for May 3-5, 1992 in Annapolis, MD

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

s 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 3/92

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)

Planned:

- 3 others being worked on

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
- Japan
- Jersey
- Scotland
- Spain
- United States
- Wales

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

REXX Trade Press Article Titles

EXOTIC LANGUAGE OF THE MONTH CLUB

REXX: A beginner's alternative

All Hail REXX

...nat you should know about IBM's soon to be ubiquitous procedural language

For Programmers

REXX — A Multifaceted Tool

WHY IS REXX SO POPULAR?

...EXX:
...EXX the Wonder Language

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

---

nalysis \ *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.

EXX

A USER'S DREAM IN A
YSTEM 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 application architecture 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

REXX Language Level ...

3.46	CMS release 6 with SPE TSO/E ver 2, rel 1 with APAR 370 compiler SAA Procedures Language level 1
3.48	OS/400 rel 1.3
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 REXX implementations for:

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

from 9 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) OS/2 (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!

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"!

COMMUNICATIONS AND EVENT HANDLING WITH REXX

RAINER F. HAUSER
IBM

Communications and Event Handling with REXX

Rainer F. Hauser

May 1992

Some Questions

REXX is a sequential procedure (macro, control or glue) language. Is it really, or could it be that it is actually a programming language? What about REXX and concurrency?

Communications:

Is REXX the right choice for programming communications software? Does it provide the necessary constructs for such programs? What about the performance?

Event Handling:

Is REXX suitable for general event handling? What is missing today for writing such programs? What are the events which fit the paradigm of REXX?

Three REXX Extension Packages:

A kind of answer "by doing" to some of these questions:

- REXXIUCV: REXX Interface to IUCV
- REXXSOCK: REXX Interface to TCP/IP Socket Calls
- REXXWAIT: REXX General Purpose Event Handling with a Central Wait Function

REXX and Concurrency Today

A REXX program can process events sequentially. To do so, it needs the possibility to find out when an event has occurred, but has not yet been consumed.

The following REXX statements determine whether a console event is pending:

```
if externals()>0 then ...
```

When no event is currently pending, it needs the possibility to wait for an event.

The following REXX statements wait for a console event:

```
say 'Enter your name, please.'
parse external name
```

To avoid being blocked despite a pending event which could be processed, it needs the possibility to wait for one event within a given list of events.

Today, a REXX program can

- sometimes not determine whether a specific event is pending
- often not wait for a specific event
- not wait for one event within a given list of events

REXX and Communications

One system facility and the two REXX extension packages REXXIUCV and REXXSOCK provide communications in REXX:

APPC:

APPC is available via the SAA Common Programming Interface Communications (CPI–C) and the Callable Service Library (CSL).

IUCV:

IUCV is a communications facility available on VM systems. The REXXIUCV program provides access to it from REXX on VM/CMS. Therefore, it is a system–dependent communications extension for REXX.

TCP/IP:

TCP/IP is a communications facility available on various different platforms. The REXXSOCK program provides access to it (i.e. to the socket calls) from REXX on VM/CMS. Therefore, it has been designed as a system–independent communications extension for REXX.

REXXIUCV:

Syntax: `result = IUCV(subfunction, arg1, ..., argn)`

Subfunctions:

- INIT, TERM, QUERY, WAIT, ...
- CONNECT, ACCEPT, SEVER, ...
- SEND, RECEIVE, ...

Examples:

```
tempdata = IUCV('CONNECT','RFH',255,'No')
parse var tempdata pathid msglim .
inttype = IUCV('WAIT',600,'NOWAIT')
nextint = IUCV('QUERY','NEXT')
parse var nextbuf . buftype bufpathid rest
...
```

Problems:

- Assembler Paradigm vs. REXX Paradigm
- Special Purpose Wait Subfunction

REXXSOCK:

Syntax: `result = TCPIP(subfunction, arg1, ..., argn)`

Subfunctions:

- INIT, TERM, QUERY, GETHOSTID, ...
- SOCKET, BIND, CONNECT, ACCEPT, CLOSE, ...
- WRITE, READ, SEND, RECV, ...

Examples:

```
inetaddr = 'AF_INET 1291 9.4.3.2'
socketid = TCPIP('SOCKET')
tempdata = TCPIP('CONNECT',socketid,inetaddr)
...
```

Problems:

- C Paradigm vs. REXX Paradigm
- Functions such as CONNECT and READ block the caller
- Data can be encoded as ASCII or EBCDIC

Common Design Decisions

Both packages are based on the following design decisions:

- The status of the communications facility is kept by the REXX extension package and can be determined by the REXX program.
- The status of the communications facility should not be destroyed when the REXX program terminates.
- Individual IUCV primitives or TCP/IP socket calls should be provided as individual function calls to REXX. In other words, there should be a one–to–one mapping between REXX functions and IUCV primitives or TCP/IP socket calls, respectively.
- A REXX program should be allowed to process events selectively as appropriate to the program (and the programmer).
- Return codes are presented to the REXX program in the REXX variable RC.
- Limits such as the maximum length of messages are necessary, but should be easy to change. (Such limits should also make sense to human beings and not to computers. Therefore, values such as 1000 are a better choice than values such as 1024).

Experiences

REXX as a programming language is well suited for communications software, but with the current language features, there are some limitations and inconveniencies:

Conversions:

REXX does not provide functions to convert ASCII strings to EBCDIC strings and vice versa.

```
astring = TCPIP('READ',socketid)
estring = A2E(astring)
...
```

Event Handling:

REXX does not provide functions to wait for one of several expected events. Assume that a REXX program needs to wait for either an IUCV or a TCP/IP event.

```
event = WAIT('IUCV PATH 5','TIME 10MIN')
event = WAIT('TCP/IP READ 4','TIME 10MIN')
event = WAIT('IUCV PATH 5','TCP/IP READ 4')
...
```

REXX and Event Handling

The REXX extension package REXXWAIT on VM/CMS provides basic and advanced event handling in REXX through a central wait function for REXX programs and a low–level interface for REXX extension programs.

State of the Art (The REXX Handbook):

- Amiga REXX: IPC (waiting on message port)
- REXX for Tandem: DELAY function and TACLIO IPC
- REXX for Unix: Plan for IPC (SOCKETS, STREAMS)
- REXXIUCV: IUCV('WAIT',seconds)
- ...

Common Events:

- Keyboard and Mouse: Character Oriented, Block Mode, Window Applications ...
- Time: Relative and Absolute Time, Time Events in Files ...
- Mail: Messages, Notes ...
- Synchronization: Inter–Process Communication, Locks, Semaphores ...

REXXWAIT

WAIT Function Syntax:

```
event = WAIT(event1 args1, ..., eventn argsn)
```

Events:

- Basic Events: CONS, WNG, MSG, ..., MAIL, FILE, TIME
- Additional Events: IUCV, TCP/IP

Examples:

```
event = WAIT('CONS NOREAD','TIME 10MIN')
event = WAIT('CONS','MSG','FILE MY TIMEFILE A6')
event = WAIT('TIME ==:=0:00','ALL')
event = WAIT('IUCV TYPE 3 PATH 15')
event = WAIT('TCP/IP READ 15 WRITE 20 21','CONS')
event = WAIT('TIME 10S','TIME 10:30:15','TIME')
...
```

SETVALUE Function Syntax:

```
result = SETVALUE(event args)
```

QUERYVALUE Function Syntax:

```
result = QUERYVALUE(event args)
```

REXXWAIT REXXTRY Sample Session

Sample session with REXXTRY, the facility to interactively execute REXX instructions:

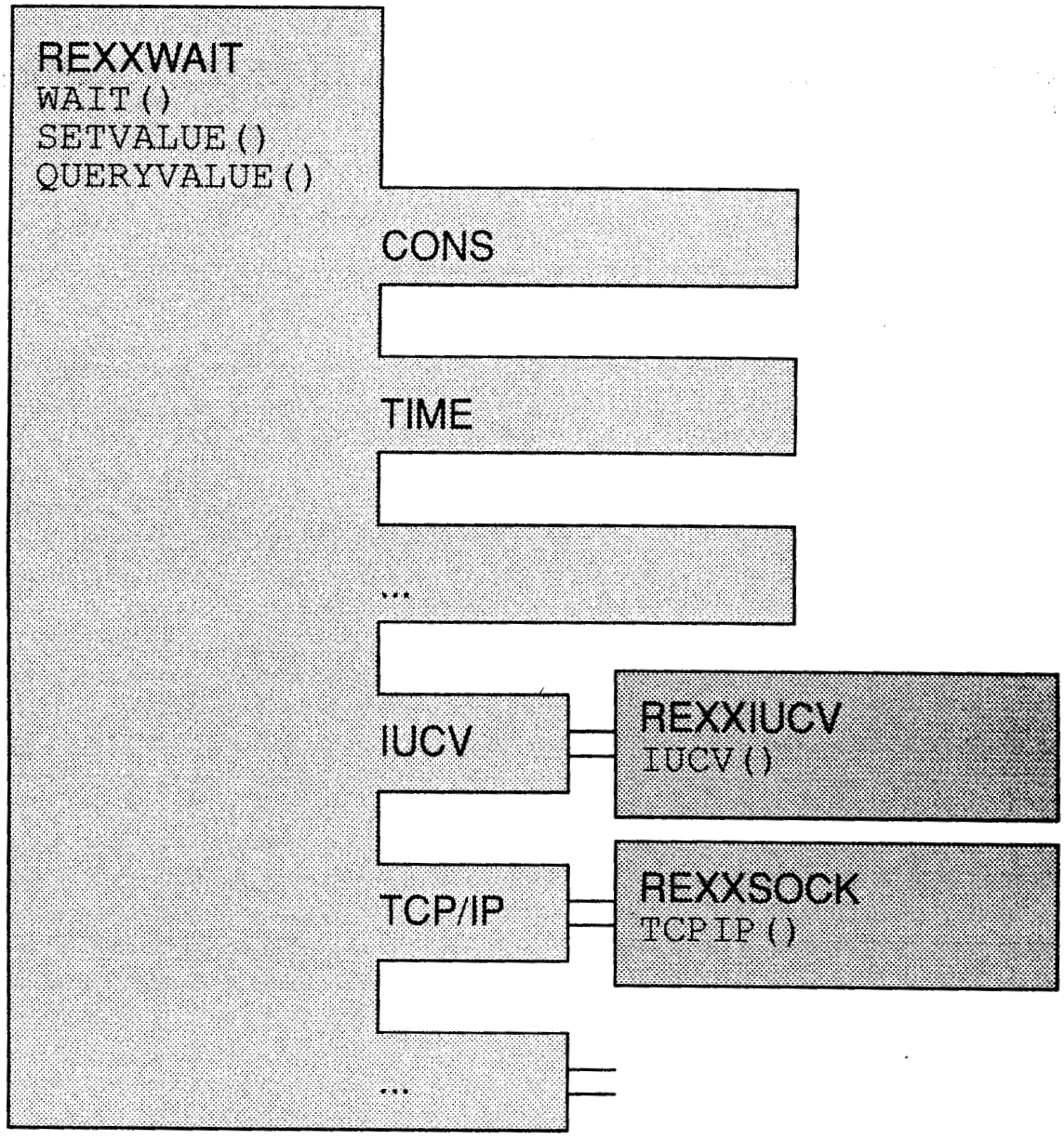
```
say setvalue('MAIL CLASS * NOHOLD')
OFF
R; <REXXTRY> ..............................................
say wait('MAIL','CONS')
RDR FILE 0050 SENT FROM NET    PUN WAS 1991 RECS 0022 ...
File (2404) spooled to HAUSER -- origin ZURLVM1(RFH) ...
MAIL 0050
R; <REXXTRY> ..............................................
say queryval('MAIL 50')
1992/04/16 05:39:41 ZURLVM1(RFH) NOHOLD A 0 0 PROFILE EXEC
R; <REXXTRY> ..............................................
say queryval('MAIL 50 TAG')
2 NetData
R; <REXXTRY> ..............................................
say queryval('MAIL 50 NETDATA')
2 ZURLVM1(RFH) ALMVMD(HAUSER) 19920416133936499588 Ack
R; <REXXTRY> ..............................................
say queryval('MAIL 50 NETDATA 1')
Note
R; <REXXTRY> ..............................................
say queryval('MAIL 50 NETDATA 2')
File A2.PROFILE.EXEC
R; <REXXTRY> ..............................................
```

REXXWAIT REXX Sample Program

Sample program using REXXWAIT and REXXSOCK (without the necessary error testing):

```
...
address command 'RXSOCKFN LOAD'
call TCPIP 'INITIALIZE', 'TCPIP'
if rc<>0 then exit rc
s = TCPIP('SOCKET')
call TCPIP 'BIND', s, 'AF_INET 1952 9.4.3.2'
call TCPIP 'LISTEN', s, 5
call SETVALUE 'TCP/IP SOCKET' s 'NON-BLOCKING'
do forever
  status = TCPIP('QUERY', 'STATUS')
  parse var status init iucvstate reason
  if iucvstate<>'Connected' then exit 3000
  eventd = WAIT('TCP/IP READ' s, 'CONS', 'TIME 1H')
  parse upper var eventd handler rest
  select
    when handler='TCP/IP' then do
      desc = TCPIP('ACCEPT', s)
      parse var desc d caf cport cipaddr
      ...
      call TCPIP 'CLOSE', d
    end
    when handler='CONS' then leave
    otherwise nop
  end
end
call TCPIP 'TERMINATE'
address command 'NUCXDROP RXSOCKFN'
...
```

REXXWAIT Low–Level Interface



Through the low–level interface provided by REXXWAIT, other programs (such as REXXIUCV and REXXSOCK) can export an event name (such as IUCV and TCP/IP) and some branch addresses for communicating with the REXX programs using the functions provided by REXXWAIT.

REXXWAIT Assembler Sample

The following /370 Assembler code shows the code to register an event handler:

```
          LA     R1,RXWPLIST        R1 -> PLIST
          SVC    202                Call CMS
          DC     AL4(1)             Error
or
          CMSCALL PLIST=RXWPLIST,CALLTYP=PROGRAM,...
...
          DS     0F                 Alignment
RXWPLIST  DC     CL8'RXWAITFN'      CMS command
RXWCMD    DC     CL4'SET'           RXW command
          DC     XL4'00000000'      Special fence
RXWNAME   DC     CL8'TCP/IP'        Registered name
RXWWTECB  DC     F'-1'              Address of ECB
RXWWAIT   DC     F'-1'              BAL address WAIT
RXWWAITE  DC     F'-1'              BAL address WAIT-E
RXWSETV   DC     F'-1'              BAL address SETV
RXWQRYV   DC     F'-1'              BAL address QRYV
          DC     F'-1'              Not used yet
          DC     F'-1'              Not used yet
          DC     F'-1'              Not used yet
...
```

The program registers the event handler name 'TCP/IP' and provides the following BAL or ECB addresses (or F'–1'):

- For WAIT: Wait ECB, WAIT and WAIT–E
- For SETVALUE: SETV
- For QUERYVALUE: QRYV

REXXWAIT Register Conventions

When REXXWAIT passes the control to an event handler, the following registers are used:

```
R0    length of arguments
R1    pointer to arguments
R2    call sequence flag (for WAIT only)
R12   address of exit routine (base register)
R13   save area for general purpose registers
R14   return address
R15   same as R12
```

The event handler passes the following data in the registers back to REXXWAIT:

```
R0    length of result string
R1    pointer to result string
R15   return code (0, 1 or error return code)
```

Example:

Event Handler 'ABCD':

```
WAIT('ABCD This is the argument string')
  == 'ABCD This is the result string'
```

The event handler sees the arguments 'This is the argument string' and returns the result 'This is the result string' to the REXXWAIT program which subtracts or adds the event handler name 'ABCD', respectively.

Considerations

Portability:

Some of the basic events are generally available on all operating system platforms on which REXX is implemented. Other events require different arguments. Again others may not be available at all.

```
WAIT('TIME 10:30:15')
WAIT('TIME 2HOURS 15MINUTES')
WAIT('CONS READ')
WAIT('FILE MY TIMEFILE A6')
WAIT('FILE C:\REXXSYS\MY.TIM')
WAIT('FILE' mytimefile)
SETVALUE('MSG ON')
SETVALUE('SMSG VMCF')
```

Operating System Support vs. REXX Support:

The same functions can be provided to a REXX program either as Operating System facilities or as REXX built–in functions:

- EXECIO vs. linein() and lineout() etc.
- WAKEUP vs. wait() etc.

The WAKEUP program (version 5.5) has strongly influenced the design and implementation of REXXWAIT!

REXXTOOLS/MVS

EARL D. HODIL
CHICAGO-SOFT

REXXTOOLS/MVS

A Toolkit for MVS Programmers

REXX Symposium
Annapolis, Maryland
May 5, 1992

Earl D. Hodil
Chicago-Soft, Ltd.
45 Lyme Road, #307
Hanover, New Hampshire 03755
Phone: (603) 643-4002
FAX: (603) 643-4571
MCIMAIL: CHISOFT

Introduction

What is REXXTOOLS/MVS?

- REXXTOOLS is a collection of assembler-based functions and utilities designed to help the REXX programmer be more productive.

Who will use REXXTOOLS/MVS?

- Application programmers - ISPF Dialogs, batch jobs, etc. for end-users.
- System programmers - function packages, utilities for themselves and application programmers.

Introduction

REXXTOOLS Components (REXX perspective):

* REXX function package - 29 new functions
* REXX host command environment - ADDRESS REXX
* REXX compiler - encapsulates REXX programs in standalone load modules
* Sample applications - TSO utilities, programming examples

Introduction

REXX Functions:

- `ABC = MYFUNC(ARG1, ARG2)`
- ARG1 & ARG2 are arguments passed to the function
- MYFUNC returns a value
- MYFUNC could have side-effects (i.e., set other variables in the caller's variable pool)

REXX Subroutines:

- `CALL MYFUNC ARG1 ARG2`
- Arguments have the same meaning as for functions
- Can optionally return a value (RESULT special variable)

Introduction

How are functions and subroutines developed?

- **REXX - internal and external subroutines and functions**
- **Compiled/assembled languages (a la REXXTOOLS)**
- **Search order**
- **Function packages**
  - **Groups related functions together**
  - **Pre-loaded at environment initialization**
  - **Can't be developed in REXX (unless you have REXX compiler!)**

Introduction

REXX Host Commands:

- **any expression not identified as a language construct**

```
"VGET (ZUSER) SHARED"
```

- **expression is evaluated and string is passed to host command environment routine.**
- **ADDRESS instruction is used to switch host command environments.**

```
ADDRESS TSO "LISTA ST H"
```

- **Each REXX environment has a default host command environment**
- **Parameters module host command environment table maps environment names to routines.**

VSAM Functions

Why VSAM?

* **Lots of existing VSAM files**
* **Better for multi-user applications than ISPF Tables**

What is supported?

* **All VSAM dataset organizations**
  - **Key-Sequenced Data Set**
  - **Entry-Sequenced Data Set**
  - **Relative Record Data Set**
  - **Linear Dataset (sort of)**
* **Interface is patterned after DFP macros**

VSAM Functions

Opening and Closing VSAM Datasets:

* **CALL OPEN 'VSAM', ddname [,acboptions]**
* **CALL CLOSE 'VSAM', ddname**
* **ACB options string**

  "(ADR,SEQ,NDF)"

Reading and Writing Records:

* **CALL GET ddname [,key] [,rploptions]**
* **CALL PUT ddname, record [,key] [,rploptions]**
* **RPL options string**

  "(KEY,DIR,GEN)"

Deleting Records:

* **CALL ERASE ddname**

VSAM Functions

Other VSAM Functions

* CALL ENDREQ ddname
* CALL POINT ddname [,key] [,rploptions]
* CALL VERIFYV ddname

VSAM Functions

ACB Options:

- Most ACB options are supported:

  ADR, CNV, KEY, DIR, SEQ, SKP, IN, OUT, DFR, NDF, NIS, SIS, NRM, AIX, NRS, RST

* Stay in effect from OPEN to CLOSE

RPL Options:

* Most RPL options are supported:

  ADR, CNV, KEY, DIR, SEQ, SKP, ARD, LRD, FWD, BWD, NSP, NUP, UPD, KEQ, KGE, FKS, GEN

- Shared between calls for each ddname.

* Stay in effect until changed

VSAM Functions

Special Variables:

* **Used to return information from functions**
* **RC and REASON - usually straight from VSAM**
* **OPEN:**
  - **$RXTTYPE**
  - **$RXTLRCL**
  - **$RXTCNVL**
  - **$RXTKEYO**
  - **$RXTKEYL**
  - **$RXTRECS**
  - **$RXTHRBA**
  - **$RXTERBA**

VSAM Functions

Special Variables (continued)

* **GET/PUT:**
  - **$RXTKEY**
  - **$RXTRBA**
  - **$RXTRECL**

VSAM Functions

Sample REXX program:

```
/* REXX */
  ADDRESS TSO "ALLOC FI(RXTKSDS) DA(RXTKSDS.DATA) SHR REU"
  CALL OPEN 'VSAM', 'RXTKSDS', '(KEY,DIR,IN)'
  CALL TPUT "ENTER KEY OR 'END':", 'ASIS'
  KEY = TRANSLATE(TGET(, 'WAIT'))
  DO WHILE KEY <> 'END'
    CALL GET 'RXTKSDS', KEY, '(DIR,GEN,KEY)'
    IF RC <> 0 THEN
      SAY 'NO MATCH FOR KEY='KEY' FOUND.'
    ELSE
      SAY 'RECORD='RESULT
    CALL TPUT "ENTER KEY OR 'END':", 'ASIS'
    KEY = TRANSLATE(TGET(, 'WAIT'))
  END
  CALL CLOSE 'VSAM', 'RXTKSDS'
  ADDRESS TSO "FREE FI(RXTKSDS)"
EXIT
```


MVS Supervisor Services

Access to System Services

- **Patterned after and interfaces to application macros**
- **Problem state only**
- **All are task related**
- **Functional areas:**
  - **Virtual Storage Management**
  - **Resource Control**
  - **Security**
  - **Operator Communication and logging**

MVS Supervisor Services

Virtual Storage Management:

STGAD = GETMAIN(amount [,subpool] [,loc] [,bndry] [,fill])

CALL FREEMAIN addr, amount [,subpool])

Uses:

- **Communicating with non-function asm programs**
- **Multi-tasking REXX application inter-task communication**

MVS Supervisor Services

Resource Control

- **Problem: How to share a resource between asynchronous processes?**
- **Reserving and freeing an arbitrary resource:**

  CALL ENQ major, minor [,control] [,scope] [,reqtype]

  CALL DEQ major, minor [,scope] [,reqtype]
- **Halting execution until conditions are right:**

  CALL WAIT 'ECB', ecbad [,longwait]

  CALL WAIT 'ECBLIST', ecblad [,eventno] [,longwait]

  CALL WAIT 'SEC', seconds
- **Signaling event completion:**

  CALL POST ecbad [,compcode]

MVS Supervisor Services

Securing a resource:

* System Authorization Facilty (SAF)
  - Works with major security packages (RACF, ACF2, etc.)
  - Router Table must be set up
  - SAF must be active
* Problem state only - can't counterfeit userid
* Modelled after RACROUTE macro:

  CALL RACROUTE 'AUTH', entity, [,class] [,attr] [,dstype]
  [,volser] [,oldvol] [,appl] [,owner] [,acclv] [,racfind]
  [,generic] [,reqstor] [,subsys]

MVS Supervisor Services

Operator Communication and Event Logging:

* Single and Multi-line console messages:

  wtoid = WTO(msgtext [,msgcount] [,route] [,desc])
  - msgcount > 0 uses multi-line format
  - no direct control of routing and highlighting

  CALL DOM wtoid
  - removes non-scrollable messages
  - not an error if message is already gone

* Two-way communication:

  CALL WTOR msgtext [,waitsecs] [,route]
  - Does wait internally
  - Handy for batch jobs

MVS Supervisor Services

Operator Communication and Event Logging (continued):

* **Logging events:**

  CALL WTL msgtext

  - **Fast way to keep track of program execution**

TSO Services

Input/Output Functions:

* **REXX SAY instruction is limited**
  - **PUTLINE only**
  - **No formatting control**
* **REXX PULL instruction:**
  - **GETLINE only**
  - **Does have nice parsing**
  - **Complicates matters when using the data stack**
* **ISPF Dialog Manager**
  - **Must be under ISPF command to use**
  - **Can't use in certain environments like TEST**

TSO Services

REXXTOOLS TPUT

* **CALL TPUT string [,tptype] [,tpwait] [,tphold] [,tpbreak]**
* **Line mode:**
  - **tptype: 'ASIS' or 'EDIT'**
  - **ASIS like CLIST WRITENR:**

```
CALL TPUT 'Enter Your Name:', 'ASIS'
```

  - **No echo prompting:**

```
CALL TPUT 'Enter Your Password:'||'24'X,,
   'ASIS'
```


TSO Services

REXXTOOLS TPUT

* **Full-screen mode:**
  - **tptype: 'NOEDIT' or 'FULLSCR'**
  - **string argument contains 3270 data stream:**

```
DS = 'F5C3'X||SBA(1,1,80)||'1DF8'X||'ENTER YOUR NAME ===>'||,
     '1DC813'X||SBA(1,40)||'1DF8'X
CALL TPUT DS, 'FULLSCR'
```

Miscellaneous Services

Stem Handling Functions

- REXX stem variables - i.e., variables with a dot
- Sorting arrays (stems) with numeric subscripts:

  CALL STEMSORT *stemname* [,startsub] [,stemcount] [,sortfields]

  - sortfields like DFSORT or SYNCSORT

    "(start,length,type,direction)"

  - Uses Heapsort algorithm (see Wirth)
- Displaying arrays with numeric subscripts:

  CALL STEMDISP 'BROWSE', stemname [,startsub] [,stemcount] [,title] [,panel]

  - Uses BRIF service for display (no dataset)

 23

Miscellaneous Services

String Handling Functions:

- Difficult parsing
  - Parsing where the location and frequency of the delimiters is difficult to predict.
  - Example:

    DSN('abc.efg(one)') KEYWORD2(two)

  - CALL PARSETOK string, stemname [,nbd] [,blankopt] [,dropopt]

 24

Miscellaneous Services

PARSETOK Example:

```
STRING = 'DSN(ABC) NONAME'
CALL PARSETOK STRING, "TOK.", "()", "BLANKS"
/* TOK.0 = 6; TOK.1 = 'DSN'; TOK.2 = '(';
TOK.3 = 'ABC'; TOK.4 = ')'; TOK.5 = ' ';
TOK.6 = 'NONAME' */
```


Miscellaneous Services

String Handling Functions (continued):

- **Sorting words**
  - **CALL WORDSORT string [,diropt]**
  - **diropt - Asending or Descending**
  - **Useful for sorting indexes into arrays:**

```
A.C = 5
A.A = 1
A.B = 2
INDEX = 'C A B'
INDEX = WORDSORT(INDEX)
/* INDEX = 'A B C' */
DO I = 1 TO WORDS(INDEX)
  SAY VALUE("A."WORD(INDEX,I))
END
```

Miscellaneous Services

MVS/Quick-Ref Function:

CALL QWIKREF fastpath, stemname [,maxlines] [dropopt]

- **fastpath just like QW command:**

  topic=item

  "M=IEF450I"

- **Possible use: trouble ticket automation**

Miscellaneous Services

Conversions:

- **Useful when working with existing VSAM files**
- **370 Packed decimal to REXX decimal**

  CALL P2D packnum [,scale]

```
PACKNUM = '1020000C'X
PRINTNUM = P2D(PACKNUM,2)
/* PRINTNUM = 10200.00 */
```

- **REXX decimal to Packed decimal**

  CALL D2P printnum [,n]

```
PACKNUM = D2P(100.45,5)
/* PACKNUM = '000010042C'X */
```

ADDRESS REXX

* **Issuing a command:**

```
ADDRESS REXX
"THIS IS A HOST COMMAND"
```

* **RXTADDRX REXX program**

```
/* REXX */
SAY ARG(1)
RETURN 4
```

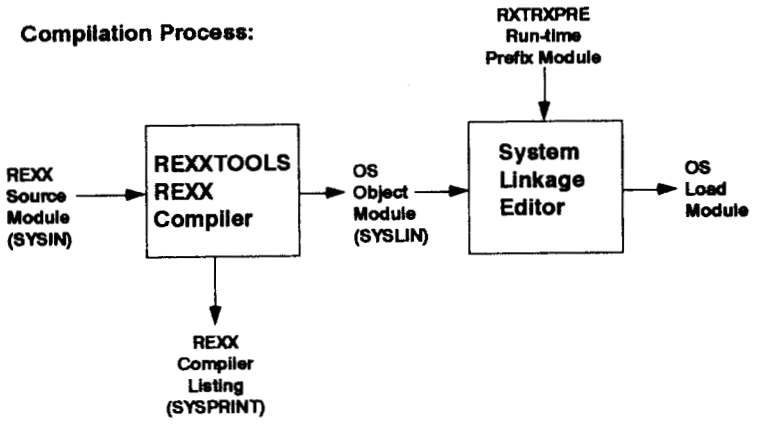
* **Argument: host command string**
* **Must return numeric return code**
* **Limitation: no way to access calling program's variables**

REXX Compiler

* **Compiles REXX programs into standalone, 31 bit, load modules.**
* **Full REXX language supported (including INTERPRET)**
* **No transient library, and no licensing for object modules**
* **Load modules can be used for:**
  - **REXX functions (function packages)**
  - **TSO commands**
  - **Batch programs**
* **Parmlist type is determined dynamically**
* **Program source is included in the load module:**
  - **Source can be compressed (50-80+% compression)**
  - **Source can be encoded (renders it unreadable)**

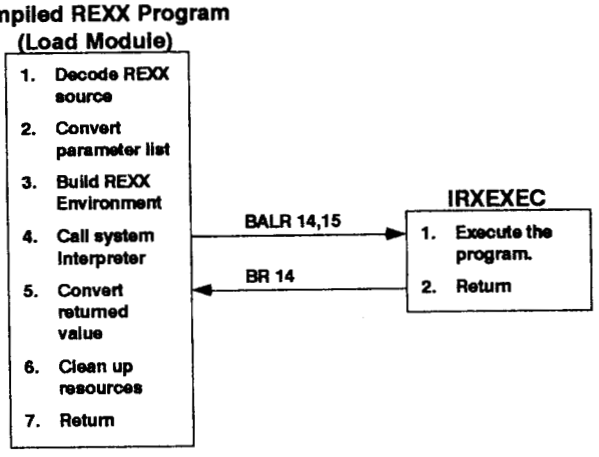
REXX Compiler

Compilation Process:



REXX Compiler

Relationship of compiled REXX programs to system interpreter:



REXX Compiler

Compiler Listing

```
1
-
ORXC 01.02.01          DSN=BI22EDH.USER.EXEC NAME=SAMPREXX
0
                REXXTOOLS/MVS REXX COMPILER  V01.02.01   18 Jan 1992 12:11:48
                COMPILING FROM BI22EDH.USER.EXEC ON VOLUME 780091 (3390)
                CURRENT USERID IS BI22EDH
                OPTIONS ARE: COMPRESS XREF VERSION(01.01.01) NAME(SAMPREXX)
```


REXX Compiler

Compiler Listing (continued)

```
1
-
ORXC 01.02.01          - A sample REXX program
0                      SOURCE LISTING
0               LINE
                   1   /* REXX - A sample REXX program */
                   2   ADDRESS TSO       /* establish host command environ.*/
                   3   /* Get the current date and
                   4      write it to the terminal */
                   5   today = date()
                   6   say 'today is 'today
                   7   /* now loop for awhile */
                   8   Do i = 1 to 30
```


REXX Compiler

Compiler Listing (continued)

```
 9     Say 'The time is now:     'time()
10     Select
11       When (i = 10) Then
12          Say "going..."
13       When (i = 20) Then
14          Say "going..."
15       When (i = 30) Then
16          Say "gone."
17       Otherwise
18          NOP
19     End    /* end of Select */
20   End   /* end of do i = 1 to 30 */
21   /* return to our caller */
22     exit
```


REXX Compiler

Compiler Listing (continued)

```
1
-
ORXC 01.02.01          - A sample REXX program
0                      COMPRESSED SOURCE LISTING
0              LINE
                  1    ADDRESS TSO
                  2    today=date()
                  3    say 'today is 'today
                  4    Do i=1 to 30
                  5    Say 'The time is now:     'time()
                  6    Select
                  7    When (i=10) Then
                  8    Say "going..."
```


REXX Compiler

Compiler Listing (continued)

```
 9   When (i=20) Then
10   Say "going..."
11   When (i=30) Then
12   Say "gone."
13   Otherwise
14   NOP
15   End
16   End
17   exit 0
```


REXX Compiler

Compiler Listing (continued)

```
1
-
ORXC 01.02.01          - A sample REXX program
0                      SYMBOL CROSS-REFERENCE LISTING
0              SYMBOL                  REFERENCES
               ADDRESS                 2
               DATE                    5
               DO                      8
               END                     19 20
               EXIT                    22
               I                       8 11 13 15
               NOP                     18
```

REXX Compiler

Compiler Listing (continued)

```
OTHERWISE           17
SAY                 6 9 12 14 16
SELECT              10
THEN                11 13 15
TIME                9
TO                  8
TODAY               5 6
TSO                 2
WHEN                11 13 15
```


REXX Compiler

Compiler Listing (continued)

```
1
-
ORXC 01.02.01          - A sample REXX program
0
               COMPILATION FINISHED
               ELAPSED TIME: 0.938795 (SEC)  CPU TIME: 0.60 (SEC)
               COMPRESSION: 523 BYTES COMPRESSED TO 204 BYTES. 60.99%
   COMPRESSION
               SOURCE RECORDS READ       22
               OBJECT RECORDS WRITTEN    12
               LIST   RECORDS WRITTEN    88
```

REXX Compiler

Compiler Benefits:

* Prevents unauthorized modifications to distributed REXX programs
* Saves DASD space
* Reduces run times
  - Load time reductions of 70+% (100% for function packages)
  - Execution time reductions 10-15%
  - Best profile for reducing time:
    + medium-to-large REXX program
    + executed frequently
    + short execution path

 41

REXX Compiler

PROCTSO EXEC

* Utility function for parsing arguments like CLIST PROC statement
* Before and after compilation comparison:

ITEM	BEFORE	AFTER	COMMENT
Bytes of code	21349	4578	(78.56% compression)
CPU secs/call	0.08	0.02	(0.06 sec. saved; %75 reduction)

* Executed approx. 1000/day (savings of 60 seconds)
* Assuming $1000.00/hour CPU time:
  - Saves $16.00/day
  - Saves $6080.00/year

 42

REXX2001—CHOSEN LANGUAGE OF MAN AND MACHINE

MARC VINCENT IRVIN
MANAGEMENT VISIONS INSTITUTE

SPEAKER: Marc Vincent irvin

FROM: Management Visions Institute

EVENT: REXX Symposium 1992 (May 4th and 5th)

DATE: Copyright 5/92

OBJECTIVE: Show NL reality, and power NL can give REXX

TITLE: REXX2001 - Chosen Language of Man and Machine

THESIS: Subtle adjustments to REXX could make it the premier language for intelligent systems and occasional programmers.

PREMISE: If science fiction is any barometer, we are heading toward computers able to understand what we say, and say what they understand. Many feel that day is far away because any system that smart requires human intelligence; true human intelligence, according to AI experts, won't be in our life times. Those experts are wrong. A language that needs no programming class, and responds coherently to English could, according to my experiments with REXX, Expert Systems (ES), and Natural Language (NL), be on Personal Computers (PCs) in no time. The secret to achieving the goal ahead of schedule lies in purging all the unnecessary baggage left over from the evolution of computers.

When computers were first conceived they were expected to handle the same kinds of information people do: words, numbers, and symbols. After their invention reality set in with bits and bytes, disks and tapes, sequential and random files, relative and hierarchical databases, and on and on and on... REXX and cover functions can do away with most of the junk that waste programming time. Once purged REXX, ES, and NL can work together to produce a language that occasional programmers will love, and professional programmers can build intelligent systems with.

EMPOWERING CODERS FOR THE 21ST CENTURY

Hardware computing power has grown geometrically over the past twenty five years. Software computing power has grown very little. It is easy to see why this is. The hard problems in hardware have been bridged and standardized because hardware is directed by a predictable element - computer programs. On the other hand, software has no such luxury. The hard problems in software are precipitous and transient because software is directed by an unpredictable element - human programmers. Achieving similar gains is not impossible, however. The hard problems in software can be bridged and standardized if software is directed by predictable elements that cultivate the unpredictable natures of human programmers. Below is a list of road blocks programmers face, a proposed set of solutions, and a some recent experiments devoted to empowering programmers.

Road blocks to empowering programmers for the 21st Century.

1. Differing data, field, and integer types.
2. Differing call formats and complex command syntax.
3. Differing sub-system interfaces and data access methods.
4. Lack of real-time development and run-time features.
5. Lack of interactive and friendly development methods.
6. Lack of cognitive psych, decision support, and AI models.

Solutions to empowering programmers for the 21st Century.

1. Dynamic data typing by use on words, numbers and symbols.
2. Common call methods with toggling for special syntax use.
3. Cover functions whose inputs look the same to all users.
4. Add date/time based initiators and scripts.
5. Workspaces, smart debug features, and natural language.
6. Intergrate Rule, Case, Genetic, Object, & Neural Models.

My experiments with empowering programmers for 21st Century.

1. Using REXX as a platform gets around first road block.
2. RUN() uses interpret not CALL, and ES/NL options toggle.
3. FILECHNG, REXXRDR, and REXXWRTR will standardize all I/O.
4. CLKQUEUE gives temporal power needed for smart programs.
5. REXXCALC w/APL's online tools and PARACODE NL syntax ANS.
6. All empowerments put in CLKRULES' leave room for more.

FILECHNG is a file copy utility with options that:

1. Finds fields and replaces them with other fields.
2. Selects portions of a file or its records to work on.
3. Input can be from disk, reader, or VM command.
4. Sequence checks, purges dupes, and writes change reports.
5. Replace field can be used to select, purge, insert data.
6. One powerful option puts code wherever FINDS occur.

One problem, involving RACF based MVS security, required a list to be made of TSO users that were given IDs, but had never used their IDs. Only two passes of file change were done on an input file that contained all the multi-record RACF reports of the TSO IDs not used in the last 60 days. A sample set of records from RACF ID report follows...

```
USER=TSOUSR   NAME=TED BUNDY   OWNER=SYSTEM  CREATED=88.289
 DEFAULT-GROUP=SYS1    PASSDATE=00.000  PASS-INTERVAL=60
 ATTRIBUTES=NONE
 REVOKE DATE=NONE   RESUME DATE=NONE
 ...
 NO-MODEL-NAME
 LOGON ALLOWED   (DAYS)     (TIME)
```

Below is REXX code that 1) selects the records, and 2) formats them into single lines for examination and display.

```
/* MVI */
'FILECHNG LISTUSER ASOF0392 A PASSDATE WORK A',
    '*PICKRECS',    /* IF REPLACE FIELD = // THEN PICK REC */
    'USER=(1 15) //',     /* PICK RECORDS WITH USER'S ID */
    'SSDATE=00 //',/* PICK RECS FOR NEVER USED IDS */
    'NO-MODEL //', /* PICK REC THAT WILL ACT AS RPT END */
IF RC ^= 0 THEN EXIT 100
'FILECHNG PASSDATE WORK A = = =',
    '*RECDLM(NO-MODEL)', /* MAKE ONE REC OF MANY RECS */
    '*PICKRECS',    /* IF REPLACE FIELD // THEN WRITE REC */
    'SSDATE=00.000 //',  /* PICK NEVER USED RECORDS ONLY */
    /* NOTE, // CAN BE FOLLOWED BY AN EXITNAME TOO       */
    '*OUTEXIT(PASDAT:)'  /* TELL FILECHNG NAME OF CHK LGC*/
IF RC ^= 0 THEN EXIT 200
EXIT 000

PASDAT:  /* THIS ROUTINE IS READ/INTERPRETED BY FILECHNG */
PARSE VAR $REC 1 'USER=' UID ' ',
               1 'CREATED=' ADDAT ' ',
               1 'SSDATE=' PASDAT ' ',
               1 'INTERVAL=' PWINT ' '
IF PASDAT = '00.000' & ADDAT <= '92.004' & PWINT = '60',
   THEN SAY 'USERID ('UID') NEVER USED SINCE ADD ON' ADDAT
/* TO CHG O/P REC PUT VAL IN $REC, TO DEL PUT '' IN $REC */
DOC:
SAY 'REXXNAME: PASSDATE      '
EXIT 000
```

CLKQUEUE is a scheduling utility with options that:

1. Run VM commands based on "date" and/or "time" requested.
2. Requests may be run once or requeued every n days.
3. Commands can be rerun every n hours, minutes, or seconds.
4. Time scripts are possible as CLKQUEUE can call itself.
5. Runs have return codes useable by later clock requests.
6. Its powerful options execute all kinds of REXX code.

Below is a ad-hoc sampling of the many ways that REXX code can be invoked on a date and time basis.

```
CHKRTC:  92/03/22 1 . . 0 IF LIBSFND() THEN 'ERASE UID LIB'
LOVE:  93/02/13 09:00:00 1.H1*17:00.93/02/14 0 0 0,
   MSG * DON'T FORGET THE VALENTINES DAY FLOWERS.
VMUSERID 92/03/22 03:00:00 1 . . 0,
  RUN(VMUSERID:)   /* execute the command beneath EOF */
  IF RC = 0 THEN DO
    'STATE VMUSERID DATA A'
    IF RC ^= 0
      THEN SAY 'ERROR BUILDING VM USERID FILE.'
      ELSE SAY 'VM USERID FILE BUILT OK.'
    END
* RUN NEXT COMMAND ONCE EVERY WEEK...
  GRPPRTADMBKT: 92/04/30 10:00:00 07 92/04/23 10:04:34 0,
    IF GRPRPTADMBKT = 0,
       THEN DO
        'CMSQ RACFMVS GRPRPORT GRP ADMBKT'
        IF RC^=0 THEN SAY 'CLKQUEUE ERROR RUNNING RACF RPT'
        END
       ELSE SAY 'CLKQUEUE ERROR RUNNING GRPPRTADMBKT AT',
            RUNTIME'.'
* CHECK THE NETWORK EVERY 10 MINUTES...
CHKNET: 92/04/28 13:10:00 01.M10 92/04/28 13:00:09 0 CHKNET
CMD1 92/01/22 14:30:00 1 0 0 0 DIRLOG RSCS
CMD2RC 92/01/22 23:59:00 01.M10 0 0 0 CP QUERY RSCS
 92/01/22 23:59:00 01.M10 0 0 0,
    IF CMD2RC = 45 THEN MSG OP *** RSCS IS DOWN! ***
CMDX: 92/01/22 23:59:00 01.M10 0 0 0 ,
    IF CMD1 ^= 0 & CMD2RC ^= 0 THEN DO
        "MSG OP ****************************"
        "MSG OP UNABLE TO RECOVER RSCS..."
        END
* RUN SPECIAL SET OF CLOCK COMMANDS ON NEW YEARS DAY.
ENDOFYEAR: 93/01/01 20:00:00 0 0 0 CLKQ EOYCYCLE
EOF
VMUSERID:
/* BUILD THE VM DIRECTORY FROM DIRMAINT SEGMENTS */
'DIRBUILD'
'STATE USER DIRECT A'
IF RC = 0 THEN RUNRC = 0; ELSE RUNRC = RC
```

REXXCALC is a calculator/memory utility with options that:

1. Calculate variables in adding machine or formula modes.
2. Manages workspaces via SAVE, LOAD, DROP, & LIST commands.
3. Passes commands to VM when they are not calculations.
4. Executes REXX code from command line or saved variables.
5. Keyboard assistant via CLKQUEUE's intelligent scheduling.
6. Many powerful options give APL like capabilities to REXX.

Below is a sample session where the user has to:

1) Figure number of cylinders required for a new file.
2) Test how the SUBWORD command works as they are developing a new REXX program.
3) Edit a function named BENEFITS, change some of the formulas, and execute it.

```
REXXCALC   /* X; prompts the user for a response.     */
REXXCALC - RELOADED 9 VARIABLES FROM PROFILE.
REXX IS ACTIVATED INTERACTIVELY...
X; vars
/* INIT VARIABLES FOR REXXCALC EXEC */
$RSCS      = 'CP SMSG RSCS'
$SMART     = 'CP VMC SMART'
$AUTOLOG   = 'CP SMSG AUTOLOG1 AUTOLOG'
$ULOG      = $SMART 'D ULOG'
FMTDATE    = TRANSLATE('34756812',910522'//','12345678')
$OP        = 'CP MSGNOH OP'
UTC_BFRTX = 3746 + 530 + 0 + 0 + 242 + 36 + 1015
BEN_ALLOW = 4112
PER_CHECK = (UTC_BFRTX - BEN_ALLOW)/24
X; $op Peter please mount tape 3003 on 580, Thx mvi.
X; $rscs q sysprtx q
X; ben_allow       /* ask what co paid beny portion is? */
4112
X; rexx say subword('a b c',4)

X; rexx say subword('a b c',2)
B C
X; * next line does calculation within another workspace.
X; rexxcalc ofcspace my_area = (deska+grade6space-isle)
442
X; weekhours = 8.5 + 9.0 + 8.0 + 8.0 + 9.5
43
X; reccnt = 327000
X; blksize = 4096    /* no. of bytes per block */
X; bpc = 180         /* blocks per cylinder */
X; reqcyls = format(((reccnt*132)/blksize/bpc)*2,1,0)
X; save          /* will save prior 5 variable. */
REXXCALC - SAVED 14 VARIABLES IN PROFILE WORKSPACE.
X; xedit benefits calcrexx
X; run(benefits 5050 ben_allow)
X; quit
INTERACTIVE REXX IS CANCELLED BY USER.
```

NODELOAD is a tool for building natural language code that:

1. Compensates for user spelling errors based on context.
2. Maps input vocabulary and loads them into node words.
3. Values that follow keywords are put in its node word.
4. Node words once set are useable by REXX based rules.
5. REXX based rules can be coded as pseudo English.
6. After registration first node word represents call tag.

Below is a sample of the NODELOAD catagories, keywords, and basic vocabulary used in mapping pseudo English grammar. It was inspired by an article written by Richard Brooks titled "A Natural Language Interface to MVS" published in the October 1991 issue of the TECHNICAL SUPPORT JOURNAL.

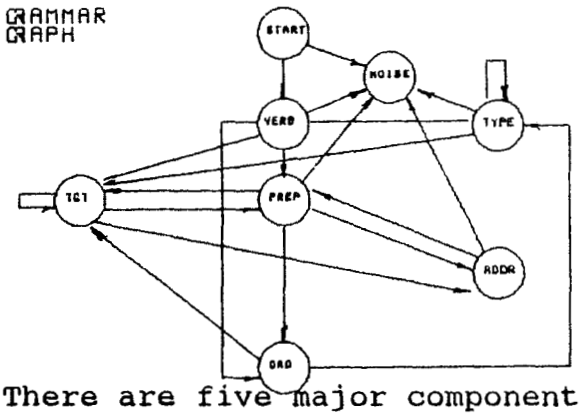
```
INITQUES: /* node type = node names allowed to follow it. */
TYP.START    = CMND NOISE
TYP.CMND     = PREP TYPE ORD TGT
TYP.PREP     = 'CHK_ADDR: ORD TGT ADDR NOISE'
TYP.TYPE     = TYPE PREP TGT NOISE
TYP.ADDR     = PREP
TYP.TGT      = 'CHK_ADDR: PREP ADDR TGT'
CHK.CMND     = SHOWME SHOW LIST GIVE PRINT DISPLAY
CHK.PREP     = AT ON IN TO FROM FOR OF
CHK.TYPE     = TAPE ALLOCATION RECORD UR UNIT ONLINE TSO
CHK.TGT      = DISK CPU TAPE STORAGE MEMORY PATH DRIVE
CHK.NOISE    = ME ADDRESS PLEASE THE A INFORMATION FOR AN
CHK_ADDR:
PARSE VAR RUNSTR $PS $ST
IF VERIFY(WORD($ST,$PS),'0123456789ABCDEF') > 0
   THEN RUNRSPNS = ''
   ELSE RUNRSPNS = 'ADDR'  /* TELL LGC WORD IS AN ADDRESS */
CMND: SELECT
      WHEN FIND('TAPE DISK DRIVE',TGT) > 0 THEN RUN(DODU:)
      WHEN FIND('MEMORY CPU PATH',TGT) > 0 THEN RUN(DODM:)
      OTHERWISE SAY TGT 'NOT RECOGNIZED AS A TARGET.
      END
RUN(ISSUECMD:)
```

```
Sample user input follows with diagnosis options turned on.
PLS SHOW ME THE ALLOC INFO ON DISK 6C1 TO 6C4!
(PLS) miswritten, (PLEASE) set instead.
(ALLOC) abbreviated, (ALLOCATION) set instead.
(INFO) abbreviated, (INFORMATION) set instead.
Attribute (NOISE) automatically set to (PLEASE).
Attribute (CMND) automatically set to (SHOW).
Attribute (TYPE) automatically set to (ALLOCATION).
Attribute (NOISE) automatically set to (INFORMATION).
Attribute (PREP) automatically set to (ON).
Attribute (TGT) automatically set to (DISK).
Attribute (ADDR) automatically set to (6C1).
Attribute (PREP) automatically set to (TO).
Attribute (ADDR) automatically set to (6C4).
FINALCMD = D U,DASD,ALLOCATION,6C1,4
```

NL THEORY FOR NODELOAD

The directed graph SHOW-ME grammar explained by Richard Brooks for developing natural language commands published in the October 1991 issue of the TECHNICAL SUPPORT JOURNAL has been automated.

Below are a list of sentences that the experimental NODELOAD logic can handle and the a sample grammar graph.



Lst al usrs. Dsply tape 001.
Gv usx dxsk 7a1 for 16.
Pls print me the cpu memory,
starting at address e000.

There are five major components in building the "Missouri, Show Me" natural language command parser. When working with rule based PARACODE all of the following steps are automatic, and require no direct coding by the user. The NODELOAD example shows how natural language is done without resorting to rule based code.

1. A Grammar Graph is made to depict how the parts of each sentence will interact. For example, what type of word can begin the sentence. The basic words in the vocabulary are going to be loaded into these words so they should be descriptive. Nouns, verbs, and modifiers are basic parts of speech common in SVO grammars. TYP variables are used to fully represent grammar graphs like the one shown above.
   TYP.attribute = attribute names that can follow
2. A Vocabulary Definition is done using the CHK variable where each attribute get attached to it all the valid words that may be loaded into it.
   CHK.attribute = list of valid words or symbols.
3. Spelling Verification is done first by context then against all attributes. For example, if the unrecognized word follows a verb and then only the attributes valid after verbs are checked for transposed letter and the like.
4. Registration, when a parsed word is successfully found in a valid CHK.attribute list and the word is loaded into its corresponding attribute for later use.
5. Construction, when all the words have been successfully registered into attributes the name of the first attribute registered is used in a RUN() statement. Thus if a VERB like "Show" was the first word in the NL command then the user would code get control via routine called:
   VERB: /* process all registered words via REXX */
6. Execution, when the user has fully constructed the command he must then execute it in such a way that the user may customize or override its use.

PARACODE is a natural language programming system that:

1. Allows users to code ES rules using pseudo English code.
2. Uses multi-word synonyms to give English flexibility.
3. Allows grammatical use of probabilities and fuzzy logic.
4. Allows user to converse logically with knowledge bases.
5. Has frame attributes like ask, why, how, check, & doc.
6. Many powerful options put code wherever users need it.

Below is a sample Expert System (ES) rule written in Paracode.

```
VCRADVSR: /* This rule advises what model VCR to purchase */
If the VCR type is VHS and heads is over three and FX wanted
   then the best purchase is probably a super VCX_1000
   else the best buy is likely to be a dumb Record_Mate99
INITGOAL: The maingoal is best buy and a three is a 3
INITQUES: /* These entries set synonyms and frame values. */
   syn(FX FX:is 'special:effects special_effects')
   syn(VCR CAM 'video:machine video:recorder')
   syn(buy purchase); syn(type model); and syn(heads tracks)
   syn(FX_wanted 'FX:wanted FX:needed') and syn(: super dumb)
   syn(VCR_type 'VCR:type') and syn(best_buy 'best:buy')
The ask.mainexit is 'The VCRADVSR says buy a' best buy'.'
The ask.VCR_type is 'Enter preference: VHS or BETA.'
The chk.VCR_type is VHS BETA  /* check allowed values  */
The dft.VCR_type is VHS       /* default value is VHS  */
The chk.FX_wanted is "CHK_YESNO:"   /* a dynamic call  */
The chk.heads is 2 3 4 5
The fmt.heads is 1 1 numeric  /* one byte numeric only */
The why.heads is "Because better VCR's have 4 or more."
```

Below is a sample dialog with VCRADVSR... (R; is user reply.)

```
Enter preference: VHS or BETA.
R; VHS
Please enter value for (HEADS).
R; which
Your input options are 2 3 4 5.
R; 4
Please enter value for (FX_wanted).
R; yes
BEST_BUY            = CNF(0.80)  VCX_1000
The VCRADVSR says buy a VCX_1000.
R; video machine model
VCR_TYPE = VHS
R; special effects needed
FX_WANTED = 1
R; reset
Enter preference: VHS or BETA.
R; The CAM model is VHS; tracks is 4; and FX needed is not true
BEST_BUY            = CNF(0.70)  Record_Mate99
The VCRADVSR says buy a Record_Mate99.
```

NL THEORY FOR PARACODE

The Subject, Verb, and Object AI paradigm works well with REXX.

REXX it could be expanded to enter NL mode if a switch is set.

```
RULENAME:   /*  basic clauses:  IF antecedent THEN consequent */
[ IF s v o [ conj s v o etc...] THEN] s v o [ conj s v o etc... ]
```

[ ]	Means fields within are optional. Only s v o required.
IF	Valid conditionals are IF and WHEN.
s v o	Subject, verb, object can be represented by a single symbolic such as with true/false values and task() executions, or as separate multi-word phrases beneath.
s	Subject can be up to three distinct words if the Computer Oriented Dialog SYN() synonyms have specified a single root word. They are the equivalent to nouns or noun phrases (NP) in English and basic ATN theory.
v	Verbs can be up to three distinct symbols and/or words if the SYN() synonyms have specified a root word. They imply the action to take against its subject and object, and are referred to as verb phrases in basic ATN theory. Normally verbs are compare symbols, but in advanced NL may be called tasks that have boolean RCs.
o	Objects can be a literal or symbol, or a phrase up to three words if the SYN() synonym's root is defined. It represents a noun and may be preceded or followed by a fuzzy logic or confidence factor (Ie. big/little or probably/definitely). If the object value, after synonym substitution is found to be unknown translation is interrupted, and the object is put into a queue for resolution. Resolution occurs by first looking for a RULE that has a consequent clause that sets the value. Next, framed variables are checked to see if their is a defined procedure to solve the rule. For example, doing a database retrieval. Finally, a previously specified text is used or a text is manufactured that asks the user to supply the value or choose the default.
conj	Conjunctions are limited to and, or, exclusive or, and their character equivalents. They can be three word phrases, but no logical use has been found for using multi-word phrases as conjunctions.

NOTES:

1. Advanced synonym substitution can handle antecedents separately from consequents.
2. Rules can have any number of consequent or antecedent consequent combinations.
3. The RUN() can dynamically invoke any task at will.
4. REXX and its NL equivalent can coexistent in PARACODE.
5. All s and o values, otherwise known as, nouns can be easily updated using NODELOAD directed graph substitution.

PARATALK is a natural way to show properties and relationships.

1. Users can easily encode semantic net diagrams of knowledge.
2. Encoding may consist of pseudo English facts, rules, and acts.
3. Acts invoke scripts, models, and step by step operations.
4. Backward reasoning will try to resolve unmatched phrases.
5. PROLOG deep structure sample: does_well_in(Student,Discipline)
6. Conditionals (ie. =<>) can now be relationals or operationals.

Below is a sample expert system rule written in Paratalk.

```
MAJORADV: /* This rule advises student in selecting a major  */
If a student is interested in a specific discipline,
   and student does well in the subject,
   and the subject is important in discipline,
   and the discipline is in demand,
   then student should major in the discipline
      and forward student transcript to Dean of discipline
   else student should not major in the discipline
INITGOAL: The maingoal is student should major in discipline
/* Examples below show how facts may be initially entered.  */
John does well in math and John is interested in business
Math is important in business and business is in demand
Bill does well in math and Bill is not interested in business
INITQUES: /* These entries set synonym and frame values.     */
   syn(is_interested_in 'is:interested:in')
   syn(does_well_in 'does:well:in') and syn(is_in 'is:in')
   syn(is_important_in 'is:important:in')
   syn(should_major_in 'should:major:in')
   syn(not:should should not) and syn(student 'name:of:student')
   syn(send transfer mail forward)  /* keyword for action logic */
   syn(discipline 'specific:discipline')
The unknowns are 'student discipline subject'
The variables are 'student'  /* if symbol not set than infer it */
The relations are 'is_interested_in does_well_in is_in'
The relations are relations 'is_important_in should_major_in'
The actions are 'send'
The ask.student is "Please enter the student's first name."
The ask.mainexit is "Enter 'reset' to get some fresh advice."
SEND: say "Transcript is being forwarded to" discipline "Dean."
```

Below is a sample dialog with MAJORADV... (R; is user's reply.)

```
Please enter the student's first name.
R; John
student should_major_in discipline = JOHN should_major_in BUSINESS
Transcript is being forwarded to BUSINESS Dean.
Enter 'reset' to get some fresh advice.
R; reset
Please enter the student's first name.
R; Bill
student should_major_in discipline = BILL should_major_in BUSINESS not
Enter 'reset' to get some fresh advice.
R; quit
```

ADVANCED NL THEORY IN PARATALK

In PARACODE a NL syntax was demonstrated that allowed a user to write conventional programming code in subject, verb, and object (SVO) based pseudo English. In NODELOAD a NL syntax was shown that facillitated the building of context free grammars. Using PARACODE and NODELOAD together it was suggested that pseudo English code and dialogs could be achieved as a side affect of Expert System development.

PARATALK takes the expression of NL code and queries to a higher plain by incorporating pattern matching, dynamic verb manipulation and execution, and dynamic variable entry and assignment.

Wherever, SVO phrases are allowed, so too are SRO (subject, relation, object) clauses and ARC (action request commands).

SRO Subject relation object clauses put pseudo English consequents into a pattern table for interrogation as antecedents. "John likes soda" is a typical SRO clause. Unlike PARACODE the central word does not have to get converted to "= > <" symbols. In PARATALK anything goes, just tell the interpreter how to recognize your SRO clauses by loading its relational word or predicate into a variable named RELATIONS. For example, to be able to say "If John likes soda then soda tastes good" you do the following in the INITQUES: section.

```
RELATIONS = 'likes tastes'
```

SRO clauses may be any length. When SROs are in an IF/WHEN (eg. antecedent) statement the associated symbols get looked up in a clause table. If found the IF/WHEN condition is set to true, otherwise it's set to false. If not part of an IF/WHEN condition the clause is put into a table. Matching clauses with previously stored clauses is called pattern matching. In the clause "John likes soda" all the words are taken literally. Now imagine having 10 "likes soda" clauses in the table, but for different people. To refer to all those people the following can be done in PARATALK.

```
RULE: Unknowns = 'Who'; and Relations = 'likes'
If who likes soda then do x is 1 for words(who)
   say word(who,x) 'likes soda'; end
```

In the above example all 10 names of people who like soda would be put into WHO. Sometimes a simple variable, set elsewhere in the logic, needs to be used. In that case enter the word within quotes. For instance, if soda were a variable filled with words like PEPSI or COKE then the PARATALK way to express it would be... "If anyone likes 'soda' then go buy soda".

ADVANCED NL THEORY IN PARATALK

In SRO PARATALK we were shown how to enter pseudo English assertions and interrogations like those that follow:

```
John likes soda
If John likes soda then soda tastes good
If who likes soda then do x is 1 for words(who)
   say word(who,x) 'likes soda'; end
If anyone likes 'soda' then go buy soda.
```

In addition to writing programmable code in English, and interrogating stored English clauses for truth there is another option. You can invoke special AI functions that carry out scripts or models of various scenes, events, speech, manual operations, and/or machine components. For instance, the consequent clause "go buy soda" is an imperative statement that requires a direct action.

ARC Action request command clauses have two parts. The first is the action part which corresponds to the program name used during CALLs from normal procedural code. The second is the request part which corresponds to the variables passed during normal procedural calls. However, for scene or model invokations to occur using pseudo English statements something must tell the PARATALK interpreter that this is an ARC phrase, rather than an SVO or SRO one. That something way is to load the primary action word (ie. verb) into a variable named ACTIONS. Since actions speak louder than words below is a sample of what I'm talking about, full blown PARATALK.

```
ACTIONS are 'go walk get fasten drive'
RELATIONS are 'likes';  and soda is PEPSI
GOBUYSODA: If anyone likes 'soda' then go buy soda
additional backward or forward chaining rules...
/* Basic script follows for going to the store */
GO: Parse var runstring whattodo withwhat .
If whattodo is 'buy' then do
     Item is withwhat  /* comments are allowed too */
     Walk to car; get in car; and fasten seat belt
     Drive to store and exit from car
     Walk into store and purchase store 'item'
     Drive back home
     end
/* The actions below can be external programs too.  */
WALK:  etc...
EXIT:  etc...
DRIVE: etc...
FASTEN: etc...
```

Basically, the above example neatly mixes all three NL methods: SVO, SRO, and ARC. It's pretty natural, wouldn't you say?

ADVANCED NL THEORY IN PARATALK

In SVO, SRO, and ARC we were shown how well PARATALK armed the Knowledge Engineer (KE) with the tools needed for building Conventional and Expert Systems using Pseudo English. Command clauses and phrases could be easily constructed that were declarative, interrogative, and imperative without requireing the KE to resort to arcane coding artifices. And there is much more...

External file data can handled be handled dynamicly using the LITERALIZER concept peculiar to the data driven pattern matching protocols of OPS5. With it files, sensors, and knowledge bases can be processed with 5th generation granularity using something resembling the well known object oriented paradigm. For instance, the clause "If cat weight is high and finickiness is extreme then type is Cheshire" is valid PARATALK terminology using LITERALIZERS.

OAV Object attribute value conditions can be employed in conjuction with SVO clause rules to provide name tags to fields in records. An example below builds a literizer for the CAT clause shown above. The basic format for entering a literalizer follows...
OPS(filename,objectname,attribute1 attribute2 etc...)

Actual sample...
OPS('FELINE DAT',CAT,'WEIGHT FINICKINESS TYPE')

Note, full power of the parse command is available.

OPS('RACF DATA A',PROFILE,'24 PW 32 1 'DATE=' DATE)

Also, some basic assumptions are now possible pertaining to context. For example, that CAT is the object for the attributes weight, finickiness, or type is easily implied. What's more cases of ambiguity (more than one literalized "object" contains the same attribute name) are easily resolved to most KE satisfaction by understanding that ununique attributes will get the object from one most recently used. Next, pronoun usages like "it" or "they" are possible and can be substituted, easily again with the most recently used object with a matching context definition (TYP.) An example of that kind of clause is shown beneath.

CAT_TYPE: If its weight is high and finickiness is extreme then its type is a Cheshire

In the above rule the value of "its" will be taken from whatever the last object happened to be that contained the attribute "weight".

PERFORMANCE ENGINEERING/MANAGEMENT OF A LARGE REXX APPLICATION

PAT MEEHAN AND PAUL HEANEY
IBM

Performance Engineering/Management of a Large REXX application.

Pat Meehan, Paul Heaney

ECFORMS[1] Development Team
IBM HSL Programming Systems Laboratory and Delphi Software Limited
Dublin, Ireland

Abstract

This paper addresses the performance engineering/management of a large REXX product (100 Kloc). This was in response to a market driven requirement to improve product responsiveness. The Software Performance Engineering methodology in conjunction with our own developed processes were used throughout. A number of Software Performance Engineering processes were adopted and became the basis for the drive towards performance improvement. A benchmark was developed with customer input. Targets for representative transactions were defined for our three main metrics, end-user response time, virtual CPU time and start I/O. Various REXX performance ideas/myths were validated/rejected on the basis of measurements. Measurement and analysis tools written in REXX were used to automate and assist in the continuous tracking of the targeted performance improvements. Bottle-necks in the code were identified by tracing and measuring some/all of each target transaction. The main performance engineering principles that were used were Fixing Point, Parallel Processing, Centering, Processing versus Frequency and Instrumenting. Follow up meetings with the IBM VM Laboratory's performance team in Endicott, discussions with Mike Cowlishaw and consultation with other REXX development sites has resulted in a pool of knowledge being built up on Performance methodologies and REXX performance rules/guidelines. Education of the development team resulted in REXX performance rules becoming instilled in the day to day design/coding of product changes. The product has a separate performance development team and eighteen months down the line focusing on performance has resulted in major improvements. Throughput in the two key service machines has been doubled and end-user response time has been reduced by a quarter. Further improvements have been prototyped which indicate we will be able to improve the throughput again on the service machines and significantly reduce the end-user response time.

Author Information

Pat Meehan B.E., M.Eng.Sc. joined IBM Ireland in 1984. He spent two years on assignment in IBM Netherlands where he worked on the performance of ECVM (EMEA Common VM) REXX applications. He then spent two years working on the design and implementation of a data extractor generator on MVS. Subsequent work included working on the future strategy of a program product, SAA/DM. In the last eighteen months, he has been responsible for the performance work being carried out on the REXX product offering, ECFORMS together with working as a consultant on performance to the rest of the development group.

Paul Heaney works with Delphi Software and is a REXX development consultant to IBM HSL Laboratory. He has been closely involved with the ECFORMS Product Offering over the past two years. He has concentrated his effort on the application of performance engineering techniques to the application over the past eighteen months . As a key member of the performance team, he also acts as a consultant to the rest of the

[1] ECFORMS is an electronic forms management system which is an IBM product offering and is a trademark of IBM.

development team. He has developed tools in REXX to assist in the measurement and analysis of the development effort.

Performance Engineering/Management of a Large REXX Application

Electronic Forms Management System (ECFORMS) is an IBM licenced program (5785-MCB) extensively used within IBM by many diverse applications and marketed in the US, Europe and Japan. It has three main functions, Forms Processing, Forms Design and Forms Administration running on the VM operating system.

The major features of ECFORMS Forms Processing are :

1. Filling in online forms (Origination).
2. Routing forms electronically.
3. Using electronic signature to :
   - Approve and Final Approve a form.
   - Reject a form.
   - Cancel a form.

Some of the main ECFORMS transactions (e.g. Filling in a form) involves communications via IUCV with two service machines in a serial fashion. There would be data validation by a data service machine followed by control checking and routing by an authorization service machine. In a multi-node situation, these service machines communicate with their peers through the use of spool files.

All of the three major functional areas are written, almost exclusively, in the REXX language.

This paper focuses on the performance engineering effort of the ECFORMS Forms Processing function.

Background

The market driven requirement to improve performance was identified by direct communication with our customers. The main performance issues were:

1. Response times from an end-user's point of view.
2. Throughput on the two key service machines.
3. CPU consumption and excessive I/O demands

These issues became the driving force behind the performance effort.

Approach

Most performance methodologies advocate (correctly) the application of performance engineering to systems in their early developmental stages. They concentrate largely on new systems and not on existing systems.

An example of such a methodology is that of Software Performance Engineering [1] (SPE). Closer examination of the methods encapsulated in this methodology highlighted a considerable degree of applicability to existing systems also. For this reason, SPE together with our own methods formed the basis of our approach to the performance effort.

Effort Allocation

The resource to improve the performance of the product has been spread over two different efforts. The initial effort (Effort 1) used three different development locations. Our own laboratory was one of these locations and also acted as the focal point for the other 2. The subsequent effort (Effort 2), currently ongoing, was concentrated in our own development laboratory and involved the setting up of a performance team for the group.

This paper focuses to a large extent on the performance work carried out in our own laboratory over the two efforts, although it draws on significant experiences from working with the other two locations.

SPE Methods

SPE is a methodology which advocates the application of performance analysis in the development of software systems. It provides a sensible method for the production of software that will meet certain

performance objectives. SPE encompasses the following methods [2]:

1. Design Principles which are an abstraction of the expert knowledge of performance specialists
2. Data collection is the means of acquiring the data necessary to describe the performance specifications
3. Modeling techniques uses execution graphing and analysis algorithms to predict performance
4. Proposal Evaluation
5. Instrumentation of the software system.

Several examples can be found in the literature of successful SPE usage [3]

SPE in its entirety was not appropriate for an existing product; we used those aspects of the SPE methodology which we found suitable for the task.

Adopted Benchmark

SPE recommends the use of a benchmark as an experiment for collecting performance data. The benchmark should be a good representation of the way the software is used by customers and should be easily reproducible.

The first step was to develop a benchmark to gather pertinent performance metrics. Compromises between representativeness and reproducibility had to be made due to resource and time constraints.

The main feature of the benchmark [4] were as follows:

- A set of transactions that were representative of the typical user workload scenarios based on a considerable depth of knowledge within the laboratory's support and development groups together with feedback from a subset of customers. A typical transaction would be to approve an ECFORMS form sent by an originator or for a service machine to process that approval request.
- A representative ECFORMS form
- Software operating conditions like compiled service machines at given priorities running on selected hardware. Due to resource constraints, the initial hardware was a 4381 running VM/SP5 but this has since been extended to include VM/XA on a 3083 and VM/ESA on a 3090.
- Single user on a single processor. This has been extended recently to include two processors communicating with each other for a subset of transactions.

The benchmark is an evolving experiment undergoing change as the software undergoes modifications and the user workloads shift.

Application of the SPE principles

The SPE Design Principles are a formalization of the performance knowledge of experienced performance engineers.

The principles of SPE were intended primarily for software creation, but we have found some of them to be equally applicable to a project which has undergone significant development work. However, it is conceded that the application of the principles is a more painful exercise at the later stages of a product's evolution.

The design principles have since formed the basis of our performance guidelines which we provide to our own and other development groups in the lab.

We now describe those adopted principles that were particularly applicable to the existing system and examples of that applicability.

1. **The Fixing-Point principle states that the connection between the data and the required result should be established as early as possible in the processing provided that the cost of retaining that connection can be justified.**

   - The product uses flat files as its file system. We found several cases where the product was accessing the same control files, several times in the same transaction sometimes for the same information.

     According to the principle, it made more performance sense to read selected files or sections of files into storage at initialization. Data was stored in a REXX array of the form x.y, where y was the key. Subsequent retrieval of information was then done from storage with great efficiency.

     Application of this principle was more appropriate to the service machines where initialization time was not a concern.

However, one of the constraints was that both service machines should still be able to run with 3meg of memory.

For some large files, the cost of retaining the connection for the entire file could not be justified because of the storage constraints.

The largest file used by the service machines is the organizational directory typically of the order of thousands of records.

Client information is retrieved by one of the service machines from this directory for auditing purposes. Here, we have prototyped the concept of a time window, where participating client information is extracted once from the directory within the time window and held in storage. Subsequent attempts to retrieve the same client's information, during the time window, is then done from storage.

In this way we can allow the installation control the cost of holding the information in storage by changing the time window.

2. **The Processing versus Frequency Tradeoff principle, says that we should consider the processing time of a transaction and the number of times that transaction is executed and minimize the product**

   - This principle can be satisfied in a number of ways. One of the less obvious ways is to expand the processing within a particular transaction to include another transaction ensuring that the new transaction is faster than the sum of the two old transactions.

     This can often be achieved by making maximum use of the overhead involved.

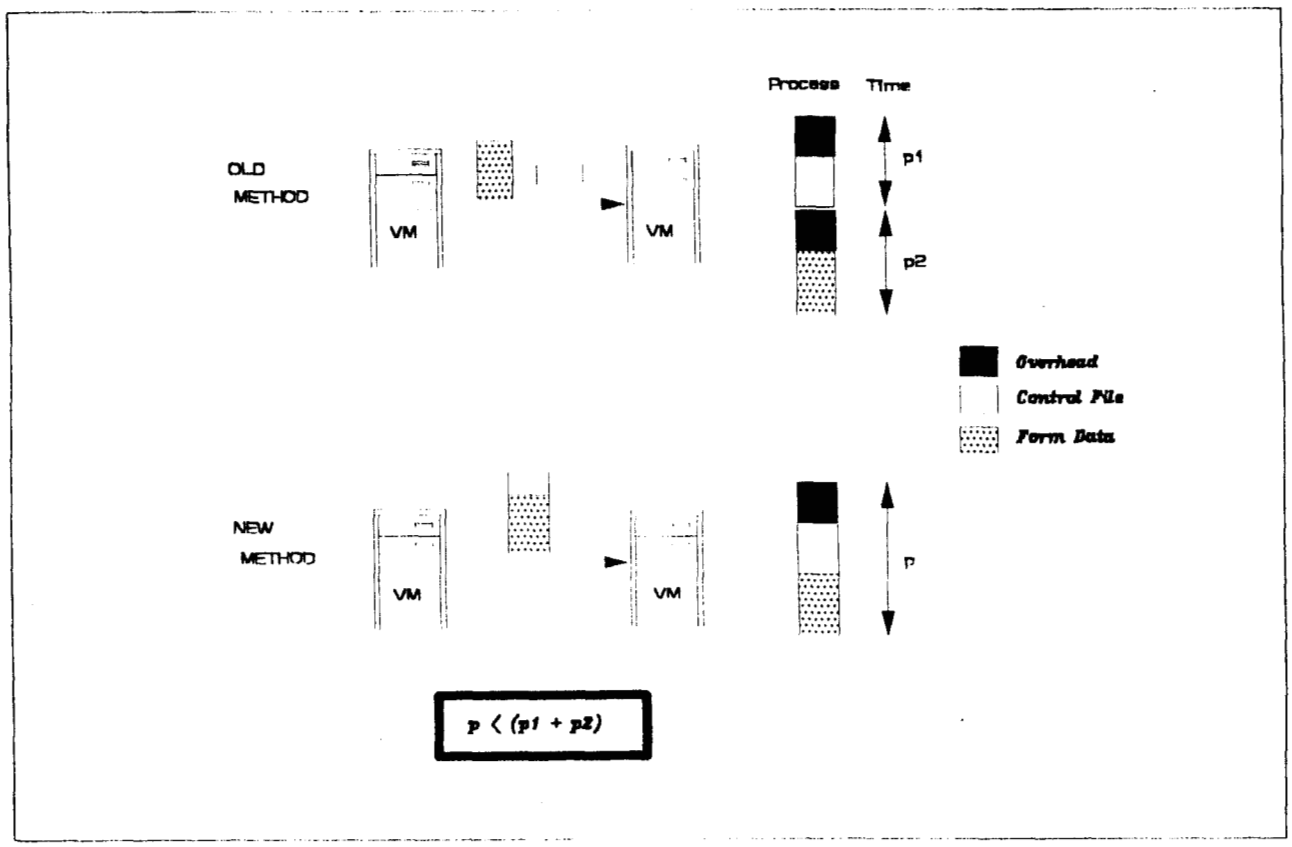


Figure 1. Processing-v Frequency Tradeoff Principle

The diagram (please refer to Figure 1) shows a typical application of this principle. Currently, when data is transmitted to participating nodes, a control file is sent and is followed by the form data file. The form data file has header information which associates it logically with the control file.

This means that the receiving node has the duplicate overhead of processing two spool files for each form data file and of estab-

lishing a logical association between the files received.

A change has been prototyped to make maximum use of this overhead, where the form data file is chained directly to the control file and a single file transmitted. The receiving server has only to process the one spool file thus ensuring that the product of the processing time multiplied by the frequency is reduced significantly.

Changes of this nature present significant architectural and migrational difficulties and underlines some of the headaches of performance engineering of developed systems.

3. **The Centering principle advocates the identification of the dominant workload functions and the minimization of their processing**

   - The transactions of form origination, approval and final approval were clearly the dominant workload functions and a large proportion of the effort was focused on these transactions.
   - A further application of the centering principle is of course within each of the dominant transactions. This performance refinement identified the dominant processes within the dominant transactions.

     Validation of form data is a typical example of a dominant sub-process. The form data is defined as a set of fields and associated values. Validation of form data occurs for all three dominant workload functions.

     Previously, during approval and final approval of each form instance, this validation process was again applied to the entire form data. A significant processing reduction was achieved on two dominant transactions by applying the validation process to the changed form data only.
   - As an indirect extension to the last point, it was realised from customer contact that form data approval and final approval often occurred without any changes to the data. When this occurred and was detected a large proportion of redundant processing was bypassed.

     Even more substantial improvements have been prototyped for this change in the multi-node scenario. Currently, the data is transmitted to participating nodes even if it is unchanged and the receiving nodes have to go through a lot of unnecessary processing to handle the large amount of spool files.

     A change has been prototyped where the form data is only transmitted when it is changed resulting in significant savings to the handling of remote requests by the service machine as illustrated by the diagram, Figure 2 on page ix.

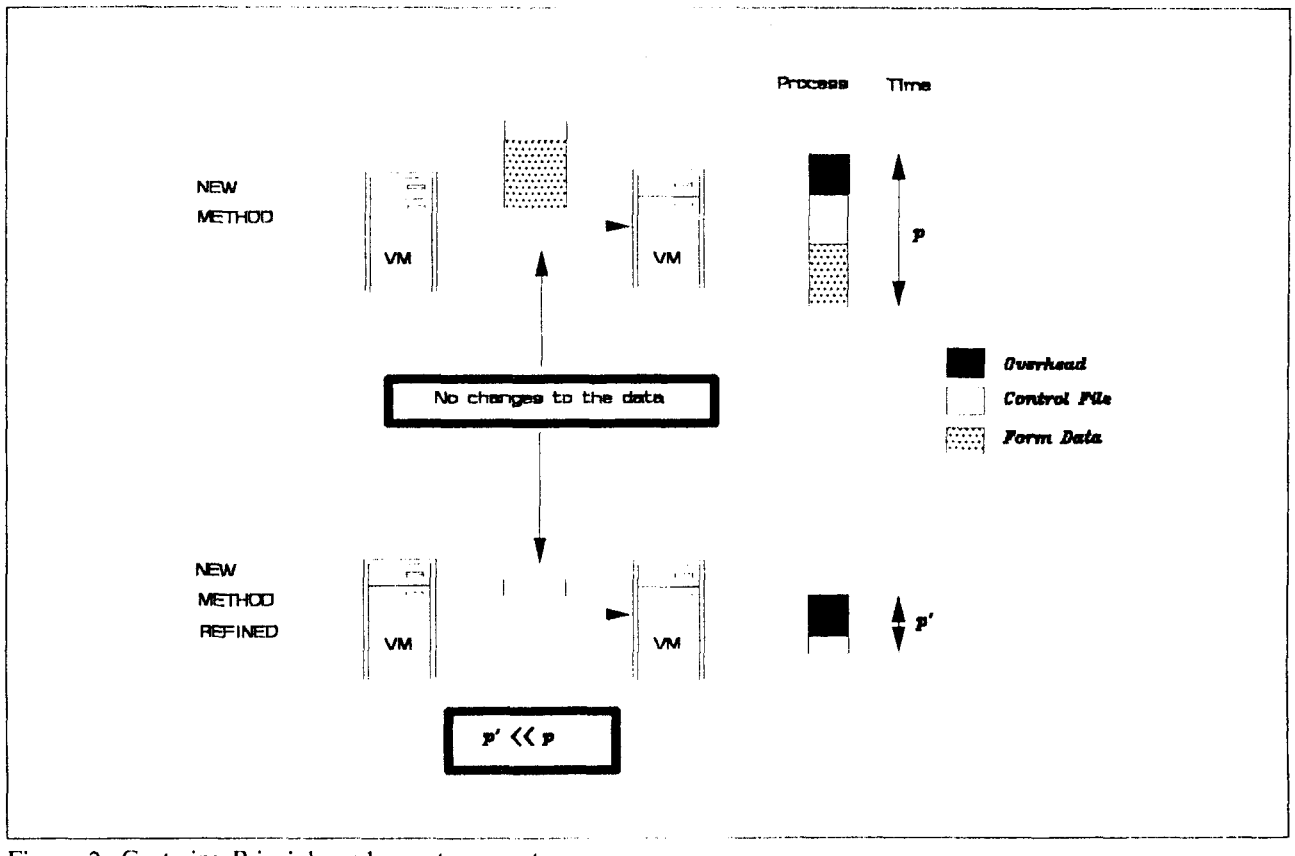


Figure 2. Centering Principle and remote requests.

- A further application of the centering principle within each transaction is the partitioning of the processes within a transaction into normal and exception partitions. Partitioning identifies the normal paths through the process and the unusual paths (exception partition) and focuses on the former from a performance point of view.

4. **The Parallel Processing principle states that processing should be partitioned into real or apparent concurrent processes provided that the benefit outweighs the communications and resource contention overheads.**

   - Some of the dominant transactions require the client to communicate via IUCV with a data server followed by an authorization server in a serial fashion. The authorization server is architected to a large extent on the basis that the data server has completed its processing successfully.

     To involve the data server in some sort of parallel processing with the client was not considered feasible because of the major architectural difficulties.

     However, in the case of the client-to-authorization server, a change has been prototyped where control is handed back to the client at a much earlier stage after some preliminary processing for appropriate transactions (Approval and Final Approval) as shown in Figure 3 on page x.

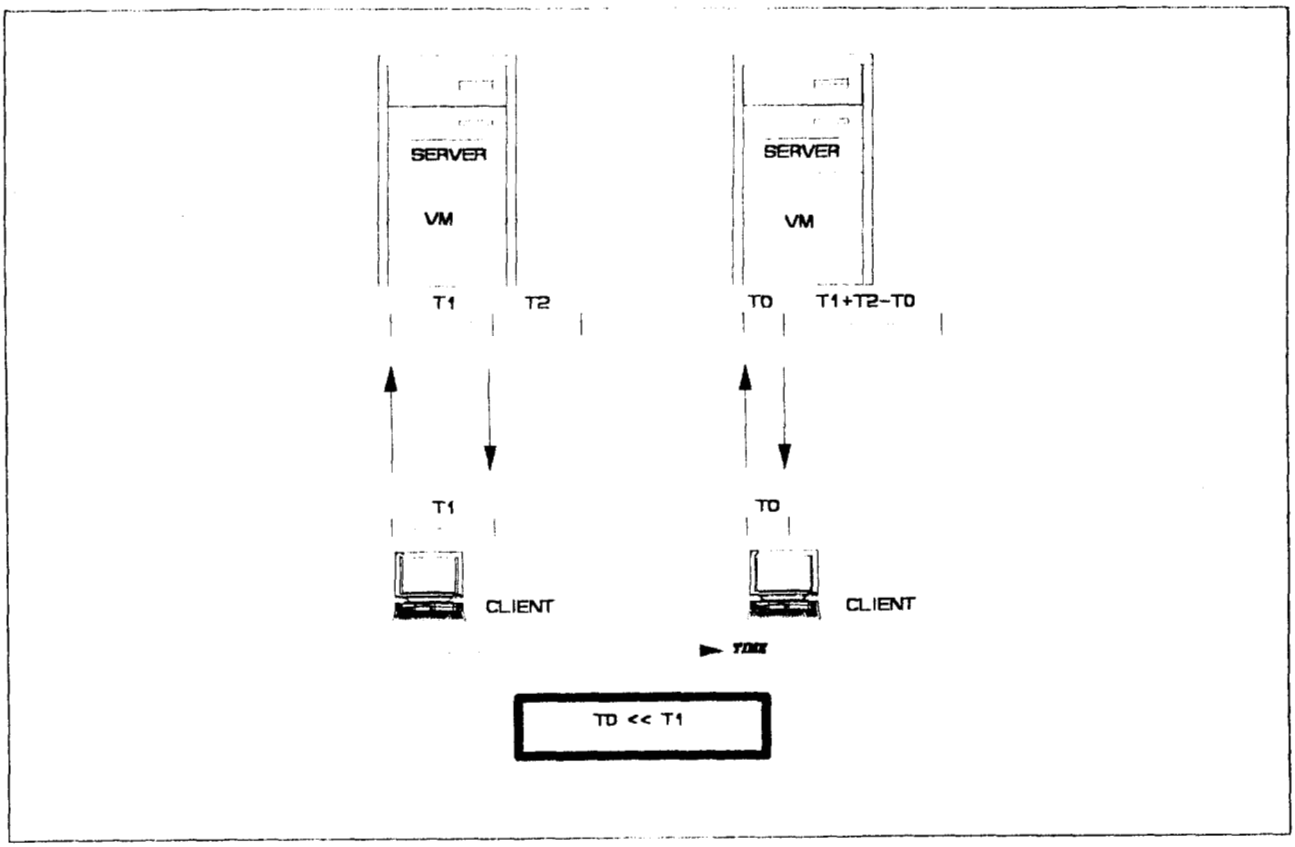


Figure 3. Parallel Processing Principle and Asynchronous request handling

The server continues processing asynchronously to the client. The ultimate outcome of the transaction is recorded, as usual, in a client log file.

From a client point of view, this reduces the transaction time very significantly (T1-T0) with little architectural impact on the server. Naturally, the improvement is maximised where there is no queue to the server and is diminished according to the number of requests in the queue.

5. **The Instrumenting principle encourages the instrumentation of the system as the means of measuring and controlling performance.**

   This is a control principle which does not directly improve software performance

   This principle was originally not part of the SPE methodology but was subsequently included because of its essential role in the performance effort

   Further discussion of this principle and the entire measurement process is continued later

Other important lessons were learned which were phased into the approach at different stages across the entire effort.

Measurements

For existing systems, measurements are the key to success. They provided us with an execution knowledge of the system enabling us to model the system. This model in turn allowed us to decide on the types of change required. Measurements were also the key to understanding the success or failure of the changes.

Experience and results have illustrated a number of important lessons:

1. Firstly, the importance of a proper measurement system is apparently not obvious to most people. Often developers express a sense of incredulity when asked if the system they had developed had been measured [5].
2. Where a satisfactory measurement process is not part of the approach, success is very dependent on intuition and luck and this is not a very scientific way to proceed.

3. The measurement process should use as many performance indicators as is practical to verify a performance prediction. A single indicator of performance (like response time) can be very misleading.
4. Early measurements of less than complete efforts are imperative. Even though these will often be contested on the grounds that further performance *tuning* will follow, they provide an early warning system which is often well-founded.

The measurement process itself is described later.

Language

Most of the product was written in REXX, the remainder in C. An important guiding principle which we adopted (and not just for language considerations) was that of benefit/cost maximization.

It's important to maximize the benefit/cost ratio where the benefit is the estimated improvement in performance to the customer, measured by the benchmark and the cost is the resource needed to develop and maintain that change.

In general, the CP and CMS commands and other external modules and not the REXX instructions were responsible for the substantial part of the product. Sample transactions show that the REXX instructions account for less than 20% of the total. In addition, poor performance was not caused by poor REXX coding but by lack of performance sensitivity in the original design stages with some notable exceptions. This was also borne out by informal discussions in the area of REXX performance with Mike Cowlishaw.

The main exceptions were the use of keyed arrays, which is a very slick way of searching for data, rather than the traditional binary search technique and the removal of Interpret statements to make the code compilable.

Conversion of the C code to REXX made good performance sense because of the way that the two languages interact differently with CP and CMS and some of this task has already been accomplished in Effort 1. This change from C to REXX has the additional benefit of easier maintainability.

Other improvements within REXX were better management of storage and in particular the dropping of storage when appropriate together with the complete specification of CP and CMS commands without ambiguity.

Changing REXX variable and procedure names, positioning of routines within the program, the use of one particular REXX built-in function over another were found to be examples of high cost - low benefit changes.

Exploratory Prototyping

We seized the opportunity to deviate from the standard practice of a documented low level design by prototyping the designed performance changes. This approach allowed us to measure progress at a stage much earlier than would have been possible with the traditional phased approach.

It also appears, based on a causal analysis of software defects found so far, that the prototyping made a very positive impact on the quality of the software shipped.

We strongly advocate prototyping as the best way to manage performance engineering of an existing system.

Other Items

For the first effort and because of resource constraints, measurements were confined to a 4381 processor running VM/SP5. This was restrictive and not very representative. The second effort has extended the measurement process to a 3083 running VM/XA and a 3090 running VM/ESA.

A comprehensive report [4] was created of the results of the first effort. This formed the basis of discussions which were held with the Performance team in the IBM VM Laboratory in Endicott, who reacted very positively to the depth of analysis and overall approach. The report has also been sent to all internal product sites to inform and encourage them to upgrade to the latest release.

The benchmark continues to be based on a single user and on a single processor for most of its transactions. The authors believe that some modeling of multiple users is a key area for the future which will help particularly in the area of capacity planning for our customers.

Functional development work of the product has continued alongside the performance effort. Both

teams have worked closely together with the performance team acting in an advisory role on the functional enhancements from a performance point of view. This close collaboration has been critical in the management of the performance work and has resulted in much greater sensitivity to performance within the entire group.

Measurement Process

The measurement process and its findings were both the guiding force behind the performance analysis along with being the ultimate arbiter on the success of the effort.

The adoption of the instrumenting principle at the very early stages of analysis, enabled us to isolate the major areas for each of the SPE design principles. The instrumenting principle is of greater significance for products that have been developed so far without the use of SPE and in a sense replaces the type of modeling, advocated within SPE for new products.

For this reason, the modeling effort is reduced and the measurement effort increased.

In this way, SPE is still very applicable to existing products but with shifts in emphasis when compared to new products.

1. **Performance Refinement**

   The initial stages of the measurement process involved a breakdown of the dominant transactions into their sub-components. These sub-components then became the subject of analysis through a limited set of unsophisticated measurements of virtual CPU time, Start I/O and response time. This provided us with important initial execution data of the system.

   In this way, the refinement provided us with a type of informal software execution graph of each of the transactions, a form of analysis advised under SPE.

   Of course, the REXX language with its rich tracing functions and its end-user friendliness, lends itself very well to this type of approach.

   This refinement pointed to those areas that should be concentrated on , which together with the other guiding principles (SPE and Benefit/Cost) already referred to, became the basis of the performance design changes.

2. **Prototyping**

   The main features of the proposed design changes were prototyped at a very high level and a new set of measurements obtained. These measurements formed the basis of the target objectives for each transaction within the benchmark which together with the design changes constituted the initial design document. This was subsequently approved by a selected list of external and internal reviewers.

   This allowed our customers an early indication of the magnitude of the performance improvements that could be anticipated and an incentive to agree to the resource investment.

   In this fashion, the refinement and limited prototyping provided us to a large extent with the necessary data to define the performance specification. The same type of data collection is also advocated under the SPE methodology, although the manner of collection is naturally different for new software systems.

   As part of the second design stage (referred to in the IBM phased approach as low level design) the prototyping exercise was continued at a lower level with the prototype being more closely aligned to the ultimate implementation.

   The prototyping exercise was really a prerequisite to the measurement process and they complemented each other very successfully.

   In a few instances predictions were made for some of the performance metrics based solely on the software execution graphs of the transactions. These predictions were then compared with actual results from the prototyping exercise and were used as a theoretical validation of the prototyping results.

   Basic measurements were periodically taken during the prototyping and modifications made where there were any deviations from the objectives. This design stage became a highly iterative process and emphasised the engineering approach to the whole problem.

3. **Data Collection**

   Even though the instrumentation was an integral part of the entire performance development cycle, it wasn't until the changes had been

designed and implemented that the more controlled measurement experiments were conducted using acquired customer data.

The entire measurement process is a complex one where there are so many contributing factors. We adopted a number of approaches to make it as realistic as possible within the working constraints. We concede that further enhancements are both desirable and necessary.

- Probes were inserted at appropriate parts of the end-user and service machines to track the metrics which included Virtual CPU time, Start I/O , Response time, free Virtual Storage and System Load. These probes were positioned to capture the metrics for

  a. The Total End-User Component which includes the waiting for a service machine to respond (a)

  b. The Total Service machine Component for both servers (b)

  c. The Interface part of the Total End-User Component (c, c < a, a-c < b)

- The system was triggered once certain initial conditions had been set up. These conditions were based on varied customer input. They included directory size, number of forms in progress, sizes of critical control files which were typical of a customer installation.

- Measurements were always conducted on both the new and old implementations in a number of ways:

  a. An old and new installation was set up on the same CPU and both were triggered simultaneously for a given set of benchmark measurements

  b. On other occasions, measurements of the old and new implementations were interwoven in the following manner - (old, new, old, new, old, new)

  c. All controlled measurements were run at off-peak times

4. **Interpretation and Evaluation**

Existing tools were used and new ones developed to enhance the measurement process.

- KEYPLAY is an IBM internal use tool which runs on OS/2 and executes a set of pre-defined keystrokes on a host machine. KEYPLAY has been used to execute the defined benchmark usually at a deferred point in time (off-peak), switch between different systems (old and new) dynamically, collect the results and invoke the other developed tools to analyse the results.

  KEYPLAY has been instrumental in providing a fully automated measurement process where it can be triggered during off-peak working hours and the following morning a summary of the results taken at off-peak is available on the disk of the requestor. The interpretation of and judgements about these results is still an important and necessary follow-up step.

- We have also developed extensive REXX tools to analyse the collected results.

- The results over a number of runs of the benchmark are treated as follows

  - The lowest and highest 10% of the runs are ignored leaving the middle 80% for interpretation in order to weed out extreme results.

  - This remainder is averaged and a comparison made between the old and new implementation.

  - Occasionally, we measure a control which is identical within both the old and the new implementations and normalize the results with respect to this control. Both the normalized and the unnormalized results are then interpreted and compared.

- Interpretation of measurement data is something which improves with performance analysis experience, familiarity with the actual task of data interpretation and knowledge of the software under investigation. The key is to treat results with caution and respect and the goal is to try to get reasonable consistency in your results.

- An important point to look out for is perturbation of the results by the measurement system itself. This is best checked by comparing the results of the probed system with the system without any probes.

- Management of the vast amounts of measurement data is important. We used a summary file to reference the data

belonging to a particular run of measurements which held key information about that run.

Results

The targets prototyped for the the original performance effort (Effort 1) were based on the three metrics of elapsed time, virtual CPU time and start I/O. These metrics were used for all the benchmark transactions throughout both performance efforts as a means of gauging our success.

We have represented a summary of the results in the following diagram (please refer to Figure 4 on page xvi and Figure 5 on page xvi) for both the end user and the service machines as follows:

1. Prototyped target results for Effort 1.
2. Actual achieved results for Effort 1.
3. Prototyped results for Effort 2 + actual achieved results for Effort 1.

A more detailed account of the actual results from Effort 1 is contained in a separate report [4].

The results are presented as a % reduction on the base at the start of Effort 1.

Our approach has led us to target a subset of the benchmark transactions. However the charts show the percentage reduction, in each metric, over the entire benchmark and not just over the targeted transactions. The reductions in the targeted transactions had to be higher to achieve this overall result. For example, the targeted transactions on the service machines had to achieve a 34% reduction in order to achieve the 31% overall reduction.

The main features of the results are that

1. End-User Response time over the entire benchmark has been reduced by 24 % and further prototyping indicates that we can achieve an overall reduction of nearly 50%.
2. We have achieved over 50% reduction for the service machine transactions and early prototyping has indicated that this reduction could be close to 70% when we have concluded the current effort (EFFORT2).

Conclusion

The main conclusions of the approach are :

1. A number of SPE methods can be applied, with significant success, to existing software products. This is particularly true of a REXX product which lends itself to in-depth analysis.
2. Prototyping is very necessary in predicting performance results. Prototyping also had a significant impact on the quality of the software shipped.
3. We can not over emphasize the importance of measuring results from an early stage in the development cycle. Constant re-measuring of results ensures that performance degradation is not allowed to creep into the project at any stage. REXX myths which had been presented to us as ways to improve performance, eg. Code tuning, were discarded by the measurement approach.
4. The key to finding what works on your product is through study of the SPE methodologies, analysis of the areas of your product where they can be applied and then measurement of the results that can be achieved to determine their cost effectiveness.

Apart from the significant performance improvement, the drive for improved product performance has also produced the following :

1. Performance culture established in the development group. It is important to recognise that alongside the performance work, functional development of the product has continued often in similar areas. It is testimony to the change in performance culture within the entire group, that this has been a relatively smooth collaboration.
2. Initial feedback from customers has shown a marked improvement in satisfaction with the performance of the product.
3. Performance guidelines established for the Laboratory.

4. Performance engineering, as part of the development cycle, highlighted within the group and the Laboratory.

In retrospect, the key to success has been in the overall approach. The use of SPE principles as a guiding force, the adoption of an exploratory prototyping approach together with a significant investment in the measurement process have been the critical success factors. The engineering concepts of design, measurement and assessment in an iterative fashion, have been the kernel of the entire approach.

In conclusion, we have proven that the use of SPE methodologies together with our own methods to improve the performance of an existing REXX product were both worthwhile and practical.

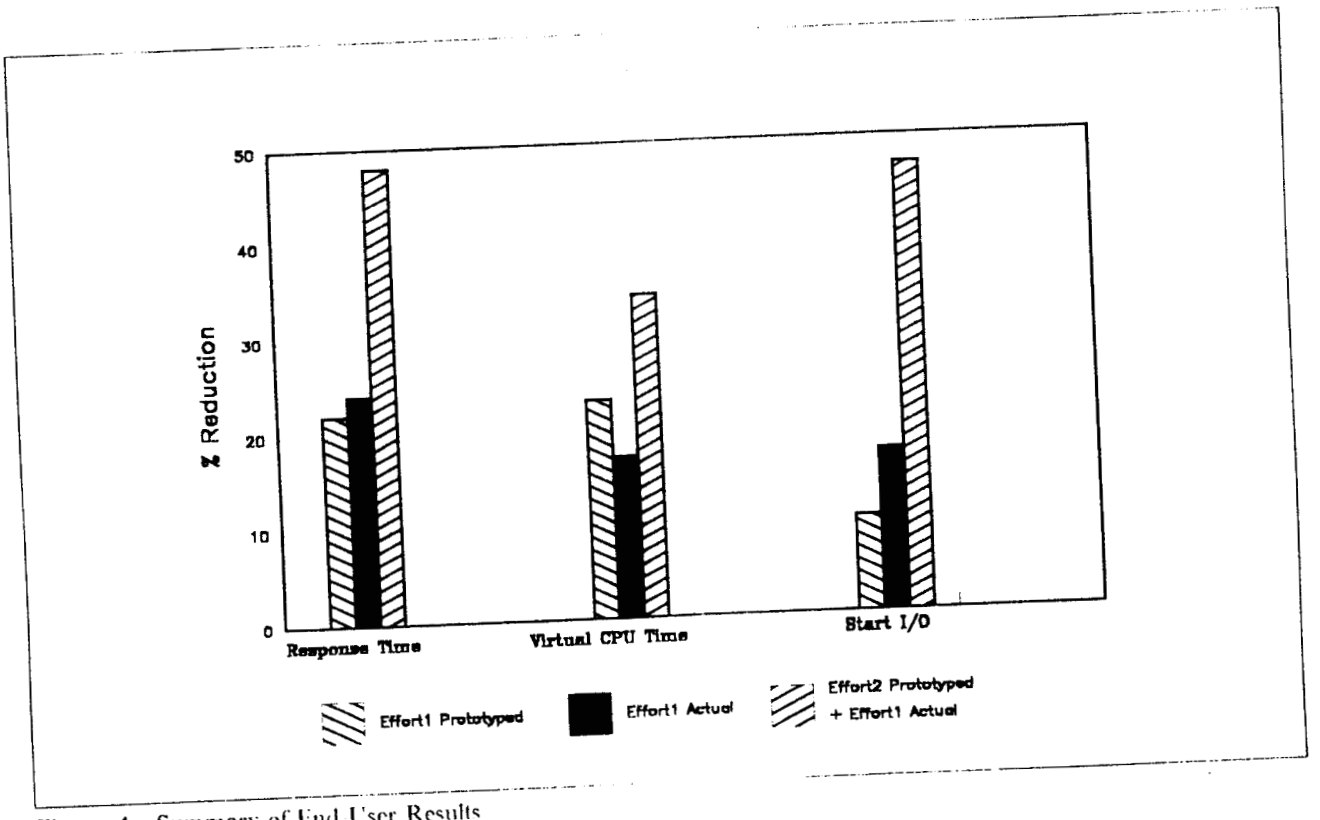


Figure 4. Summary of End User Results

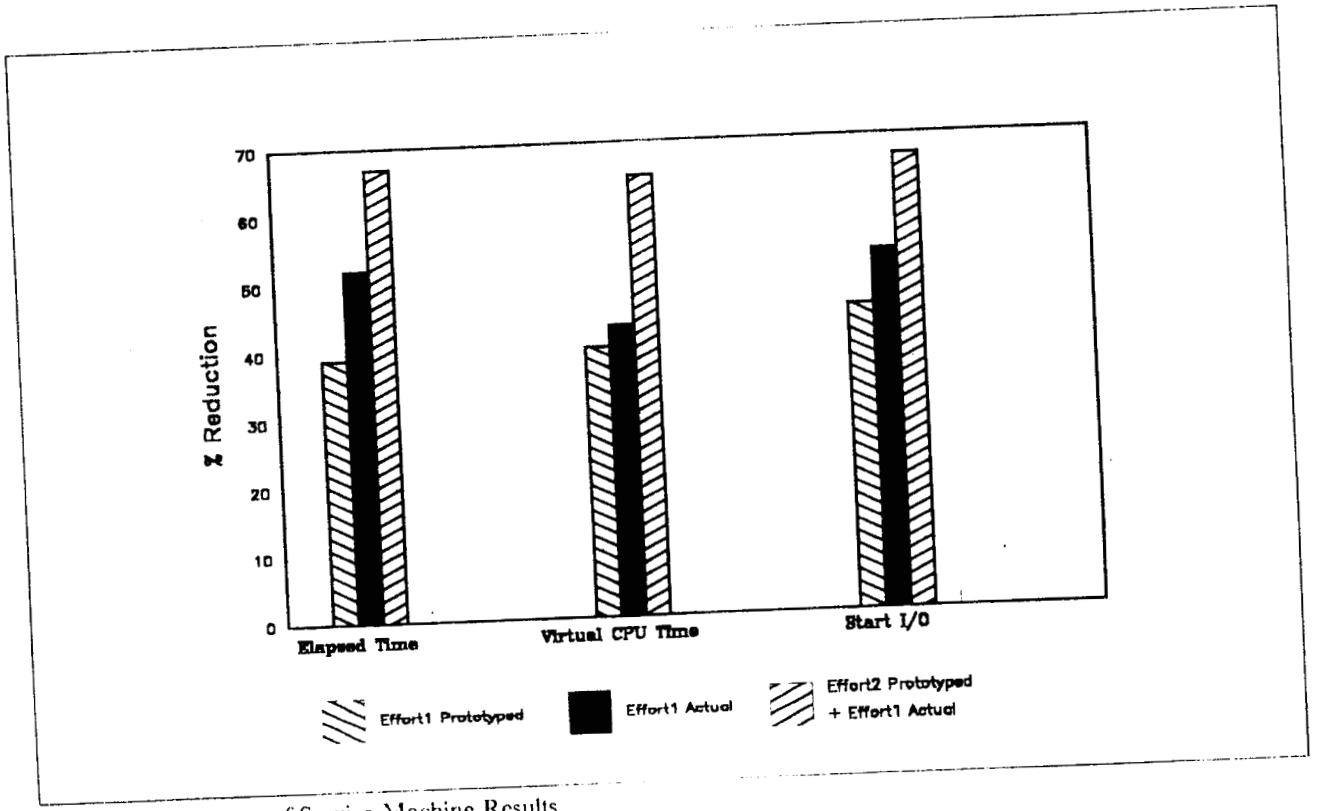


Figure 5. Summary of Service Machine Results

References

[1]Connie U. Smith, "Software Performance Engineering", Proc. Computer Measurement Group Conference XII, Dec 1981, 5-14

[2]Connie U.Smith, "Performance Engineering of Software Systems", Addison-Wesley, 1990

[3]Connie U. Smith, "Who uses SPE?", CMG Trans., Spring 1988, 69-75

[4]P. Meehan, IBM Internal Use document, ECFORMS Release 3.0 SPE 1 Performance Report, June 1991

[5]T.E. Bell and A.M. Falk, "Performance Engineering: Some Lessons from the Trenches", Proc. CMG 87, Orlando, Florida, Dec 1987

Other References

Gordon E. Anderson, "The Coordinated Use of Five Performance Evaluation Methodologies", CACM, 27,2 Feb 1984, 119-125

C.T.Alexander, "Performance Engineering: Various techniques and Tools," Proc. CMG Dec 1986, 264-267

P.J.Jalics, "Improving Performance the Easy Way", Datamation, 23,4, Apr 1977, 135-148

Connie U. Smith, 'General Principles for Performance Oriented-Design', Proceedings CMG 87, Orlando, Florida, December 1987, pp 138-144

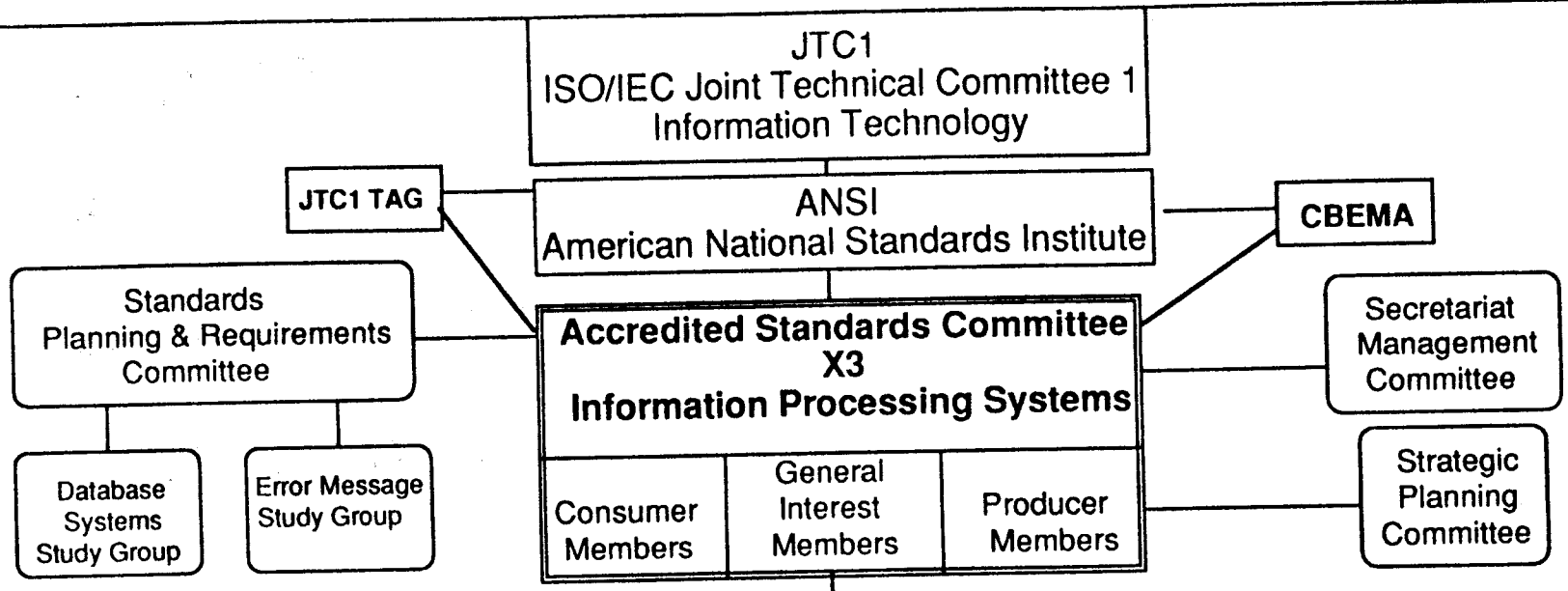
Gwen A. Morrison, IBM Corporation, 'Performance for a large, complex application', Proc CMG 86, Dec., 1986, 316-320 Proceedings CMG 87, Orlando, Florida, December 1987, pp 138-144

X3J18—THE REXX STANDARD

NEIL MILSTED
IX CORPORATION

X3J18 - The REXX standard

Organizational Chart for X3



X3 Technical Committees

- X3A1 Optical Character Recognition
- X3B5 Digital Magnetic Tape
- X3B6 Instrumentation Tape
- X3B7 Magnetic Disks
- X3B8 Flexible Disk Cartridges
- X3B9 Paper/Forms Layout
- X3B10 Credit/Identification Cards
- X3B11 Optical Digital Data Disks
- X3H2 Database
- X3H3 Computer Graphics
- X3H4 Information Resource Dictionary System
- X3H5 Parallel Processing Constructs for High Level Programming Languages
- X3H6 CIS
- X3J1 PL/I
- X3J2 Basic
- X3J3 FORTRAN
- X3J4 COBOL
- X3J7 APT
- X3J9 PASCAL
- X3J10 APL
- X3J11 C
- X3J12 DIBOL
- X3J13 LISP
- X3J14 FORTH
- X3J15 DATABUS
- X3J16 C++
- X3J17 Prolog
- X3J18 REXX
- X3K5 Vocabulary
- X3L2 Codes & Character Sets
- X3L3 Audio/Picture Coding
- X3L8 Data Representation
- X3S3 Data Communications
- X3T2 Data Interchange
- X3T3 Open Distributed Processing
- X3T4 Security Techniques
- X3T5 Open Systems Interconnection
- X3T6 Non-Contact Info. Systems Interfaces
- X3T7 Internationalization
- X3T9 I/O Interface
- X3V1 Text: Office & Pub Systems
- X3W1 Office Machines
- SC21 TAG - Information Retrieval, Transfer & Management for ISO
- SC22 TAG - Languages

Current Members - Vendors

- IBM
- Kilowatt
- The Workstation Group
- Quercus
- Commodore
- Wishful Thinking
- iX
- System Center

Current Members - Users

- Amdahl
- SHARE
- SLAC
- Computer Associates
- Tritus

Officers

- Brian Marks - Chairman
- Neil Milsted - Vice Chairman
- Ed Spire - Secretary
- Dean Williams - Editor
- Reed Meseck - Vocabulary Representative

To become a voting member:

- Attend two consecutive meetings
- Pay $250 in dues

To become an interested party, simply ask Neil Milsted

- CIS: 76050,3673
- Internet: nfnm@wrkgrp.com
- Phone: (312) 902-2149

The REXX standard is being done in two parts:

- Part one: Codify "The REXX Language" by Mike Cowlishaw (TRL).
- Part two: Define new features and extensions.
- Completion of the first part is scheduled for March 1994.

Work underway for standard:

- Syntax
- Assignments and variables
- Messages
- Parse
- I/O
- Arithmetic

Work to be done for standard:

- Commands
- Keyword Instructions & Function calls
- Builtin functions
- Trace
- Conditions
- Environment Interface
- Limits
- Standard Outline
- Standard Introduction
- Standard Rational

Information Processing - Programming Languages - The REXX Language

1 Scope

1.1

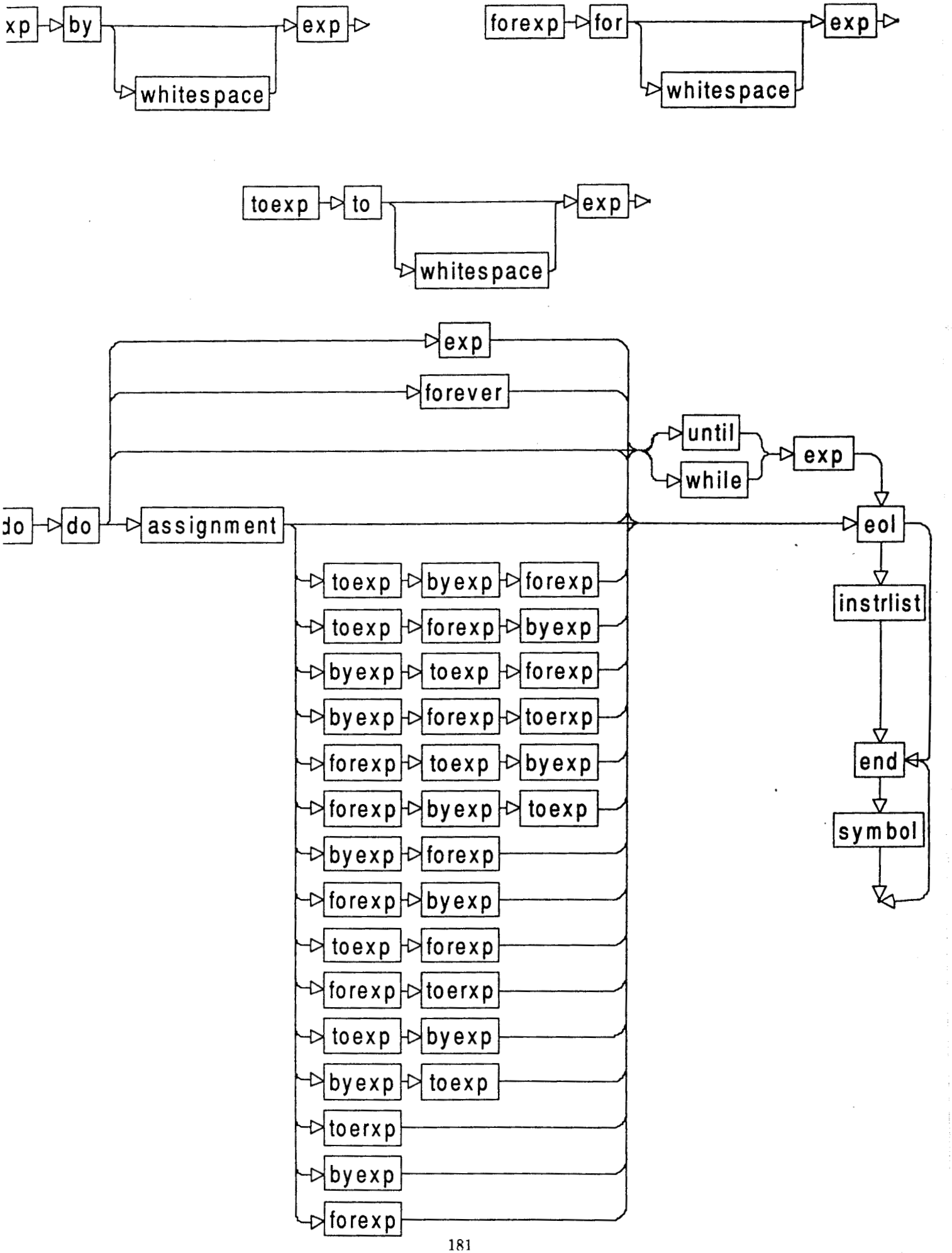
This Standard specifies the semantics and syntax of the computer programming language REXX by specifying requirements for a processor and for a conforming program. The scope of this standard includes

- the representation of REXX programs;
- the syntax and constraints of the REXX language;
- the semantic rules for interpreting REXX programs;
- the representation of input data to be processed by REXX programs;
- the representation of output produced by REXX programs;
- the restrictions and limitations imposed by a conforming implementation of REXX;
- the behavior of environmental interfaces.

1.2

This Standard does not specify:

- the mechanism by which REXX programs are transformed for use by a data-processing system;
- the mechanism by which REXX programs are invoked for use by a data-processing system;
- the mechanism by which input data are transformed for use by a REXX program;
- the mechanism by which output data are transformed after being produced by a REXX program;
- the size or complexity of a program and its data that will exceed the capacity of any specific data-processing system or the capacity of a particular processor;
- all minimal requirements of a data-processing system that is capable of supporting a conforming implementation;


xp
by
exp
whitespace
forexp
for
exp
whitespace
toexp
to
exp
whitespace
exp
forever
until
while
exp
do
do
assignment
eol
instrlist
end
symbol
toexp byexp forexp
toexp forexp byexp
byexp toexp forexp
byexp forexp toerxp
forexp toexp byexp
forexp byexp toexp
byexp forexp
forexp byexp
toexp forexp
forexp toerxp
toexp byexp
byexp toexp
toerxp
byexp
forexp

Issues discussed

- Normal comparision operator is intransitive
  - 9ea > 1e1 > 9e0 > 9ea (in EBCDIC).
  - 1ea < 9e0 < 1e1 < 1ea (in ASCII).
- Extensibility within the standard

X3J18 will not:

- Create a test suite
- Police vendors

IBM COMPILER AND LIBRARY FOR REXX/370

WALTER PACHL
IBM

IBM Compiler and Library for REXX/370

Walter Pachl

IBM Vienna Software Development Lab
c/o IBM Austria, Dept 00/705
Obere Donaustrasse 95
A-1020 Austria

May 1, 1992

3rd Annual REXX Symposium for Developers and Users

An Overview

- What's new?
- User Interfaces, CMS and TSO
- Performance Comparisons
- Building a Standalone Program
- Building REXX External Functions
- Packaging an Application Using REXX

In August 1991 two new IBM products were made generally available: the IBM Compiler and Library for REXX/370. The major news coming with these products will be covered in the first part of this presentation.

As the author has a close affiliation with user interfaces for invoking the compiler, a little description of these interfaces will be given — the nearly unchanged CMS invocation dialog and the new MVS foreground compilation panel.

Apart from other benefits, performance is a major aspect in compiling Rexx. The performance expectations and the results of a running a few benchmark programs will be shown.

Finally the new products offer new possibilities for packaging applications. The presentation will close by demonstrating how these possibilities can be used and what advantages can be expected.

Product Improvements

- Support MVS/ESA
- Smaller, Faster, Less Expensive

  Smaller compiler and compiled programs
  Faster compilation and program execution
  Lower price, in particular for smaller processors

- CONDENSE option to get significantly smaller compiled code
- DLINK option to allow for new packaging
- Support different parameter passing conventions on MVS
- Tolerate Interpret

One Product

- Cross Compiler
  Programs compiled on one system can be run on either system

- Gives enterprise the ability to purchase a single compiler

- Library for REXX/370 is system dependent

- Programs compiled with the predecessor product (CMS/Rexx Compiler) can be run without recompilation.

Compiler produces on either system

- A **compiled EXEC** that can be used instead of the source program. (Moving from one system to another or from one library to another on MVS may require conversion from one record format to a different one — a utility is provided for performing this task.)
- An **object module** that can be turned into an executable load module or that can be link-edited with other programs.

The compiler's system interfaces, the user interface, and the run-time support are, of course, system dependent. Ordered by product number with feature for CMS or MVS. Packaging in predecessor product was Compiler and Runtime Library or (on customer request) Runtime Library alone. The new products are Compiler alone and Runtime Library. User must order both products for compiling and executing programs.

Upward compatibility is maintained: Programs compiled with predecessor product can be run with the new Library. However, no downward compatibility — we move forward. Predecessor product is now withdrawn from marketing.

Customer Complaint:

"Compiled programs are (much) larger than source code"

A new compiler option is offered that allows to condense the object code.

- Compiled program uses less disk space
- Literal strings (and source lines) become illegible
- Unpacking the program for execution takes a little time.

Consider	should use CONDENSE		should not use CONDENSE
Machine bottleneck	I/O	CPU	Storage
Program location	Disk	Storage (single use)	Storage (shared use)
Program size	large	medium	small
Program execution	long-running		short-running
Program invocation	seldomly		frequently
Other	source/constant protection		DLINK required

Another Performance Boost

Significant (search) time is spent when external programs are invoked. CMS/Rexx Compiler allowed to create TEXT and MODULE for a Rexx program. A new compiler option, DLINK, generates weak external references for external functions and subroutines in compiled object modules.

These can be resolved by combining caller and callees, using the linkage editor.

Under MVS, modules must be pre-linked using an appropriate **stub** to accommodate the different parameter passing conventions.

Use of DLINK can make an application self-contained: No name clashes with user's environment.

- MVS has many different parameter passing conventions
- REXX programs understand arguments
- These arguments are passed in a table
- Compiler for REXX/370 supports four types of parameters

  MVS type, used in PARM= on JCL

  CALL type, used in the TSO/E CALL command

  CPPL type, used in TSO/E commands

  EFPL type, used in REXX external functions and function packages

- Source of "stubs" is provided as examples
- Can be modified for other parameter passing conventions

MVS Parameter Passing Conventions - NotesREXX

The MVS type and CALL type are very similar. The CALL type is limited to a single parameter, and it will have an address less than 16 Meg. The PARM= on JCL is a single parameter, but other programs that use this convention may have more than one parameter.

Register 1 points at a list of addresses, the last of which has the high order bit on. Addresses point at the individual argument strings each of which consists of a length field followed by the actual data.

The CPPL is a four word construct mapped by the macro IKJCPPL.
The DSECT for this macro is:

```
*********************************************************************
*    THE COMMAND PROCESSOR PARAMETER LIST (CPPL) IS A LIST OF
*    ADDRESSES PASSED FROM THE TMP TO THE CP VIA REGISTER 1
*********************************************************************
CPPL     DSECT
CPPLCBUF DS    A        PTR TO COMMAND BUFFER
CPPLUPT  DS    A        PTR TO UPT
CPPLPSCB DS    A        PTR TO PSCB
CPPLECT  DS    A        PTR TO ECT
```

The EFPL is a six word construct mapped by the macro IRXEFPL.
The DSECT for this macro is:

```
EFPL     DSECT
EFPLCOM  DS A             * RESERVED
EFPLBARG DS A             * RESERVED
EFPLEARG DS A             * RESERVED
EFPLFB   DS A             * RESERVED
EFPLARG  DS A             * POINTER TO ARGUMENTS TABLE
EFPLEVAL DS A             * POINTER TO ADDRESS OF EVALBLOCK
```

Stubs transform what comes in to what is expected. Under CMS, the compiled Rexx program is "self-adjusting."

Closer to Thee

- Interpret was flagged as SEVERE error by CMS/Rexx Compiler
  No compiled code was generated.

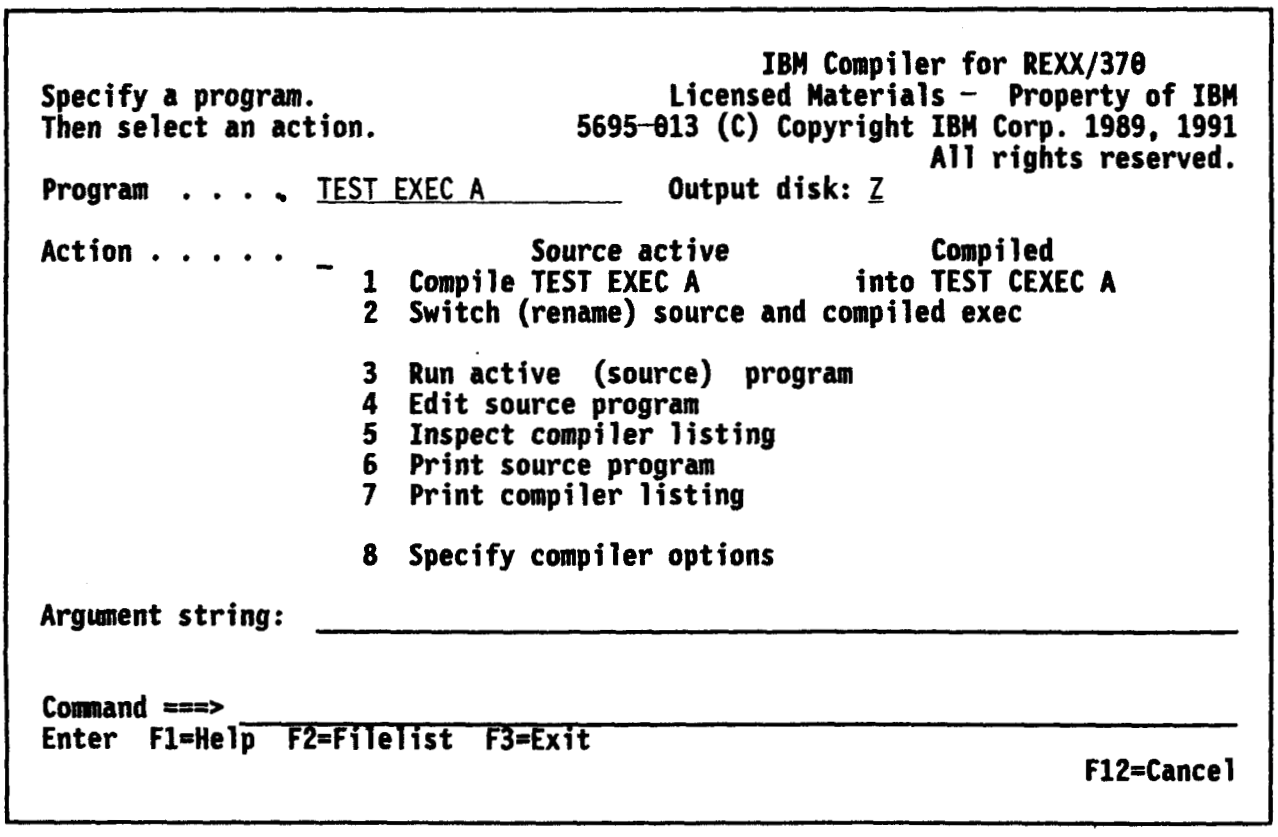
- Interpret is now flagged as ERROR
  Code is generated and causes a run-time error when the Interpret instruction is actually encountered.

  This can be avoided by:

```
Parse Version v
If left(v,5)<>'REXXC' Then          /* running compiled program     */
  Interpret instruction             /* we can interpret             */
```

- Support of Interpret is now an "Accepted Requirement"

CMS Compiler Invocation



REXXD - Compiler Invocation Dialog - Notes REXX

The compiler invocation dialog is intended to support all tasks involved in compiling for programmers as well as for casual users.

To use the compiler-invocation dialog under CMS enter the command:

rexxd test exec a

The panel appears as shown in the previous foil.

You may now select *Actions*:

1. Select Action 1 to compile the source program.
2. Select Action 2 to rename the source program and the compiled program.
3. Select Action 3 to run the currently active program.

If you need more information, refer to the online help by pressing the F1 key.

The name of the program to be compiled is carried over from the REXXD invocation or from the last invocation of this dialog. The name can, however, be changed on this panel. The panel is identical to that of the predecessor product, with one addition: the possibility to specify an output disk.

The panel indicates whether the source program or the compiled EXEC is currently active. The effect of switching between the two is reflected by appropriate highlighting.

Compiler options in effect can be displayed, changed, saved, and reset by selecting Action 8.

The Compiler Options Specification Panel REXX

```
                    REXX Compiler Options Specifications

Specify which output files you want, and their File-IDs

                                File identifiers
     Program name               ROULETTE  EXEC      G1
  Y  Compiler listing (Y/N/P)   =         LISTING   =
  Y  Compiled EXEC    (Y/N)     =         C*        =
  Y  TEXT file        (Y/N)     =         TEXT      =

Specify compiler messages to be issued
  I  FLAG   Minimum severity of messages to be shown (I/W/E/S/T/N)
  N  TERM   Display messages at the terminal (Y/N)
  N  SAA    SAA-compliance checking (Y/N)

Specify contents of compiler listing
  Y  SOURCE Include source listing (Y/N)
  Y  XREF   Include cross-reference listing (Y/S/N)
  N  LC     Number of lines per page (10-99 or, for no page headings, 0 or N)

Additional options
  N  SL     Support SOURCELINE built-in function (Y/N)
  Y  TH     Support HI immediate command (Y/N)
  S  NOC    Error level to suppress compilation (*/W/E/S)
  N  COND   Condense compiled program (Y/N)
  Y  DLINK  Include ESD and RLD in TEXT output (Y/N)
Special compiler diagnostics
  N    DUMP Produce diagnostic output (0-2047, Y, or N)

Command ===>
Enter  F1=Help  F2=Filelist  F3=Exit  F4=Save  F5=Refresh  F6=Reset
                                                          F12=Cancel
```

The options in effect are shown. Using entry fields and PF keys, the user can

- Change each compiler option individually (user input is checked and errors are diagnosed top down, field by field)
- Save the options in effect (in LASTING GLOBALV)
- Refresh the options (from LASTING GLOBALV)
- Reset the options to the installation defaults (taken from REXXC)

Help panels explain the available options and their meaning.

MVS Compiler Invocation

Under MVS, the usual methods of compiler invocation are supported:

- Foreground Compilation
- Background Compilation
- Cataloged Procedures

```
                        FOREGROUND REXX COMPILE
COMMAND ===>

ISPF LIBRARY:
   PROJECT ===> TEST
   GROUP   ===> LIB1      ===> LIB2      ===> LIB3      ===>
   TYPE    ===> REXX
   MEMBER  ===>                  (Blank or pattern for member selection list)

OTHER PARTITIONED OR SEQUENTIAL DATA SET:
   DATASET NAME  ===>

LIST ID ===>

COMPILER OPTIONS:             (extended REXXC options can be used)
        ===>
        ===>
```

Invoking the Compiler with ISPF Panels (MVS/ESA)

Under ISPF, you can invoke the Compiler from the Foreground REXX Compile panel and the Batch REXX Compile panel. The panels are similar to those for other high-level language compilers.

To use the Foreground REXX Compile panel:

1. Select FOREGROUND on the ISPF/PDF Primary Option Menu.
2. Select REXX Compiler.
3. Enter the appropriate data set names and (extended) compiler options. Extended compiler options allow to specify data set names where compiler output is to be stored.

From data entered on the panel, a command is built that allocates data sets as appropriate and that invokes the compiler with the appropriate compiler options. This command is, of course, implemented as a Rexx EXEC.

Rather unconventionally, background compilation **does not** employ file tailoring but uses also this REXXC command — albeit in batch.

Run-Time Performance Improvements

Programs with a lot of ...	TIMES faster than Interpreter	Performance Category
Arithmetic operations with default precision	6 - 10+	VERY HIGH
Arithmetic operations with other precision	4 - 25	
Assignments	6 - 10	
Changes to variables' values	4 - 6	HIGH
Constants and simple variables	4 - 6	
Reuse of compound variables	2 - 4	MEDIUM
Host commands	1 -	LOW

Performance - Notes

The performance improvements that you can expect when you run compiled REXX programs depend on the type of program. A program that performs large numbers of arithmetic operations of default precision shows the greatest improvement. A program that mainly issues commands to the host shows limited improvement because REXX cannot decrease the time taken by the host to process the commands.

Up to 30% CPU-load reduction have been reported on a heavily REXX-loaded machine.
" ... *better than last CPU upgrade.* On average 10-15% reduction are reported.

BENCHMAR EXEC

SPI-XA VM/XA System Product Interpreter Rel 5.6
REXX-370 IBM Compiler and Library for REXX/370 Rel 1.0

	TOTAL CPU TIME		RATIO
	SPI-XA	REXX-370	SPI/370
magnifier	0.76538	0.03749	20.42
forloop	7.97983	0.42967	18.57
whileloop	13.58915	0.57711	23.55
repeatloop	13.09567	0.54219	24.15
literalassign	10.34849	0.53132	19.48
memoryaccess	10.70410	0.56278	19.02
realarithmetic	3.29728	1.02355	3.22
realalgebra	2.69790	0.39420	6.84
vector	18.70649	1.89689	9.86
equalif	16.77867	0.91997	18.24
unequalif	16.74953	0.91375	18.33
noparameters	13.53182	1.29613	10.44
values	11.98723	2.49335	4.81
reference	19.90034	2.51308	7.92
wordscan400	21.16018	0.70951	29.82
command	0.68720	0.61554	1.12

An Example

- Compile the program using the OBJECT compiler option
- Turn it into a load module
  - Under MVS, by link-editing with the MVS-stub specified
  - Under CMS, by LOAD/GENMOD
- Place the load module
  - into an accessible library
  - onto an accessed minidisk
- Invoke it
  - from REXX (under MVS using Address LINKMVS)
  - from other languages, e.g., PL/I:

```
DCL  REXPGM ENTRY EXTERNAL OPTIONS(ASSEMBLER,INTER);
...
FETCH REXPGM;                          /* Bring it into storage   */
CALL REXPGM(VARSTRING);                /* Call the REXX program   */
RELEASE REXPGM;                        /* Release it from storage */
```

Performance opportunities

- External functions in load libraries generally found quicker (MVS)
- Function packages are first in the search order (Rexx search order)

Easy transition to load module

- Proceed as for standalone program
- but use EFPL stub instead of MVS

Building function packages

- Essentially a collection of external functions
- Each external Rexx function needs EFPL STUB (under MVS)
- When building package, naming convention consideration is important
- Description of packages in Rexx Reference manuals (system dependent).

Packaging Concept

- Can write an entire application in REXX
- External routines are directly LINKed
- Enabled through the use of DLINK compiler option

DLINK advantages

- Tremendous performance improvements from interpreted

  Mostly by eliminating search time

  Also due to inherent better performance in compiled REXX

- Functional isolation

  Each function can be in an external routine

  No name clashes with other system execs or commands

  No maintenance problems due to inadvertent modification of the exec

Packaging Considerations for MVS

- Naming convention unique to seven characters
- Use the DLINK option with all OBJECTs created by the Compiler for the application package
- All external functions use EFPL stub
- Main program may have different type of STUB
- All programs need to have a STUB created using a catalogued procedure
- Link edit all created programs together to create package

Example (for MVS)

- Begin with the following three execs

  DLT to drive the process

  CPUTIME to get the CPU time

  INCR simply returns the passed argument

DLT

```
/* REXX * DLT ************************************************************
* Performance Test for DLINK option:
* Invoke external routine INCR 50 times and tell how long it took
************************************************************************/
  n='DLT'
  Parse Version v .          /* Use Parse Version to see if compiled  */
  If left(v,5)='REXXC' Then what=n 'compiled'
                       Else what=n 'interpreted'
  Say what
  num=50

  t0=cputime()
  Call time 'R'
  Say num 'invocations of INCR will be measured'
  Do i=1 To num
    Call incr i
    End
  Say 'This took me' (cputime()-t0) 'CPU-seconds.',
                                          '(elapsed:' time('E')')'
```

One of the aspects of REXX that makes it an easy to use language is the ease with which it can concatenate strings. This is observed in the if statement, where the name of the exec in the variable n is concatenated with the string indicating whether the exec is compiled or interpreted.

Also note that the PARSE VERSION gives the programmer the ability to determine if the exec is running compiled or interpreted. If needed, different logic paths can be followed, depending on whether the exec is being interpreted or run as compiled program.

Similarly Parse Source lets you determine how the exec was invoked and on which system.

CPUTIME

```
/* REXX * CPUTIME ************************************************************
* Return the cpu-time used up so far
*******************************************************************************/
  Parse Version v
  Parse Source  s

  Parse Var s sys .

  Select                            /* Figure out which system we are on */
    When sys='CMS' Then Do
      qt="DIAG"(8,'Q TIME')
      Parse Var qt . 'VIRTCPU=' mm . ':' +1 ss +6
      cpu=mm*60+ss
      End
    When sys='TSO' Then Do
      cpu=sysvar('SYSCPU')
      End
    When wordpos(sys,'PCODS OS/2')>0 Then Do
      t=Time()
      Parse Var t hh ':' mm ':' ss
      cpu=(hh*60+mm)*60+ss
      End
    Otherwise Do
      Say 'System' sys 'is unknown to CPUTIME'
      cpu=0
      End
    End
  If word(s,2)='COMMAND' Then
    Say 'CPU time used so far:' cpu
  Else                              /* When an external routine     */
    Return cpu                      /* Return the CPU time          */
```

INCR

```
/* REXX * INCR ***************************************************************
* Performance Test for DLINK option:
* Return the argument
*******************************************************************************/
  Return arg(1)
```

Building this package

- Interpreted Case

  Normally all three execs reside in SYSEXEC

  Invoked by entering DLT from TSO command line

  CPUTIME and INCR are external routines

- Hence, DLT will need CPPL STUB

- CPUTIME and INCR need EFPL STUB

- All OBJECTs are created with DLINK option

- Catalogued procedure used for each one

Creating the final module

- Link edit all load modules together

  After each has its STUB added

  Using INCLUDE and NAME control cards

- In this example, BJVLIB is the DDNAME of the library containing the programs

- Control cards would be

```
  INCLUDE BJVLIB(DLT)
  INCLUDE BJVLIB(INCR)
  INCLUDE BJVLIB(CPUTIME)
  ENTRY   DLT
NAME DLT(R)
```

Under CMS, simply

```
LOAD DLT INCR CPUTIME
GENMOD DLT
```

Output Comparison

- When Interpreted

```
DLT interpreted
50 invocations of INCR will be measured
This took me 1.30 CPU-seconds. (elapsed: 11.14)
```

- When Compiled using OBJECT and DLINK

```
DLT compiled
50 invocations of INCR will be measured
This took me 0.74 CPU-seconds. (elapsed:  0.89)
```

Under CMS

```
This took me 0.23 CPU-seconds. (elapsed: 1.891623)
vs.
This took me 0.06 CPU-seconds. (elapsed: 0.142037)
```

Significant Performance Improvement

- Interpreted uses 75 % more CPU
- Interpreted is 12.5 times slower in elapsed time

The IBM Compiler and Library for REXX/370

- Open new programming possibilities
- Support both function and application packaging
- Give you more time on your own CPU!
- And we did not even touch
  - Program Documentation
  - Plug-compatibility
  - 31-Bit Capability (VM/XA)
  - New language on old systems

OS/2 PROCEDURES LANGUAGE 2/REXX

RICHARD K. MCGUIRE AND STEPHEN G. PRICE
IBM

OS/2 Procedures Language 2/REXX "A Practical Approach to Programming" and "Adding REXX Power to Applications"

Richard K. McGuire
Stephen G. Price

IBM Corporation
G09/20M
P.O. Box 6
Endicott, NY 13760

....

OS/2 Procedures Language 2/REXX

A Practical Approach to Programming

OS/2 *Rexx*

What is REXX?

- Powerful end-user programming language
- Easy to learn, easy to remember
- Can powerfully extend any application
- Common language available on all SAA systems
- Becoming an ANSI standard (X3J18 Committee)

OS/2 *Rexx*

Why REXX?

- Small, easy to use, yet powerful language
- Programming interfaces for application extension
- Rapid development of an interpreter, performance boost of compiler technology

OS/2 *Rexx*

Keep the Language Small

- Friendlier to new users
- Documentation is smaller and simpler
- Few exceptions or special cases (low "astonishment factor")
- Users can "embrace" the entire language

OS/2 *Rexx*

Natural Datatyping

- No internal or machine representation is exposed to the user
- Single number concept

```
Say "The interest is a*b'%'

Say 5 + 1.0 + 0.54 +
1.23e-2
```

OS/2 *Rexx*

No Defined Size or Shape Limits

- Data sizes limited only by available memory
- Limits are set using "human readable" values
- SmallTalk-like dynamic data-typing

OS/2 *Rexx*

Powerful Symbol Manipulation

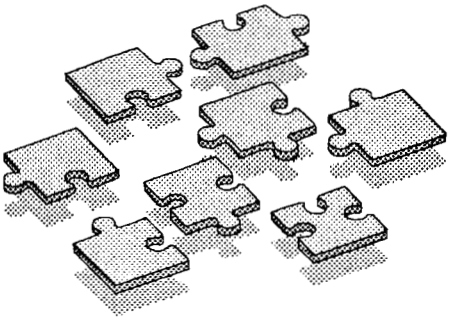
- Natural concatenation
- Powerful string parsing ability
- Many functions for string and word manipulation

```
Parse Arg first initial last
Say "Hello" first'.'

pos = wordpos(first, list)
if pos <> 0 then
  nickname = word(list, pos)
```

OS/2 *Rexx*

System Independence

- The REXX language is independent of both operating system and hardware
- Suitable for any system or application environment
- Part of the IBM Systems Application Architecture



OS/2 *Rexx*

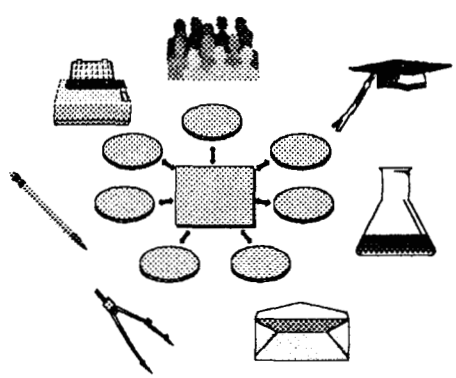
REXX Uses

- Tailoring user commands (".CMD" files)
- End-user problem solving
- Universal macro or scripting language
- Prototyping Applications
- Education

OS/2 *Rexx*

Universal Macro Language

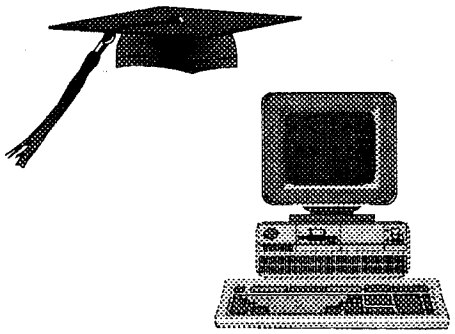
- Editors
- Spreadsheets
- Language preprocessors
- Communication programs
- Rexx can be the macro language for any application



OS/2 *Rexx*

REXX is a Good Introduction to Programming

- Easy to learn
- Easy to program
- Few new concepts required
- Powerful debugging features
- No separate compile or link step



OS/2 *Rexx*

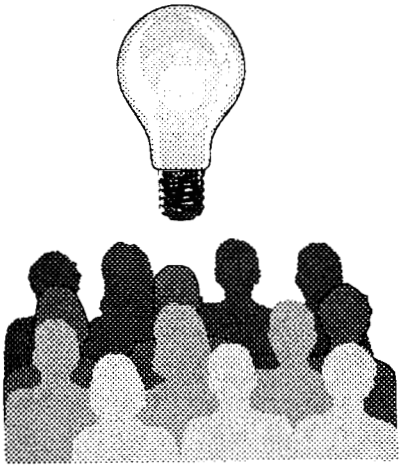
What's New in OS/2 2.0?

- Interpreter runs in 32-bit mode
- Dramatically improved performance
- New 32-bit interfaces
- 16-bit interfaces still supported
- On-line REXX reference manual
- OS/2 utility functions
- New 32-bit sample programs in toolkit
- On-line programming interfaces reference
- RXHLLAPI interface
- SAA Communications interface
- Communications Manger configuration
- LAN utilities

OS/2 *Rexx*

More than a Fancy .CMD Language

- Fill multiple roles on OS/2
- Places more power in the hands of users
- Powerful automation of OS/2 operations
- Powerful extensions to OS/2 applications



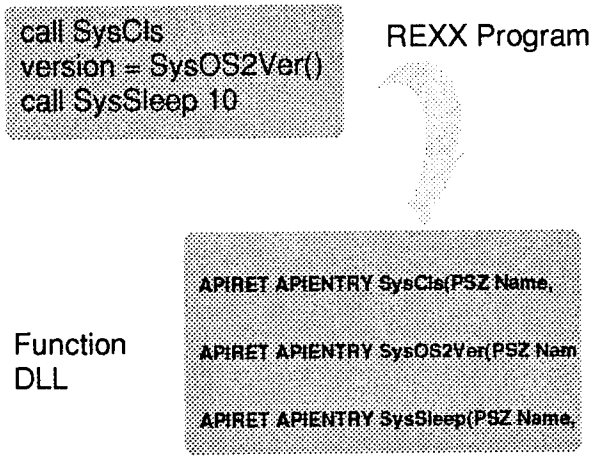
OS/2 *Rexx*

OS/2
Procedures Language
2/REXX

Adding REXX Power to Applications

OS/2 *Rexx*

Creating New REXX Functions



OS/2 *Rexx*

Function Registration

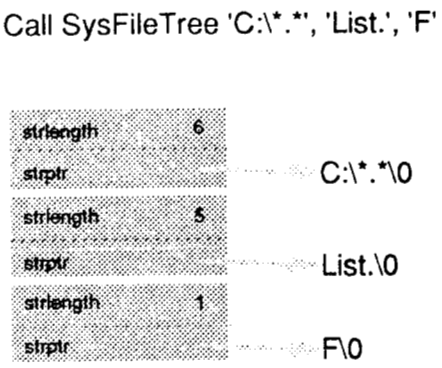
- REXX external functions are registered with RxFuncAdd
- Acts as a form of program linkage

```
Call RxFuncAdd 'SysCls',,
   'REXXUTIL', 'SysCls'
```

OS/2 *Rexx*

RXSTRINGs

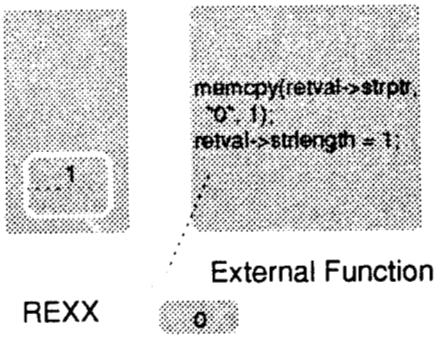
- External functions are passed arguments as RXSTRINGs
- Defined as a pointer and length pair defining a REXX character string



OS/2 *Rexx*

RXSTRING Return Values

- External functions pass an RXSTRING value back to REXX
- The function can use the buffer provided by REXX or create a new one



OS/2 *Rexx*

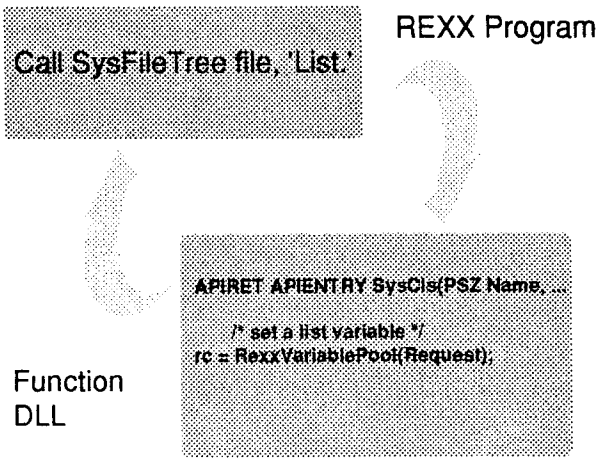
Function Packages

- REXX external functions can be registered from C code also

```
RexxRegisterFunctionDll(
  "SysCls", "REXXUTIL",
  "SysCls");
```

OS/2 *Rexx*

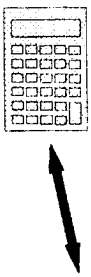
Accessing REXX Variables



OS/2 *Rexx*

Using REXX for Macros

- An application can call the REXX interpreter to run any REXX program



```
/* calculate factorials */
parse arg number, factor
accum = 1
do i = 1 to factor
  accum = accum * i
end
return accum
```

OS/2 *Rexx*

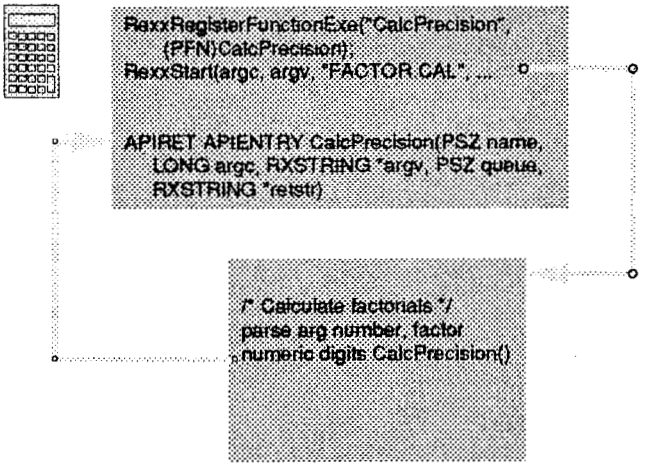
Invoking REXX

- An application can call use the REXX interpreter with the RexxStart programming interface

```
rc = RexxStart(argc, argv,
   "FACTOR.CAL",
   NULL, NULL,
   RXFUNCTION,
   NULL,
   &return,
   &retstr);
```

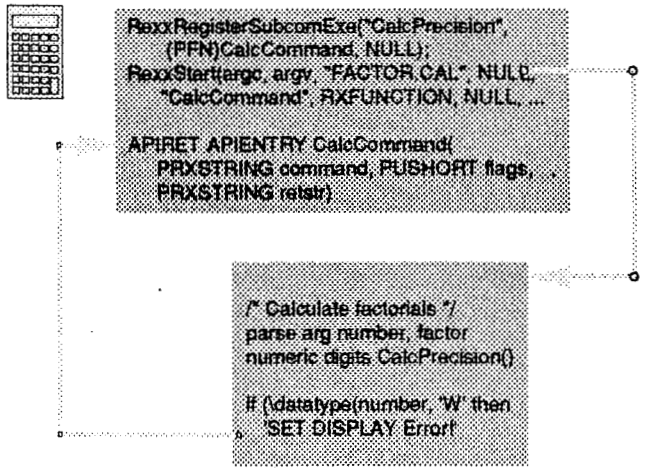
OS/2 *Rexx*

Application External Functions



OS/2 *Rexx*

Subcommand Handlers



OS/2 *Rexx*

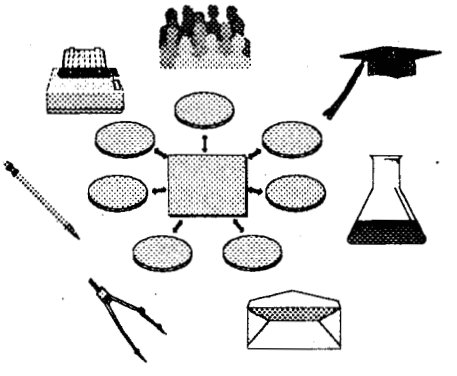
And Still More...

- Exits to tailor REXX program behavior
- REXX programs executed directly from storage
- Macro Space repository for REXX programs
- Halting a running REXX program
- Tracing a running REXX program
- Subcommand handlers as dynamic link libraries

OS/2 *Rexx*

REXX
The Universal Macro Language

- Same language used for all applications
- Places control into user hands, making people more productive
- Easily added to any application



OS/2 *Rexx*

INTERFACING WITH REXX

ANTHONY RUDD
DATEV

Interfacing with REXX

ABSTRACT

This aim of this paper is to give an overview of the interfaces available in REXX, and to show how these interfaces can be used. This paper deals only with the MVS environment – however, most other environments (e.g. OS/2) offer similar facilities.

Although REXX is a powerful language in its own right (especially now that REXX compilers are available), there are certain features missing (for example, processing of VSAM files, direct SQL processing). Furthermore, there are REXX features (e.g. parsing) that can simplify the processing of programs written in conventional languages (Assembler, PL/I, COBOL, etc).

REXX caters for both these situations by providing interfaces. There are two forms of interface:

- high level
- low level.

High-level interfaces are invoked directly from a REXX exec. *Low-level* interfaces are those routines (services) provided by the REXX processor.

There are three forms of high-level interface:

- function
- (address) environment
- program invocation.

A **function** can be written in either REXX or a conventional programming language. To improve performance functions can be physically grouped together as a **function package**. A function is invoked by its name, and serves to extend the standard functions provided with REXX (e.g. `WORD`, `WORDINDEX`). A function may be passed arguments, and may return a value (the **function return value**).
An **address environment** can only be written in a conventional programming language. High-level interfaces may (and normally will) make use of low-level REXX interfaces. REXX as an address environment processes any non-REXX statements. A user-address-environment extends the standard REXX environments (e.g. `MVS`, `TSO`).
A **program invocation** is made with the `LINK` or `ATTACH` command.

1. INTRODUCTION

REXX implementations offer many interfaces for using REXX services from programs written in conventional programming languages. This paper describes only those interfaces of interest to the applications developer – there are a number of other interfaces which can be used by systems specialists to customise the system.

The interfaces can be grouped into the following categories:

- program invocation of a REXX exec
- programs as REXX functions (and the grouping of such programs into function packages)
- program access to REXX variables
- stack operations
- general service routines.

1.1 High-level REXX interfaces

High-level REXX interfaces are invoked directly from REXX execs. Such interfaces can be regarded as being extensions to the REXX language.

Standard address environments:

- `ISPEXEC` (ISPF Dialog Manager)
- `ISREDIT` (ISPF/PDF Edit Macro)
- DB2 (program that runs in the DB2 environment)
- QMF.

Typical user environments:

- `REXXDB2` process SQL query
- `REXXVSAM` process VSAM dataset.

Representative examples of user functions:

- `SHIFT` function (perform bit-shift on REXX variable)
- `SIN` function (calculate trigonometric sine value).

1.2 Low-level REXX interfaces

The most useful low-level REXX interface routines:

- `IRXEXCOM` access REXX variables
- `IRXEXEC` invoke REXX exec
- `IRXINIT` process REXX environment
- `IRXJCL` invoke REXX exec (batch mode)
- `IRXLOAD` load exec
- `IRXRLT` get result
- `IRXSTK` access REXX stack.

REXX programs (i.e. programs that make use of REXX services) can access certain REXX control blocks:

- Argument List (AL). The Argument List describes the input arguments passed to a function. Each argument passed to the function has one Argument List entry (consisting of two words) in the Argument List. The Argument List is terminated with two words each containing binary -1 (`X'F...F'`).
- External Functions Parameter List (EFPL). The EFPL describes the external arguments for a function; the pointer to the input arguments and to the result field. The input arguments are defined in the Argument List. The result is defined in the Evaluation Block (EVALBLOCK).
- Environment Block (ENVBLOCK). The ENVBLOCK describes the REXX operating environment. An ENVBLOCK is automatically created when the REXX environment is initiated. The ENVBLOCK is principally used by the application developer to obtain error messages.
- Evaluation Block (EVALBLOCK). The EVALBLOCK describes the result passed back from a function.
- Execution Block (EXECBLK). The EXECBLK specifies the information necessary to locate an external exec.
- In-Storage Control Block (INSTBLK). The INSTBLK describes (address and length) the individual records (lines) of a REXX exec contained in main-storage. The `IRXLOAD` service can be used to build the INSTBLK.
- Shared Variable (Request) Block (SHVBLOCK). The SHVBLOCK describes the variable to be accessed from the variable pool. SHVBLOCKs can be chained together.

- Vector of External Entry Points (VEEP). The VEEP contains the addresses of the external REXX service routines.

Most of these control blocks are read-only, although some can be altered (INSTBLK, SHVBLOCK).

2. HIGH-LEVEL INTERFACES

2.1 MVS-TSO/E implementation

The MVS-TSO/E implementation allows a REXX exec to run in several environments, both dialogue and batch. From within this invoking environment the ADDRESS instruction can be used to select a sub-environment for non-REXX statements. This sub-environment is the interface to other components, for example, the ISPEXEC sub-environment for ISPF Dialog Manager services.

2.1.1 Invocation

A REXX exec can be invoked from:

- TSO/ISPF dialogue
- TSO batch
- MVS batch.

The REXX exec is stored as member of a partitioned dataset (library). The name of this dataset must be made available to the REXX interpreter.

2.1.2 Linkage to host (MVS-TSO/E) environment

A REXX exec can link to components from the host environment. The ADDRESS instruction is used to set the host environment.

Example:

```
ADDRESS TSO "TIME";
```

invokes the TSO TIME command.

2.1.3 Linkage to programs

A REXX exec can pass control to a program written in a conventional programming language. The program is invoked with either the ATTACH or LINK host command. The ATTACH command invokes the program asynchronously (i.e. as a separate task), the LINK command invokes the program synchronously. The program is loaded from the program (load) library assigned to the environment.

The program may be passed a single parameter, which may contain subparameters. The invoked program receives two parameters on entry:

- the address of the parameter string;
- the length of the parameter string (full-word).

Note: This is not the standard MVS program linkage convention. TSO/E V2R3.1 offers new facilities: LINKMVS, ATTCHMVS, LINKPGM, ATTCHPGM. These pass multiple parameters according to MVS conventions.

2.1.4 Interface with ISPEXEC (ISPF Dialog Manager)

REXX execs invoked from the TSO/ISPF environment can use the ADDRESS ISPEXEC instruction to access ISPEXEC (ISPF Dialog Manager) services. The parameters for the ISPEXEC service are passed as a normal REXX string, i.e. may be a literal, symbol or mixture. However, ISPEXEC accepts only upper-case characters. The return code from the ISPEXEC service is set into the RC special variable.
REXX execs and ISPF Dialog Manager share the same function pool, with two restrictions:

- variable names longer than 8 characters cannot be used in ISPF;
- the VGET and VPUT services cannot be used with stem variables.

Example:

```
panname = "PAN1";
ADDRESS ISPEXEC "DISPLAY PANEL("panname")";
SAY RC;
```

uses ISPEXEC to display panel PAN1, the return code from the service is displayed.

2.1.5 Interface with ISREDIT (ISPF/PDF Edit macro)

The ISPF/PDF Editor can invoke a procedure to perform processing on a dataset – this procedure is called an Edit macro and can be a REXX exec. The ADDRESS ISREDIT instruction invokes Edit macro services. The parameters for the ISREDIT service are passed as a normal REXX string, i.e. may be a literal, symbol or mixture. The return code from the ISREDIT service is set into the RC special variable.

Edit macros can make full use of REXX facilities. The powerful string processing features of REXX make it an ideal language for the implementation of Edit macros.

Example:

```
/* REXX Edit macro */
ADDRESS ISREDIT;
"MACRO (STRING)"
"FIND" string "NEXT"
IF RC <> 0 THEN SAY "search argument not found";
"END" /* terminate macro */
```

2.1.6 Interface with DB2 (Database 2)

The TSO DSN command is used in initiate the DB2 session. The DB2 RUN subcommand is used to invoke a program which is to run in the DB2 environment.

The DB2 subcommands to invoke the program, and to terminate the DB2 session, RUN and END, respectively, are set into the stack in the required order before the DB2 session is initiated.

Note: The subcommands cannot be passed directly, as is the case with CLISTs.

Example:

```
QUEUE "RUN PROGRAM(TDB2PGM) PLAN(TDB2PLN) LIB('USER.RUNLIB.LOAD')";
QUEUE "END";
ADDRESS TSO "%DSN"; /* invoke DB2 */
```

or

```
ADDRESS ISPEXEC "SELECT CMD(%DSN)"; /* invoke DB2 with ISPF services */
```

2.1.7 Interface with QMF (Query Management Facility)

With QMF Version 3 Release 1 the SAA Callable Interface (DSQCIx) is now available for REXX. This means that there are now two methods of invoking QMF:

- Callable Interface
- Command Interface.

The Callable Interface:

- ISPF not required
- QMF does not need to be active.

The Command Interface:

- requires ISPF
- requires QMF to be active.

The Command Interface invocation of QMF is more involved; two steps are required:

- initiate the QMF session (program DSQQMFE), and execute a QMF procedure;
- this QMF procedure passes control to a REXX exec, which in turn uses the QMF Command Interface (CI, program DSQCCI) to process a QMF command.

The following three QMF examples all perform the same function: run the QMF query Q1.

2.1.7.1 Callable Interface - Version 1

Example:

```
/* REXX - QMF Callable Interface */
ADDRESS "TSO";
/* allocate QMF files */
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
CALL DSQCIX "START (DSQSSUBS=DB2T,DSQSMODE=INTERACTIVE"; /* start QMF */
CALL TESTRC;
CALL DSQCIX "RUN QUERY Q1"; /* run query */
CALL TESTRC;
CALL DSQCIX "EXIT"; /* terminate QMF */
CALL TESTRC;
EXIT; /* terminate exec */
TESTRC:
  IF DSQ_RETURN_CODE > 4 THEN DO;
    SAY "QMF RC:" DSQ_RETURN_CODE;
    SAY  DSQ_MESSAGE_TEXT;
  END;
RETURN;
```

2.1.7.2 Callable Interface - Version 2

Example:

```
/* REXX - QMF Callable Interface */
ADDRESS "TSO";
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
CALL DSQCIX "START (DSQSSUBS=DB2T,DSQSMODE=INTERACTIVE"; /* start QMF */
CALL TESTRC;
ADDRESS "QRW"; /* QMF environment */
"RUN QUERY Q1" /* run query */
CALL TESTRC;
"EXIT" /* terminate QMF */
CALL TESTRC;
EXIT; /* terminate exec */
TESTRC:
  IF DSQ_RETURN_CODE > 4 THEN DO;
    SAY "QMF RC:" DSQ_RETURN_CODE;
    SAY  DSQ_MESSAGE_TEXT;
  END;
RETURN;
```

Version 2 is basically the same as version 1, except that the QMF environment `QRW` is used.

2.1.7.3 Command Interface

Example:

Phase 1 - Initiate QMF session (`DSQQMFE` program). The following exec allocates the (minumum) QMF files, initiates QMF session and invokes the QMF procedure `QP1`:

```
/* REXX - QMF COMMAND INTERFACE */
ADDRESS "TSO";
"ALLOC F(DSQDEBUG) DUMMY REUS"
"ALLOC F(DSQPNLE) DSN('qmf.test.dsqpnle') SHR REUS"
"ALLOC F(ADMGGMAP) DSN('qmf.test.dsqmape') SHR REUS"
ADDRESS "ISPEXEC";
"SELECT PGM(DSQQMFE) NEWAPPL(DSQE) PARM(S=DB2T,I=USER.QP1)"
```

Phase 2 - The QMF procedure `QP1` passes control to the TSO procedure (REXX exec) `QR2`:

```
TSO %QR2
```

Phase 3 - The `QR2` exec invokes the QMF Command Interface (`DSQCCI` program) to process the specified QMF commands (this REXX exec actually causes the QMF query (`Q1`) to be run):

```
/* REXX */
ADDRESS "ISPEXEC";
"SELECT PGM(DSQCCI) PARM(RUN Q1)"
"SELECT PGM(DSQCCI) PARM(INTERACT)"
"SELECT PGM(DSQCCI) PARM(EXIT)" /* terminate QMF */
```

Fig. 1 illustrates the use of the QMF Command Interface.

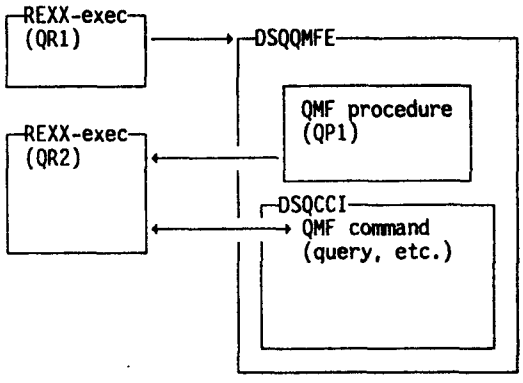


Fig. 1 – Schematic use of QMF Command Interface

2.2 User interfaces

User programs can be invoked as:

- function (e.g. `x = funct(p1,p2,...);`)
- host command (e.g. `ADDRESS userenv; "cmd p1 p2 ...";`)
- program (e.g. `LINK "pgm p1 p2 ...";`).

The most suitable interface depends on such aspects as:

- the form of the arguments to be passed (a natural calling sequence);
- the form of the results to be returned;
- the programming language used.

2.2.1 Function interface

A user function receives zero or more parameters (parsed in the Argument List), and must return a function result (in the Evaluation Block). Fig. 2 illustrates the function interface.

Example:

```
y = SIN(x);
```

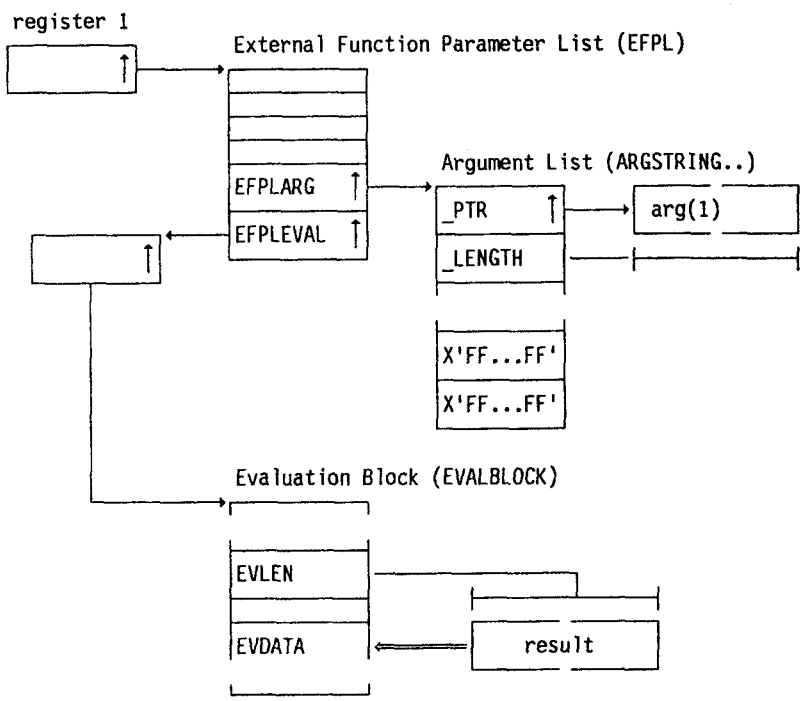


Fig. 2 – Function interface

2.2.2 Host command interface

A host command is processed by the currently active environment, i.e. the environment activated with the ADDRESS command. All non-REXX commands are passed to the host command environment. A host command cannot directly return any data (other than a return code for the command) – data can be passed back in the stack or as (stem) variables. Fig. 3 illustrates the host command interface.

Many installations have a single router program that passes control to the appropriate processing program.

Example:

```
ADDRESS USER;
"REXXVSAM READ DDNAME GE ALPHA(STEM A.";
```

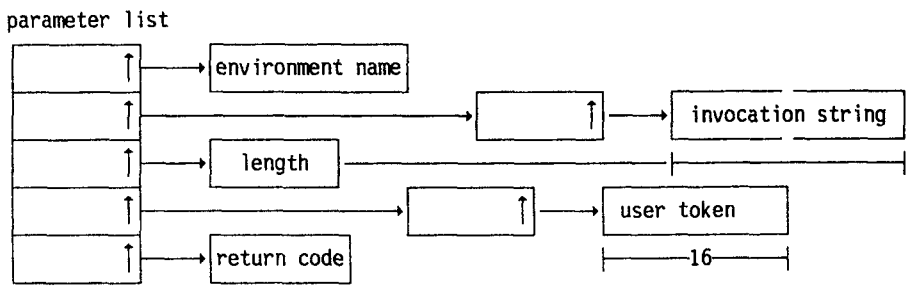


Fig. 3 – Host Command Environment Interface

2.2.3 Program invocation interface

A program can be directly invoked with the ATTACH (asynchronous) or LINK (synchronous) command. This is the only way of invoking a C/370 Version 1 program. *Note*: The parameters passed to a program do not conform to the MVS calling convention. Fig. 4 illustrates the program invocation interface.

Example:

```
ADDRESS LINK "ALPHA BETA GAMMA";
```

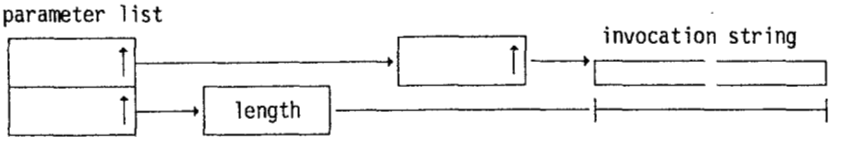


Fig. 4 – Program invocation (via LINK, ATTACH)

3. LOW-LEVEL INTERFACES

3.1 General conditions

The low-level interfaces are subject to the following conditions:

- Programs can be written in Assembler, COBOL, PL/I, and C/370 Version 2 (to a limited extent Version 1). Not all high-level programming languages provide full support for all the required facilities.
- Programs using REXX services must use 31-bit addressing (AMODE 31).
- Numeric fields are in binary format, either fullword (4 bytes) or halfword (2 bytes).
- Standard calling conventions are used:
  - register 15 – entry point address;
  - register 14 – return address;
  - register 13 – address of save-area.
- The return code is passed back in register 15 (PL/I: PLIRETV variable, COBOL: RETURN-CODE special register, C: function return value). Many routines also set an error message in the Environment Block.
- Parameter address lists passed in register 1 must have the high-order bit set in the last address word.
- Standard macros (in the SYS1.MACLIB system macro library) are available for use by Assembler programs to map the more important control blocks. Programs written in high-level programming languages (e.g. COBOL, PL/I) must themselves define the required control block structures – Fig. 5 shows the equivalent field types in various programming languages.

type	Assembler	PL/I	COBOL VS II	C
address	A	PTR	POINTER	*
character string	CLn	CHAR(n)	PIC X(n)	char [n+1]
fullword	F	FIXED BIN(31)	PIC S9(9) COMP	int
halfword	H	FIXED BIN(15)	PIC S9(4) COMP	short
hexadecimal	X	BIT(8)	X' ... '	0X

Fig. 5 – Equivalent field types

Notes:

1. Only the most important information for the interfaces is described in this paper – the appropriate manual should be consulted if a more detailed description is required.
2. The entry *symbol.*. in diagrams denotes that *symbol* is used as prefix to the field names in the corresponding block. The diagrams show only the significant fields. Any fillers at the end of field layout figures are omitted.

Sample PL/I program:

```
BETA: PROC OPTIONS(MAIN);
DCL IRXSTK EXTERNAL OPTIONS(RETCODE,INTER,ASSEMBLER);
DCL PLIRETV BUILTIN;
DCL 1 FC CHAR(8);              /* function code */
DCL 1 ADDR_ELEM PTR;           /* pointer to data */
DCL 1 LEN_ELEM FIXED BIN(31); /* length of data */
DCL 1 FRC FIXED BIN(31);       /* function return code */
DCL 1 ELEM CHAR(256) BASED(ADDR_ELEM); /* data */
FC = 'PULL';                   /* function */
FETCH IRXSTK;                  /* load address of entry point */
CALL IRXSTK(FC,ADDR_ELEM,LEN_ELEM,FRC);
IF PLIRETV = 0 THEN PUT SKIP LIST (SUBSTR(ELEM,1,LEN_ELEM));
END;
```

This PL/I program retrieves and displays the next element from the data stack.

3.2 Invocation of a REXX exec

There are three ways of an application program to invoke a REXX exec:

- using the IRXJCL program;
- using the TSO Service Facility (IJKEFTSR program);
- using the IRXEXEC program.

These three methods are listed in order of ease of use. This is also the order of increasing flexibility, e.g. the IRXEXEC program interface offers more flexibility than the IRXJCL program interface but is more difficult to use.

3.2.1 Interface from programs to batch REXX (IRXJCL)

Programs written in a conventional language can use IRXJCL to invoke a REXX exec. Fig. 6 shows the form of the parameter as passed from the invoking program.

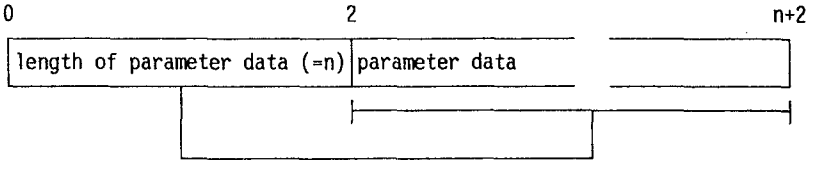


Fig. 6 – Format of parameter passed to IRXJCL

3.2.2 Invocation of a REXX exec using the TSO Service Facility (IJKEFTSR)

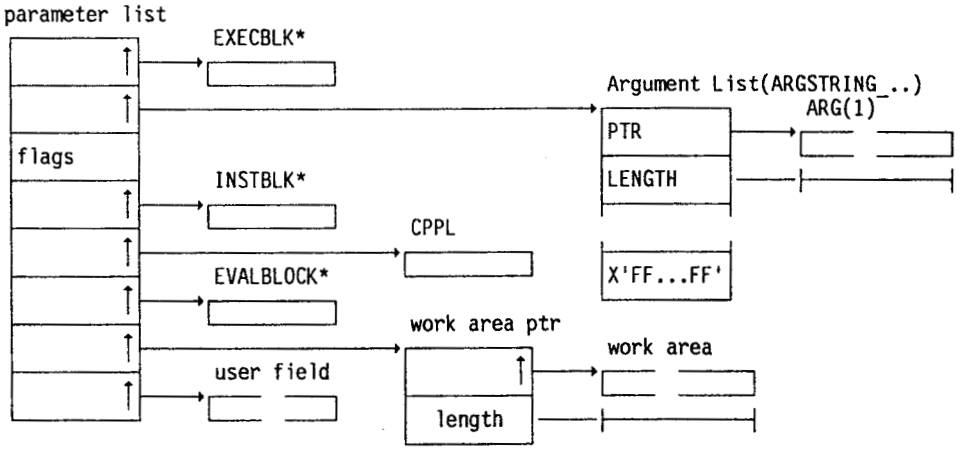
REXX execs can also be invoked from the TSO environment (either dialogue or batch) with the TSO Service Facility (IJKEFTSR program) – the TSO Service Facility has the alias TSOLNK.

3.2.3 Interface from program to REXX processor (IRXEXEC)

The IRXEXEC routine is the most flexible method of invoking a REXX exec:

- it can invoke either an internal or external exec;
- it can pass more than one parameter.

If the INSTBLK address is zero, an internal exec is invoked, otherwise an external exec is loaded using the information in the EXECBLK (EXEC_BLK_DDNAME – library ddname, EXEC_BLK_MEMBER – member name). Fig. 7 illustrates the IRXEXEC service.



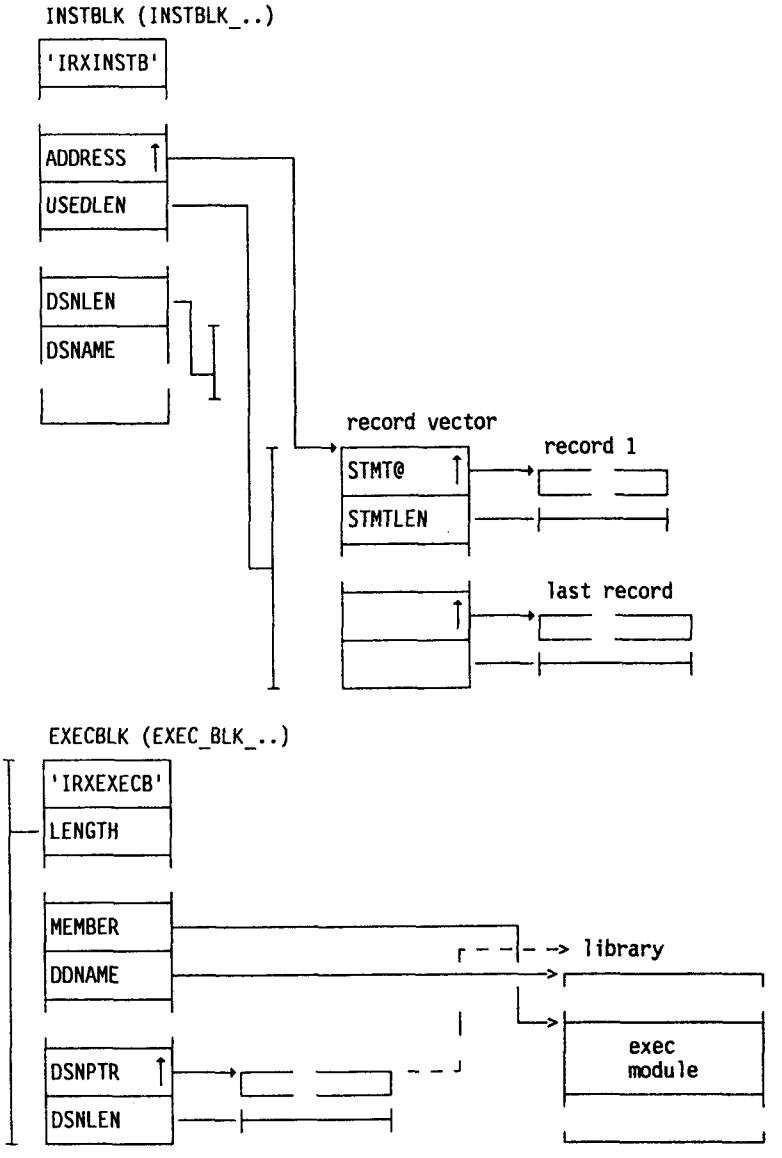
*•Detailed diagram follows (in part 2)

Fig. 7 – IRXEXEC interface (part 1 of 2)

3.3 Program access to REXX variables (IRXEXCOM service)

Programs running in a REXX environment can use the IRXEXCOM service to access variables in the environment pool. Fig. 8 illustrates the IRXEXCOM service. The following functions are available:

- copy value
- set variable
- drop variable
- retrieve symbolic name
- set symbolic name
- drop symbolic name
- fetch next variable
- fetch user data.



– – implicit (only informative)

Fig. 7 – IRXEXEC interface (part 2 of 2)

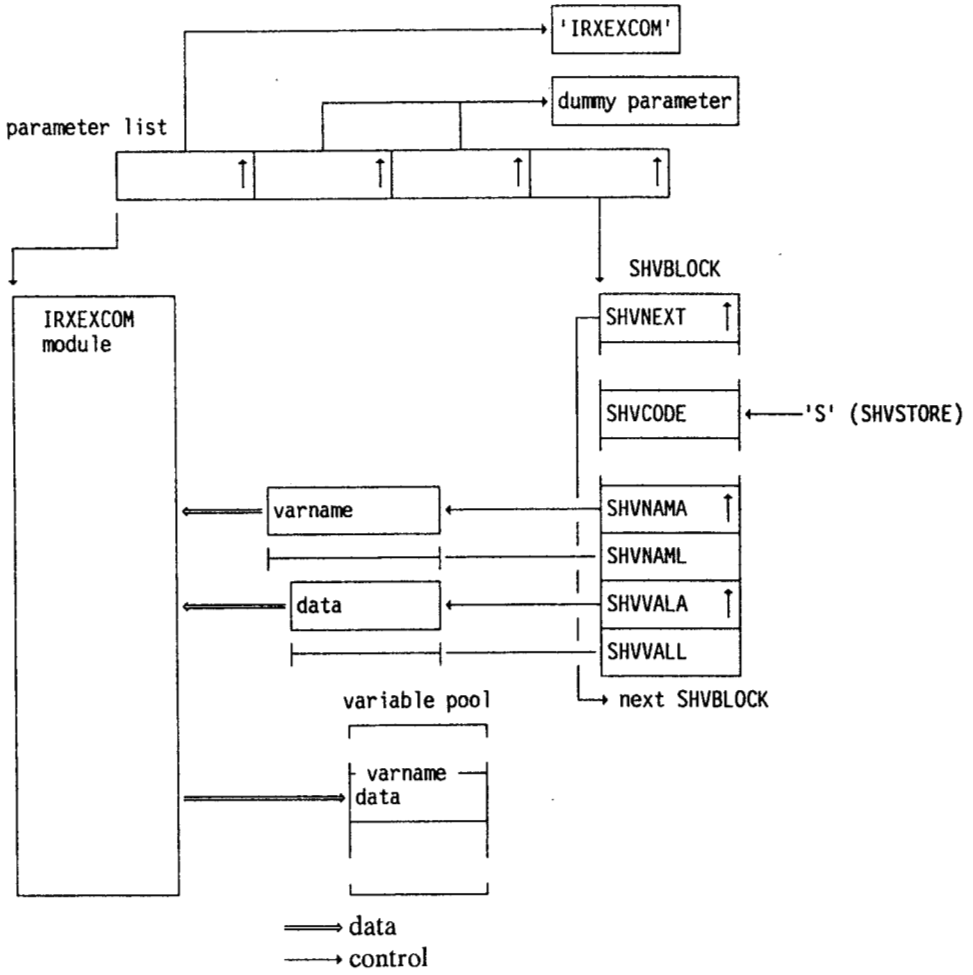


Fig. 8 – IRXEXCOM service to store a variable

3.4 Stack processing (IRXSTK service)

Programs can use the IRXSTK service to perform processing on the current stack. The operations:

- DELSTACK
- DROPBUF
- MAKEBUF
- NEWSTACK
- PULL
- PUSH
- QELEM
- QSTACK
- QUEUE
- QUEUED

have their standard function.

The two operations:

- DROPTERM
- MAKETERM

are used by system routines to coordinate stack access from TSO and ISPF. These operations should not be used by application programs.

3.5 Function interface

Programs written in a conventional programming language and stored as a load module in a library can be invoked as external REXX functions or subroutines. A function differs from a a subroutine in that it must return a value.

3.5.1 Function package

For reasons of efficiency, functions can be grouped together as a **function package** – function packages are searched before the other libraries. Three classes of function package can be defined:

- user function package
- local function package
- system function package.

The system support personnel will usually be responsible for the local and system function packages, and so they will not be discussed in this paper, although the general logic is the same as for the user function package.

A function package consists of a **function package directory** and functions. The function package directory is a load module contained in the load library – IRXFUSER is the standard name for the load module defining the user function package. Fig. 9 shows the diagrammatic representation of a function package.

The function package directory contains the names of the functions (subroutines) as invoked from a REXX exec and a pointer to the appropriate load module. This pointer can have one of two forms:

- The address of a load module which has been linkage edited together with the function package directory – such load modules must be serially reusable, as they are loaded only once.
- The name of a load module which will be loaded from the specified load library.

3.5.1.1 Function directory

The Function Directory defines the functions contained in a function package. The Function Directory consists of a header and one entry for each function contained in the Function Directory.

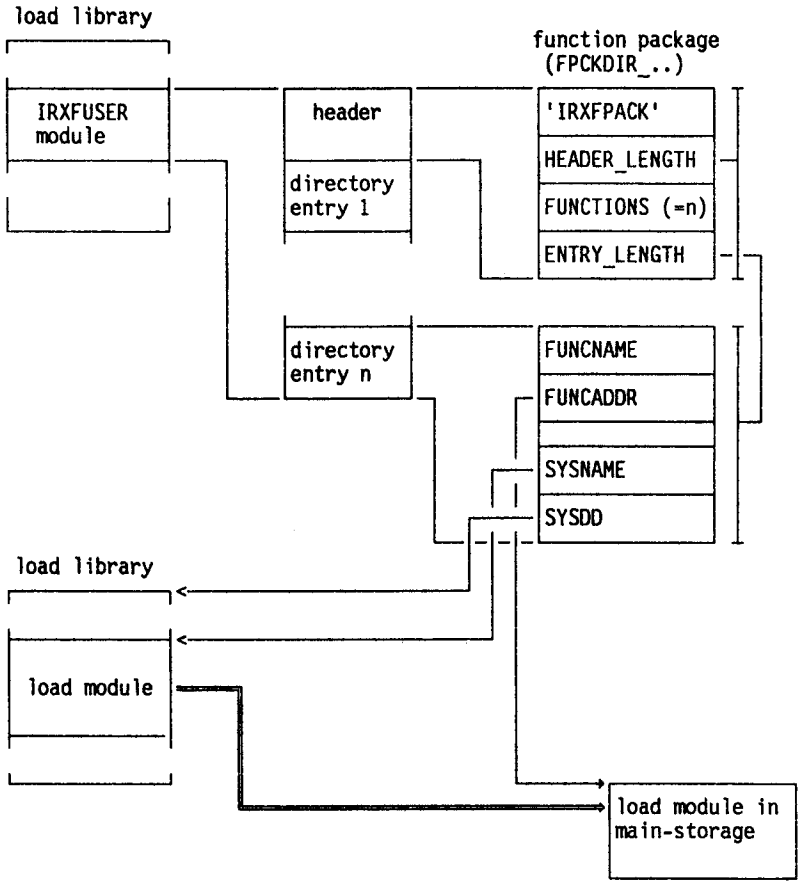


Fig. 9 – Diagrammatic representation of a function package

Sample Function Package Directory:

```
IRXFUSER CSECT
         DC    CL8'IRXFPACK'      identifier
         DC    AL4(SOD-IRXFUSER)  length of header
         DC    AL4(ND)            no. of entries in directory
         DC    FL4'0'             zero
         DC    AL4(LDE)           entry length
SOD      EQU   *                  start of directory (first entry)
         DC    CL8'FDIGIT'        function name
         DC    VL4(FDIGIT)        address, reserved
         DC    FL4'0'             reserved
         DC    CL8' '             name of entry point
         DC    CL8' '             DD-name of load library
LDE      EQU   *-SOD              length of directory entry
* next entry
         DC    CL8'FGEDATE'       function name
         DC    AL4(0)             address, 0 = load from library
         DC    FL4'0'             reserved
         DC    CL8'FGEDATE'       name of entry point
         DC    CL8'ISPLLIB'       DD-name of load library
EOD      EQU   *                  end of directory
ND       EQU   (EOD-SOD)/LDE      no. of directory entries
         END
```

This sample Function Package Directory contains two functions:

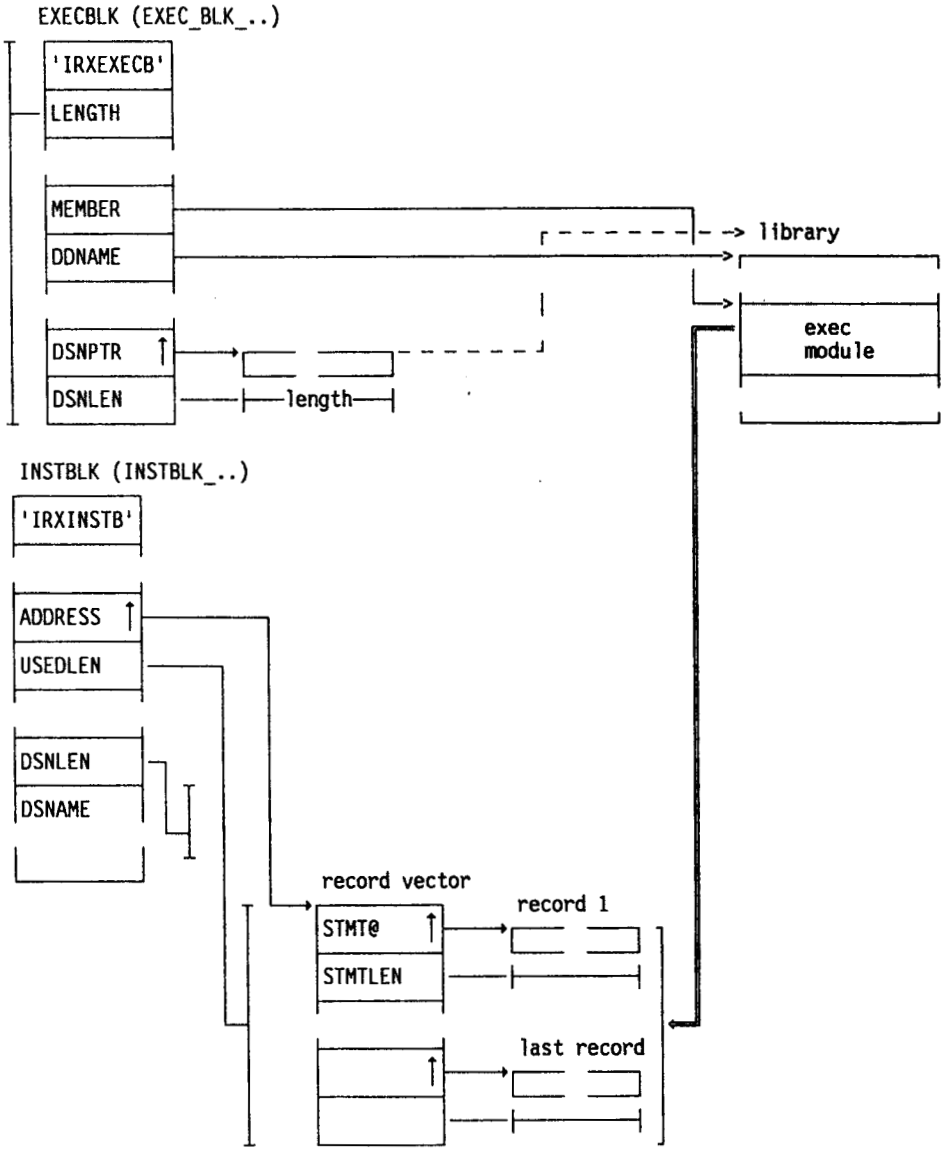
- FDIGIT – linkage edited with the Function Package Directory;
- FGEDATE – to be loaded from the ISPLLIB library.

3.6 Load routine – IRXLOAD service

The load routine (IRXLOAD) can be used in several ways:

- load an exec into main-storage – this creates the In-Storage Control Block for the exec;
- check whether an exec is currently loaded in main-storage;
- free an exec;
- close a file from which execs have been loaded.

IRXLOAD is also used when the language processor environment is initialised and terminated. Fig. 10 illustrates the IRXLOAD service (load function).



– - implicit (only informative)

Fig. 10 – IRXLOAD interface

3.7 Initialisation routine – IRXINIT service

The initialisation routine (IRXINIT) can be used in two ways:

- initialise a new environment;
- obtain the address of the current Environment Block.

The first function is normally only used by system specialists. The second function is used principally to access an error message which has been set by a service routine. Fig. 11 illustrates the ENVBLOCK.

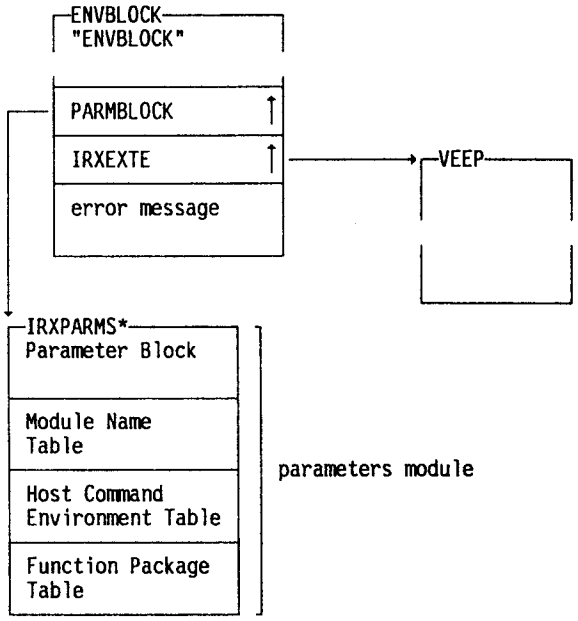


Fig. 11 – ENVBLOCK

3.8. Get result – IRXRLT service

The get result routine (IRXRLT) can be used in two ways:

- fetch result set by an exec invoked with the IRXEXEC service;
- allocate an Evaluation Block of the specified size.

This paper is adapted from my book:
Practical Usage of REXX
published in 1990 by Ellis Horwood Limited, Chichester.

Anthony Rudd, April 1992.

REXX IN THE CICS ENVIRONMENT

DAVID I SHRIVER
IBM

REXX in the CICS Environment

May 5, 1992

David I. Shriver

IBM
Mailstop 01-03-50
5 West Kirkwood Blvd.
Roanoke, TX 76299-0001
(817) 962-4142

Third REXX Symposium, Anapolis, Maryland

ABSTRACT:

CICS/REXX is an IBM internal implementation of REXX, the IBM SAA Procedures Language, under CICS/MVS and CICS/ESA. Specifically, it provides REXX environment support under CICS for both the TSO/E Version 2 REXX interpreter and the REXX/370 compiler. This environment support includes interface routines for storage management, I/O handling and other miscellaneous REXX facilities. It also includes providing a command-level interface to CICS from REXX, and also provides interfaces to other CICS based products, such as IBM's OfficeVision/MVS.

Contents

CICS/REXX Overview

Copyright

Trademarks

The following terms used in this paper, are trademarks or service marks of IBM Corporation in the United States or other countries:

AIX, CICS/ESA, CICS/MVS, DB2, GDDM, IBM, QMF, MVS/ESA, OfficeVision, OS/2, PROFS, REXX

Disclaimer

This discussion of REXX under CICS does not imply that IBM either does, or does not, have plans to incorporate all, or part of, this function into a product.

Purpose of this paper

The purpose of this paper is to share information on an internal IBM implementation of REXX under CICS so as to promote technical discussion and generate customer feedback.

Function/Feature Highlights

As follows are some of the highlight features of CICS/REXX:

- Full REXX language support under MVS CICS
- Support for both compiled and interpreted EXECs
- CICS based text editor for REXX EXECs and data
- VSAM based file system for REXX EXECs and data
- Support for popular EXEC CICS commands (not complete yet)
- Support for Subcommands written in REXX
- Support for application macros, written in REXX
- High-level client/server architecture support
- Command definition of REXX Subcommands
- Flat/Universal default REXX Subcommand space
- Transparent CICS Psuedo-conversational terminal support
- Support for system and user profile EXECs
- Shared EXECs in virtual storage

- Nested INCLUDE support in EXEC Loader
- EXEC Suspend/Resume support
- REXX interface to OfficeVision/MVS
- Compatibility support for several popular VM/CMS commands

Full REXX language support under MVS CICS

CICS/REXX is currently at REXX language level 3.46 and supports all REXX language statements and built-in functions, as described for MVS in the *SAA Common Programming Interface Procedures Language Reference, SC26-4358.*

Support for both compiled and interpreted EXECs

CICS/REXX includes support for both interpreted and compiled EXECs. Compiled and interpreted EXECs can be freely intermixed. Such a combination is powerful because the use of the interpreter provides a very productive development environment (quick development cycle, source level interactive debug, CICS based development) whereas the compiler allows the developed REXX code to be later optimized for the performance requirements of critical production systems. Since compiled and interpreted REXX EXECs can be intermixed transparently, compilation can be done selectively on those modules that need it most, and the replacement of interpreted REXX EXECs can be done gradually, without affecting system function.

CICS based text editor for REXX EXECs and data

KEDIT, a full function text editor, similar to the VM/CMS XEDIT and TSO ISPF/PDF editors is provided as part of CICS/REXX, so EXECs can be written and modified directly under CICS, and from CICS based application platforms, such as OfficeVision/MVS.

VSAM based file system for REXX EXECs and data

CICS/REXX includes a REXX file system that is hierarchically structured (similar to OS/2, AIX and the VM Shared File System), and automatically provides each REXX user with a file system in which to store EXECs and data. There is a FILELIST utility to facilitate working with this file system, the KEDIT editor will support editing members of this file system, and EXECs to be run are loaded from this file system. This library (file) system is VSAM RRDS based for performance, security and portability reasons.

Support for popular EXEC CICS commands (not complete yet)

Support for several EXEC CICS commands is already included in CICS/REXX, and support for all popular CICS Command Level commands is planned.

Support for Subcommands written in REXX

CICS/REXX supports the ability for users to write new REXX subcommands in REXX. These subcommands do not function as nested REXX EXECs, and unlike nested REXX EXECs will have the ability to get and set the values of REXX variables in the user EXEC that invoked them. Thus subcommands written in REXX can have similar capabilities as subcommands written in Assembler or other languages. Therefore subcommands can be quickly written in REXX to speed systems development (in a building block structure), and then can selectively be rewritten in Assembler, for example, at a later date, as performance requirements dictate. Or they may simply be compiled with the REXX compiler.

Support for application macros, written in REXX

One of the strongest uses for REXX is to support the extension of existing applications via Application Macros. This provides a natural mechanism for the extension of product or application capability, and does so in a natural building block fashion. Since REXX Application Macros are separate from application code, this means they can be effectively created by application users, with little chance of causing application failure.

High-level client/server architecture support

CICS/REXX includes built-in client/server architecture support to facilitate the use of this important new technology in systems development and to help enable a higher level of host involvement in Enterprise-wide computing solutions.

Command definition of REXX Subcommands

CICS/REXX includes as one of its basic facilities, the ability for systems administrators and users to easily and dynamically define new REXX subcommands, either on a system-wide or user-by-user basis. One of the greatest strengths of REXX is its ability to be interfaced cleanly with other products, applications and system services. The goal for providing a command definition facility for new or existing subcommands is to facilitate the rapid and consistent high-level integration of various products and services together through the use of REXX. REXX subcommand definition is accomplished though the CICS/REXX DEFCMD and DEFSCMD subcommands.

Flat/Universal default REXX Subcommand space

The CICS/REXX subcommand definition facility also optionally supports the use of a flat (or universal) REXX subcommand space. This would be consistent with the REXX goal of maintaining simplicity and naturalness. With this support, all REXX subcommands (which might span interfaces for multiple applications) would be mapped into one default subcommand environment. This would allow one global and consistent subcommand set to be provided and documented, and would free programmers from having to understand which subcommand environment a subcommand exists in, and it would remove the need to be constantly switching subcommand environments (switching environments is accomplished with the ADDRESS statement).

Transparent CICS Pseudo-conversational terminal support

CICS/REXX supports both conversation and pseudo-conversational terminal I/O in REXX based transactions. Transparent, underlying pseudo-conversational support is provided if the PSEUDO ON subcommand is specified in an EXEC. This means that a program written in REXX can be switched between conversational and pseudo-conversational without changing the program structure.

Support for system and user profile EXECs

To facilitate CICS/REXX system and user environment tailoring, CICS/REXX will attempt to execute a SYSPROF EXEC and user PROFILE EXECs if they exist. The SYSPROF EXEC must exist in the system base directory and is invoked before the first user EXEC runs after a CICS system restart. A user's PROFILE EXEC (if it exists in that user's base directory) will be invoked before the first EXEC is invoked for this user (after a CICS system restart).

Shared EXECs in virtual storage

CICS/REXX supports both shared and unshared copies of REXX EXECs residing in virtual storage. Preloaded shared EXECs improve interactive response time of REXX applications, and sharing reduces the total virtual storage requirement.

Nested INCLUDE support in EXEC Loader

Often in real world REXX programming, a programmer is torn between making a function or subroutine written in REXX, internal or external to a REXX application. There are significant performance and variable sharing advantages to making a subroutine internal. But there is a major drawback if this subroutine is to be shared by several REXX EXECs. Duplicate copies must be placed in all programs that use the subroutine and it is a nightmare trying to update all of these copies and to keep them the same, whenever a change is made to a subroutine. CICS/REXX nested INCLUDE support improves this situation by allowing one or more INCLUDE statements to be placed in REXX source files so that subroutines can be maintained as separate external files but be included as internal routines at EXEC load time. An additional opportunity is that only one copy of the source for a particular subroutine needs to be loaded into virtual storage, no matter how many EXECs are using it as an internal routine.

EXEC Suspend/Resume support

When CICS/REXX is used as a Procedures Language under CICS, there are times that EXECs are used to contain command lists of CICS commands (applications) to be STARTed. Since these CICS transactions often require a terminal to be available before they can run, a way is need to cause the transaction the EXEC is running under to end to free up the terminal, causing the EXEC to be temporarily suspended so it can be resumed later at the point after it was suspended. The CICS/REXX SUSPEND subcommand provides this capability.

REXX interface to OfficeVision/MVS and ASF Version 2

OfficeVision/MVS and ASF Version 2 provide CICS based Application Integration platforms. Applications may be integrated with each other or with Office functions, for added value. CICS/REXX has special support to facilitate REXX EXECs being invoked from OfficeVision/MVS (or from ASF Version 2) and/or OfficeVision/MVS services being invoked from REXX EXECs in a CICS environment.

Compatibility support for several popular VM/CMS commands

Compatibility support for several important VM/CMS commands has been provided in CICS/REXX to make it easier to port or migrate VM based EXECs to a CICS environment. This helps preserve customer investments in VM/CMS EXECs when such a migration is necessary, it helps facilitate the porting of a considerable amount of VM/CMS REXX based software to the CICS environment, and helps preserve investments in VM/CMS training and allows VM/CMS users to come up to speed more quickly in the CICS/REXX environment.

CICS/REXX Benefits

Business Solutions

CICS/REXX is an ideal system to use to deliver superior, valuable, and appropriate business solutions, in a much more timely and cost effective manner.

CICS/REXX is an excellent platform for the delivery of CICS based business solutions for the following reasons:

- ***CICS/REXX is a simpler, uniform, self contained development environment***

  To use CICS/REXX, a new programmer no longer has to learn TSO, ISPF, JCL, COBOL and much of the technical detail of CICS (such as the proper use the translator).

  For both new and experienced programmers, there is no longer the need to constantly switch back and forth between TSO and CICS, all the while flipping between several manuals for needed system and development information.

  CICS/REXX is a uniform, self contained system that supports development directly under CICS and provides everything the average CICS developer needs in one manageable package.

- ***CICS/REXX allows solutions to be delivered sooner***

  There is a combination of benefits that CICS/REXX delivers to cause major gains in application productivity and reduced delivery time. The REXX language alone has proven to be a major boost to application productivity because of its high level, simplicity, strong parsing and naturalness. On top of that, the synergy of an interpreter/compiler combination is a strong addition. The interpreter provides a very quick development cycle and provides excellent source-level interactive debugging capability. Experience has proven a ten-fold improvement in productivity, when using REXX over conventional languages and techniques, to be a conservative figure. The ability to deliver business solutions more quickly is an important advantage in today's competitive marketplace.

- ***CICS/REXX makes practical highly incremental development***

  One of the biggest advantages of the fact that CICS/REXX includes support for a REXX interpreter as well as a compiler, is that the interpreter, with its quick, natural development cycle and excellent source-based interactive debugging make it feasible to switch to an Incremental Development Methodology. This is also sometimes called a Prototyping Development Methodology.

  REXX is of a sufficiently high level to be a powerful language for quick and expressive prototyping, and because of the compiler and the robustness of the language, is also suitable for serious application development. This provides an ideal situation where prototypes can be quickly developed to test system feasibility, to gather requirements, to get customer involvement, and can then be "grown" into useful production systems.

  This approach bypasses the nasty surprises of finding late in the development cycle that the project isn't technically feasible, of delivering a system that isn't what the customers want (or even what they thought they were going to get), or of major schedule overruns without any deliverables. And a final nice benefit of incremental development is that it has the tendency to test the code much more thoroughly during development, usually resulting in much higher quality code.

- ***CICS/REXX applications are easier to maintain and support***

  REXX based applications, being high-level in nature, are usually smaller than comparable applications in other languages (in lines of code) and are easier to read. And the interactive source level debug capability of the REXX interpreter makes it easier to locate and fix problems, and to deliver enhancements. This equates to a cheaper, more effective support of REXX based applications.

- ***CICS/REXX is useable by business people***

  Quite often business people who best understand the business and their needed solutions have ideas as to ways to modify, customize, or enhance applications that they use. But when they discover the difficulty involved and the investment in education required, they often give up in frustration. But those who have persevered have often delivered some of the most timely and on-target solutions. One of the greatest strengths of REXX is its simplicity and naturalness on one hand, and its powerful capability, on the other hand. CICS/REXX will make it possible for CICS application users to more extensively customize and even extend their applications, without requiring a programmer. This will provide more timely, on-target solutions, and will free real programmers up for involvement in more strategic projects.

  This is in line with what many industry analysts believe is a fundamental shift happening in the model for application development within Fortune 1000 companies. Business is organizing into more autonomous units, competitive pressures have increased (demanding quicker solutions), and new technology such as workstations and Client/Server computing, have made it feasible for much application development to be moved from central MIS to line-of-business organizations.

- ***CICS/REXX makes complex systems manageable***

  One of the design goals of REXX has always been to bend over backwards to make programming simple and natural for the REXX programmer, even if this makes things complicated for the REXX implementer. The simplistic power of REXX makes it a good candidate for today's complex business systems, because it simplifies them and thus makes them more manageable.

  CICS/REXX organizes (breaks down) complex systems in several related ways to make them more manageable. One is that it promotes a natural building block approach made up of EXECs, application macros, and subcommands transparently implemented in a variety of languages. In close relationship to these, is built-in Client/Server computing support that encourages greater host involvement in the Enterprise-wide Client/Server Distributed Computing model, with all of the many benefits this entails. Another strength of CICS/REXX in this arena, is the facilities it has for integrating multiple applications, products, and system facilities together into one seamless package, from a user perspective, which greatly simplifies systems development efforts.

The KEDIT story: The KEDIT text editor was written so as to be externally similar to the IBM XEDIT and ISPF/PDF editors, so as to minimize user retraining needs.

KEDIT is an excellent example of the sophistication that is possible with REXX based applications under CICS/REXX. And it it is a good example of the development productivity improvements that are possible.

KEDIT was written completely in REXX (except for some general purpose primitives it uses that are written in Assembler, as will be the case with most REXX applications) by Kevin Wriston, who was new to REXX. Kevin wrote a useable editor (which he used for his own REXX development) in three weeks, and has spent a total of about three person months, developing KEDIT. And the finished product is only about 1000 lines of REXX code, a mere fraction of the XEDIT Assembler code.

The other nice thing is the quickness with which Kevin can respond to requests for changes or enhancements to KEDIT (often quicker than the average programmer can go get a cup of coffee).

Kevin recently added REXX macro support to KEDIT, a demonstration that under CICS/REXX, applications written in REXX, can also support application macros, written in REXX, an important new capability.

Investment Protection

The IBM MVS CICS computing environment has one of, if not the, largest concentration of customer production applications and data, **in the world.** There has been tremendous customer investment in CICS based mainframe systems, CICS based application development, data collection for CICS based systems, and employee education relating to the use and support of CICS based systems. CICS/REXX helps to preserve and enhance the usefulness of this investment.

Not only does CICS/REXX enhance the delivery of traditional CICS based production applications, it makes the CICS environment suitable for a broader range of information processing activities. With CICS/REXX, it is now practical to also perform end-user computing, prototyping, and application development, directly within the CICS environment.

Also, CICS/REXX, which currently runs under MVS CICS, was designed so it can be later ported to provide REXX support for CICS running under OS/2, AIX, VSE and OS/400. One goal is to provide consistent REXX support across these environments, so as to preserve customer investments. Another is to facilitate the use of cooperative processing, between these environments.

User Productivity

CICS/REXX can enhance CICS user productivity in several ways:

- Allows simpler, but more flexible application customization by typical users. This allows them to more effectively tailor these applications to their individual business needs.
- Advanced users will be able to make application enhancements that normally would have been reserved for professional application developers. This has the effect of providing solutions needed to improve productivity and satisfy business needs more quickly. It also reduces the demand on application developers for application changes and frees them to work on more significant long range efforts.
- Facilitates the use of a prototyping methodology. This means that the users of an application in development participate very closely in the application development process (if they do not own the process outright). The end result is that the users, who have the best understanding of the business and their needs can better ensure that the application solution delivered matches their needs. This close involvement will also have the added benefit that the human factor needs (useability) of the user audience will also tend to be addressed in the application, enhancing their productivity.

Growth Enablement

Because CICS/REXX reduces the complexity of application development and maintenance, it makes it feasible to develop and support larger and more complex systems. This is true because:

- REXX is a high level language whose major emphasis has been to be natural to use and to free its user (the programmer) from any unnecessary detail. Thus REXX programs tend to be shorter and easier to follow.
- REXX encourages the use of a more manageable building block approach to systems development. The integrated Client/Server and dynamic subcommand definition capabilities of CICS/REXX even further enhance this.
- Major productivity improvements achieved by using the powerful interactive source level debugging capability and the quick development cycle of the REXX interpreter will make larger, more sophisticated development efforts feasible.

Systems Management

One of the major strengths of REXX is its usefulness as a Procedures Language. When used in this way, it can automate sequences of CICS system and application Systems Management activities, providing greater productivity and reliability.

Also, since CICS/REXX supports application development (and testing) directly under CICS, systems management can be greatly simplified. For example, the need for many CICS developers to have a TSO userid, could be removed, in many situations. Reducing the volume of TSO userids that need to be administered and managed would equate to an overall reduction in systems management activities.

CICS/REXX General Architecture/Implementation

General Design Goals

Some specific design goals/objectives for this project were:

- Provide the CICS or OfficeVision/MVS user or application developer/integrator with a simple but powerful self contained REXX based environment with the necessary interfaces to productively accomplish application development, application integration and customization.
- Provide a high-level, easy to use, REXX interface to the existing CICS command level facilities so as to improve the productivity of existing, experienced CICS developers.
- Provide a high-level, easy to use alternative programming environment that removes the need for casual programmers (or users) to learn the CICS environment.
- Bring product interfaces together, in one, self contained place for both ease of use and added synergy.
- Provide a flexible CICS REXX implementation that can be easily customized, tailored or extended by customers for their own unique needs.
- Capitalize on new REXX/370 compiler, C/370 Version 2 and other products
- Provide an environment conducive to the building block approach to code development. One of important needs in this area is to allow administrators and users to replace one type of building block or primitive with one written in a different language or with a different name without having to change the programs that reference it. Support interfaces to multiple programming languages.
- Provide an architecture capable of supporting large complex systems
- Perform acceptably for use in large production CICS environments
- Provide security sufficient for CICS production environments
- Exploit CICS/ESA and MVS/ESA when available

Basic structure of REXX running under CICS

CICS/REXX support provides a program called REXX which is used to load and invoke REXX EXECs within a CICS region. This program uses the Clearly Differentiated Programming Interfaces (CDPI) of TSO/E Version 2 REXX to define a new CICS specific REXX language processor environment for the user EXEC, and then invokes the EXEC. The REXX program also contains several REXX replaceable routines to handle all REXX storage requests, line-mode I/O and various other functions. On the very first invocation of the REXX program within a CICS region, a REXX system server is automatically started, under its own REXX environment control block. Thereafter, the REXX system server receives notification before the invocation and after the termination of each user EXEC invoked by the REXX program. The REXX system server is a shared server that all REXX user execs can route requests to, by ADDRESSing the subcommand environment SYSTEM. The GLOBALV global variable command support that is provided is an example of using the system server to add additional subcommands to REXX.

REXX EXEC invocation

- EXECs invoked from a terminal

  REXX EXECs are invoked by a CICS/REXX program named REXX. A CICS transaction id must be defined for this program. If the tran id is REXX then the name of the EXECs and its arguments follow on the command line. For example: REXX MYEXEC ABC will invoke the REXX EXEC MYEXEC and pass it the string ABC as an argument. If a transaction id other than REXX is associated with the REXX program, the name of the EXEC that is invoked is the same as the transaction id.

- EXECs invoked by a START command

  The REXX transaction associated with the REXX program may be invoked with the EXEC CICS START command. If start data is provided, that is passed to the EXEC as an argument. The name of the EXEC to invoke is normally expected to be provided in the start data.

- EXECs invoked by a LINK or XCTL

  The REXX program, when invoked by a LINK or XCTL, will attempt to find the name of the REXX EXEC to invoke in the COMMAREA, if one is available. The entire COMMAREA will also be passed to the EXEC as an argument.

Where EXECs execute

CICS/REXX EXECs are executed as part of the CICS task that invokes them, within the CICS region. The REXX interpreter is fully reentrant and runs above the 16 MB line (AMODE=31,RMODE=ANY).

How EXECs are located and loaded

The directories of specified REXX libraries are searched, in concatenation sequence in an attempt to locate an EXEC. If it is located, it is read into storage and control is given to the REXX interpreter to invoke it. Before REXX libraries are searched, there is first a check to see if the EXEC is already loaded in storage, and if so, since REXX EXECs are re-entrant, control is given immediately to the REXX interpreter.

How EXECs are edited

CICS/REXX includes a CICS-based text editor, wich is similar to the IBM XEDIT and ISPF/PDF editors, to edit EXECs and data files, directly under CICS.

Control of EXEC execution search order

A PATH subcommand is provided to control the search order of REXX File System directories. The directories specified in the PATH command are searched after the current directory (specified by the CD command).

REXX EXEC File System structure

Execs are currently stored as members a VSAM-based REXX file system. Some features of the REXX File System are: are:

- Hierarchical Directory structure (like OS/2, AIX, VM SFS)
- No need to register new users
- No need to register individual EXECs
- Basic support without an External Security Manager
- Import/Export to MVS Partitioned Datasets
- Management functions for members (COPY, DELETE, RENAME)
- FILELIST file directory interface utility
- An EXECIO-like I/O utility (FSIO)
- Supports insertion of records in middle of files
- Maximum records per member is approx. 2**32 minus 2
- Maximum record length is 2**32 minus 2
- Maximum VSAM datasets per a RFS filepool is 511
- Number of filepools is limited by system storage
- Execute-only support by library and by member
- Support for authorized REXX libraries (for authorized primitives)

Support of standard REXX features

SAY and TRACE statements

The REXX SAY and TRACE terminal I/O output statements use CICS Terminal Control Support to provide simulated line-mode output.

PULL and PARSE EXTERNAL statements

The REXX PULL and PARSE EXTERNAL terminal I/O input statements use CICS Terminal Control Support to provide simulated line-mode input.

REXX stack support

Same as TSO/E Version 2 REXX

REXX function support

CICS/REXX supports the same built-in function set as TSO/E Version 2 REXX with the following exceptions. The USERID function will return a 1 to 8 character CICS userid if the user is signed on, otherwise it will return blanks. The STORAGE function, which allows a REXX user to freely display and/or modify the virtual storage of the CICS region will be disabled or restricted.

REXX Function Packages

The function packages provided with TSO/E REXX that are not TSO specific, are provided and system administrators will have the ability to define/add additional function packages using standard documented interfaces.

REXX Subcommand Environment Support

REXX subcommand environments that are currently available (to use with the REXX ADDRESS command) are CICS, COMMAND, MVS and SYSTEM.

Invoking another EXEC as a subcommand

EXECs may be invoked as subcommands using the new client/server support (described later in this document).

Invoking CICS load modules as user provided subcommands

Support is provided for site provided subcommands, in the form of CICS LOAD modules (loaded using an EXEC CICS LINK) to be defined using the DEFCMD and DEFSCMD commands.

Adding REXX host subcommand environments

Support is provided to allow new CICS/REXX host subcommand environments to be added and supported in a variety of languages, including REXX. This is done using the DEFCMD and DEFSCMD subcommands, or by using the standard documented TSO/E REXX interfaces.

Support of standard CICS features/facilities

CICS mapped I/O support

Support is not yet available for CICS BMS I/O commands as REXX subcommands in the CICS subcommand environment.

Dataset I/O Services

Verbs for standard CICS dataset I/O services commands are planned as REXX subcommands.

Interfaces to CICS Facilities and Services

From within the ADDRESS CICS subcommand environment, support is planned for most popular CICS commands (as defined in the CICS Application Programmer's Reference Guide). Currently support is provided for the function provided by the following CICS Command Level commands:

- EXEC CICS SEND
- EXEC CICS SEND TEXT
- EXEC CICS RECEIVE
- EXEC CICS READQ TS

- EXEC CICS WRITEQ TS
- EXEC CICS DELETEQ TS
- EXEC CICS ASSIGN USERID
- EXEC CICS READ RRN
- EXEC CICS WRITE RRN
- EXEC CICS REWRITE RRN
- EXEC CICS DELETE RRN
- EXEC CICS UNLOCK
- EXEC CICS START
- EXEC CICS LINK
- EXEC CICS XCTL
- EXEC CICS SUSPEND

Invoking user applications from EXECs

EXEC CICS START, LINK and XCTL commands are currently supported.

REXX interfaces to CICS temporary & transient storage queues

Currently subcommand support exist for reading, writing and deleting CICS temporary storage queues from REXX.

Pseudo-conversational transaction support

CICS pseudo-conversational support for REXX EXECs is provided. If this support is enabled, an EXEC CICS RETURN TRANSID could is automatically issued before each CICS RECEIVE, the execution state of the EXEC preserved and the REXX transaction ended. The the next terminal I/O event would cause the REXX transaction to be re-invoked and the EXEC to be resumed at the next statement after the RECEIVE.

REXX EXEC Suspend/Resume support

CICS/REXX support includes a primitive (subcommand) to suspend the execution of the EXEC and causes the invoking transaction to end, allowing another transaction to run, attaching the terminal. The next time the REXX program is invoked, the suspended transaction will resume the suspended EXEC. Any start data passed is placed in the reserved REXX variable SDATA.

Interfaces to other programming languages

The goal is to provide interfaces to COBOL, C/370, Assembler, and maybe PL/I.

Security

Normal CICS interfaces to the MVS System Authorization Facility (SAF) will create the framework for CICS/REXX security. Advanced security needs for REXX subcommand and client/server security is expected to be provided under CICS/ESA using the EXEC CICS QUERY SECURITY command.

Performance discussion

Because of the production nature of CICS, much emphasis is being placed on performance. There are many design choices that can affect security. These include how REXX environments are defined, how the REXX file system structure is implemented, how security interfaces are implemented, how much virtual storage is given to an EXEC at invocation.

REXX is an excellent performer, especially for an interpreter, because it internally uses many sophisticated techniques, such as look-aside tables, for good performance. REXX has proven itself to be a reasonable performer in the VM arena as much of PROFS code is written in REXX. Many PROFS systems today support thousands of users in production. Another point to note, is that although REXX EXECs are interpreted, most of the actual processing for the typical application is spent executing REXX subcommands which do most of the actual work. These primitives can be and usually are written in a compiled language, when performance is an important consideration. Usually, for the majority of small to medium scale CICS applications the productivity benefits of using REXX far outweigh the performance penalty of using REXX. A similar analogy is customers using DB2 vs IMS. DB2 often requires more resources, but the benefits more than outweigh the added processing cost. The net result is that DB2 users are happy because they are more productive.

The best news from a performance perspective, is that the IBM REXX/370 compiler will work with CICS/REXX, whenever performance critical applications need it.

Miscellaneous features

A TERMID subcommand has been provided to return the four character terminal identifier of a CICS user.

A RETRIEVE PF key has been setup to retrieve the last input line enter using line-mode I/O.

Supported Environments and prerequisites

CICS/REXX currently runs under CICS/MVS and CICS/ESA. CICS/REXX requires that TSO/E V2.0 or later be installed and, if the REXX/370 compiler is used, in addition to the interpreter, then TSO/E V2.3.1 or later must be installed. Certain advanced functions, such as the planned REXX interface utilizing the programming interface of CICS 3.2 for Resource Definition Online, will only supported under CICS/ESA.

National language and DBCS support

The full range of DBCS functions and handling techniques that are included in TSO/E Version 2 REXX are available to the CICS/REXX user.

It is expected that the national languages supported for CICS/REXX will match those supported for TSO/E Version 2. Refer to announcement 288-694, dated December 6, 1988. The support for national languages will likely lag the initial American/English language support.

Building block S/W development - Common Interface Routine

One of the foundation architecture pieces of the CICS/REXX support code is a routine called the Common Interface Routine (CIR).

The purpose of this routine is to allow transparency and flexibility as to the implementation method and language of programs that make up software systems under CICS/REXX. That is, systems implementers should be free to create systems comprised of a mixture of traditional and client/server interpreted REXX EXECs, compiled REXX EXECs, COBOL, C and Assembler language programs. And they should be later free to change the language or implementation method of a program without affecting the correct functioning of the system as a whole.

This is accomplished by having REXX and all other programs that wish to participate in this system, to call the Common Interface Routine whenever control (or a client/server request) needs to be passed to another program. The CIR then determines from a table or data dictionary, the type and language of the target program, so it can invoke it (or pass the request to it) properly.

All programs that use the Common Interface Routine must use a consistent format for the passing of parameters (or requests) to the target and for the returning of any resulting data.

The use of the Common Interface Routine does not require the use of client/server computing, but is a closely related technique.

CICS/REXX Client/Server Architecture

High-level Client/Server support

A major new thrust of data processing is in the area of client/server processing. Many realize that this method of computing holds much promise for accomplishing their computing needs in a more responsive and cost-effective manner, especially in today's ever more increasingly work station based computing environments. However, many realize the promise and recognize the opportunity, but lack the tools to effectively accomplish their goals. The goal here is to augment the general CICS/REXX environment with a high-level, easy to use, REXX-based client/server processing support that will make it feasible for customers to easily implement client/server processing applications that they could have never before considered, better utilizing mainframe and workstation resources.

Client/Server Design goals

- Allow REXX servers to service multiple REXX clients, which may be located on a variety of remote systems (long-term)
- Provide an identity service to dynamically track and route requests and responses between servers and requestors on multiple systems by server name. It should support the concept of local and global resource management (long-term)
- Provide security interfaces to effectively and efficiently control authorization of access and communication between servers and requestors.
- Support both synchronous and asynchronous communication between servers and requestors
- Very high level, easy to use but flexible REXX interface to this server/requestor support
- Support parallel communication activities between a client and a server, at least separate command and data sockets/sessions (long term)
- Provide syncpoint and recovery capability
- Good performance through use of efficient techniques
- General enough in design to have a wide variety of uses

Current Client/Server Implementation

- Provides client/server support within a CICS region
- High-level REXX based
- Provides a common shared REXX system server
- Supports requests from both REXX and assembler clients
- Supports automatic server initiation

Requests are sent from a REXX client to a server as follows:

ADDRESS serverid 'request'

The server waits on and receives requests from a client by issuing the 'WAITREQ' subcommand. The server is suspended until a client request arrives (which is placed in the reserved REXX variable REQUEST).

There are subcommands available to REXX servers to get or replace the contents of client REXX variables, by name.

The security characteristics/authority level of a client are automatically inherited by the server while it is processing the request from that client.

CICS/REXX OfficeVision/MVS Environment Support

REXX EXECS for Application Integration

Currently OfficeVision/MVS provides the capability for the user to add new menu items or commands along with their associated CICS applications to their OfficeVision/MVS desktop. This is done using the Application Services component online administration utility to define new applications (represented by Application Type Descriptor (ATD) definitions).

Since REXX EXECs are invoked as a normal CICS program or transaction, REXX EXECs can easily be invoked from the OfficeVision/MVS desktop.

REXX EXECs under CICS/REXX are enabled to use the OfficeVision/MVS System Interface Block (SIB). The REXX program (or transaction) can be STARTed or XCTLed with a SIB passed to indicate what EXEC to invoke and also where to transfer control to when the EXEC has finished its processing. REXX EXECs can also pass an outbound SIB to OfficeVision/MVS or another SIB enabled application. This should greatly facilitate OfficeVision/MVS based Application Integration.

For security reasons, CICS/REXX will not allow a user EXEC to pass a SIB to OfficeVision/MVS unless that user is already signed on.

REXX EXECS as exits

It is planned to support the use of REXX EXECs as exit programs for OfficeVision/MVS and other CICS based applications. It is the exit implementer's responsibility to determine if a REXX exit would be suitable as an exit (for performance reasons, especially when an interpreted EXEC is used). However, it should be noted that REXX EXECs are successfully being used to code exits routines for production applications running under VM/CMS.

CICS/REXX Interfaces to other products

One of the strengths of REXX is the ease with which high-level interfaces to other products can be provided. It seems a logical next step to add interfaces from CICS REXX to DB2, GDDM and other products, on an as needed basis.

Description of interface to DB2

This interface would be similar to the REXX to SQL interface available under VM but would use the CICS dynamic SQL interface to DB2.

Description of interface to GDDM

This interface would function essentially the same as the existing GDDM/REXX product under VM.

CICS/REXX CMS Environment Compatibility/Emulation

To facilitate the migrating of systems and the porting of software from VM/CMS to MVS CICS, the following VM/CMS capabilities are provided:

- Global variable support compatible with the VM/CMS GLOBALV command has been provided.
- Full-screen terminal I/O support, compatible with the VM/CMS WAITREAD command has been provided.
- EXECIO command is supported for I/O to sequential datasets
- Xedit editor limited compatibility

Summary

Prototype development experience

My prototype development experience has led me to the conclusion that it is feasible to do a good implementation of REXX under CICS. However what will do more to guarantee a good implementation of REXX under CICS, more than anything else, is the feedback, input and participation of IBM customers in this effort.

Much more than just another language for CICS

I hope that by now you have come to the conclusion that CICS/REXX is much more than just another CICS language. That it is rather the beginning of a new environment with the potential to dramatically improve the way that we work.

Appendix - Sample CICS/REXX screens

Sample FILELIST screen

```
USER=WRISTON  DIRECTORY=/
CMD   FILENAME FILETYPE ATTRIBUTES RECORDS BLOCKS DATE     TIME
 • • • Top Of File • • •
      TEST2    EXEC     FILE       5       1      03/27/92 10:31:04
      TEST1    EXEC     FILE       11      1      03/27/92 10:30:29
      GENID    EXEC     FILE       7       1      03/13/92 09:00:37
      SECURITY EXEC     FILE       21      1      03/13/92 08:59:31
      TEST     EXEC     FILE       14      1      03/11/92 15:06:53
      FSIO     LIB      FILE       493     3      03/11/92 08:42:04
      WINDOWS  EXEC     FILE       50      0      03/10/92 10:46:19
      KEDIT    EXEC     FILE       1278    5      03/10/92 08:49:14
      USERS              DIR       1       1      03/10/92 08:49:10
 • • • End Of File • • •









COMMAND ===>
```

Sample KEDIT Screen

```
WRISTON  /USERS/WRISTON NONAME SIZE=0 LINE=0 CHANGED=NO
K E D I T  1.1.9 -  CICS Editor









00000 *•*•*•Top Of File *•*•*•
00001 *•*•*•End Of File *•*•*•









COMMAND ===>
```

DEMO EXEC

Source listing

```
EDIT ---- SHRIVER.REXX(DEMO) - 01.08 ------------------------ COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
****** ***************************** TOP OF DATA ******************************
000100 /* REXX */
000200 TRACE 'o'
000201 arg parms
000202 parse source . . . . . environm
000203
000204 say '***-----------------------------------------------------***'
000205 SAY '*** This is a test REXX program running under' environm '***'
000206 say '***-----------------------------------------------------***'
000207 say
000208 say 'The arguments passed were:' parms
000209 say
000210
000211 /* example of REXX standard line-mode input */
000212 say 'What is your name?'
000213 parse pull name
000214 say
000215 say 'Welcome to' environm 'REXX,' name
000216 say
000217
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(DEMO) - 01.08 ------------------------ COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
000218 /* example of nesting */
000219 address mvs
000220 'demo2 xxx'
000221
000222 /* example of CICS subcommands */
000223 address cics
000224 'TERMID'      /* get my CICS terminal id */
000225 outbuf = sba(22 12)||'This is fullscreen output to terminal' termid
000226
000227 /* perform CICS fullscreen output */
000228 'SEND' outbuf /* do a CICS EXEC CICS SEND */
000229 outbuf = sba(23 12) || 'Now try some fullscreen input'
000230 'SEND' outbuf
000231
000232 /* perform CICS fullscreen input */
000233 'WAITREAD'    /* do an EXEC CICS RECEIVE and parse into vars */
000234 say 'The AID key that was pressed =' waitread.1
000235 say 'The cursor was at (Row Col):' subword(Waitread.2,2,2)
000236 say 'The data that was entered (Row Col Data):' subword(waitread.3,2)
000237 say
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(DEMO) - 01.08 ------------------------ COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
000238
000239 /* example of using the system server */
000240 say 'send a GLOBALV SET and GET commands to the system server'
000241 address system
000242 'GLOBALV SELECT GROUP1 SET VAR1 test data'
000243 'GLOBALV SELECT GROUP1 GET VAR1'
000244 say 'The contents of VAR1 =' var1
000245 say
000246
000247 trace 'o'  /* don't want to trace large loop */
000248 /* example of stand REXX line-mode output with more than 1 screen */
000249 do i = 1 to 20
000250   do j = 1 to 1000
000251     a = 5
000252   end
000253   say i*1000 'assignment statements have been executed'
000254 end
000255
000256 say
000257 /* show that built-in REXX functions are available */
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(DEMO) - 01.08 ------------------------ COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
000258 say "Today's date is" date('w') date()
000260 say 'The time is' time()
000400 EXIT
****** ***************************** BOTTOM OF DATA ******************************
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

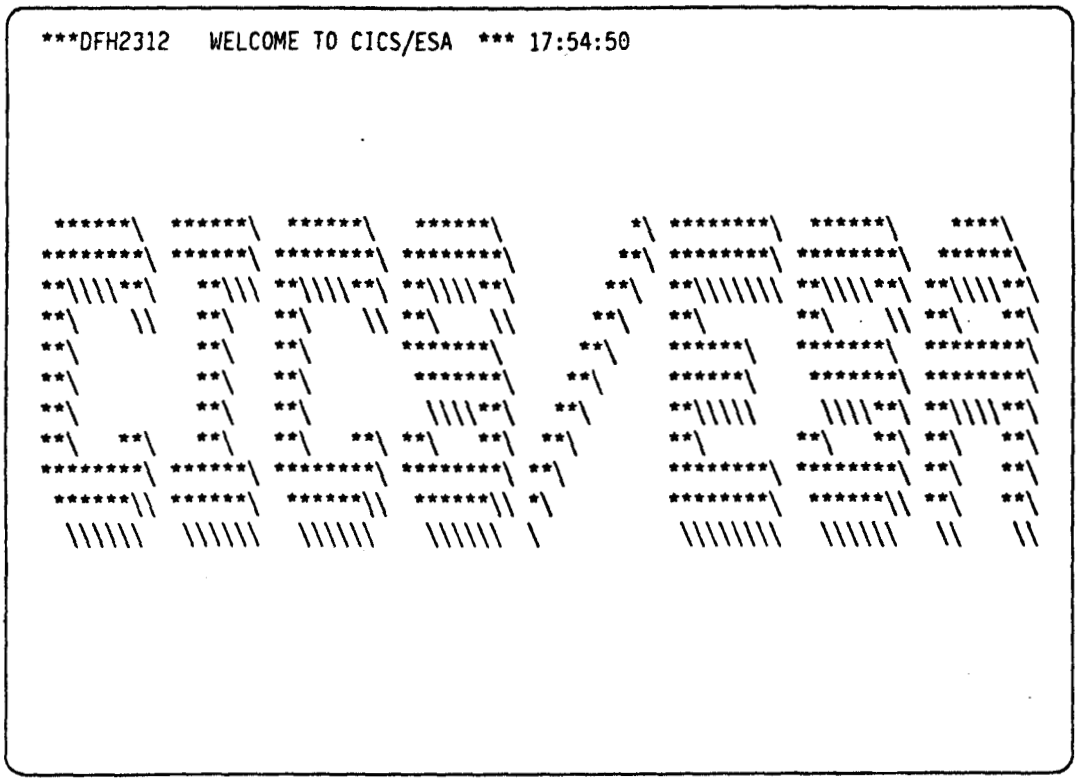
```
EDIT ---- SHRIVER.REXX(DEMO2) - 01.03 ---------------------- COLUMNS 001 072
COMMAND ===>                                                 SCROLL ===> PAGE
****** ***************************** TOP OF DATA ********************************
000001 /* nest level 2 */
000010 trace 'r'
000100 say 'you entered demo2 exec'
000110 address mvs
000200 'demo3 yyy'
000300 exit
****** **************************** BOTTOM OF DATA ******************************



F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

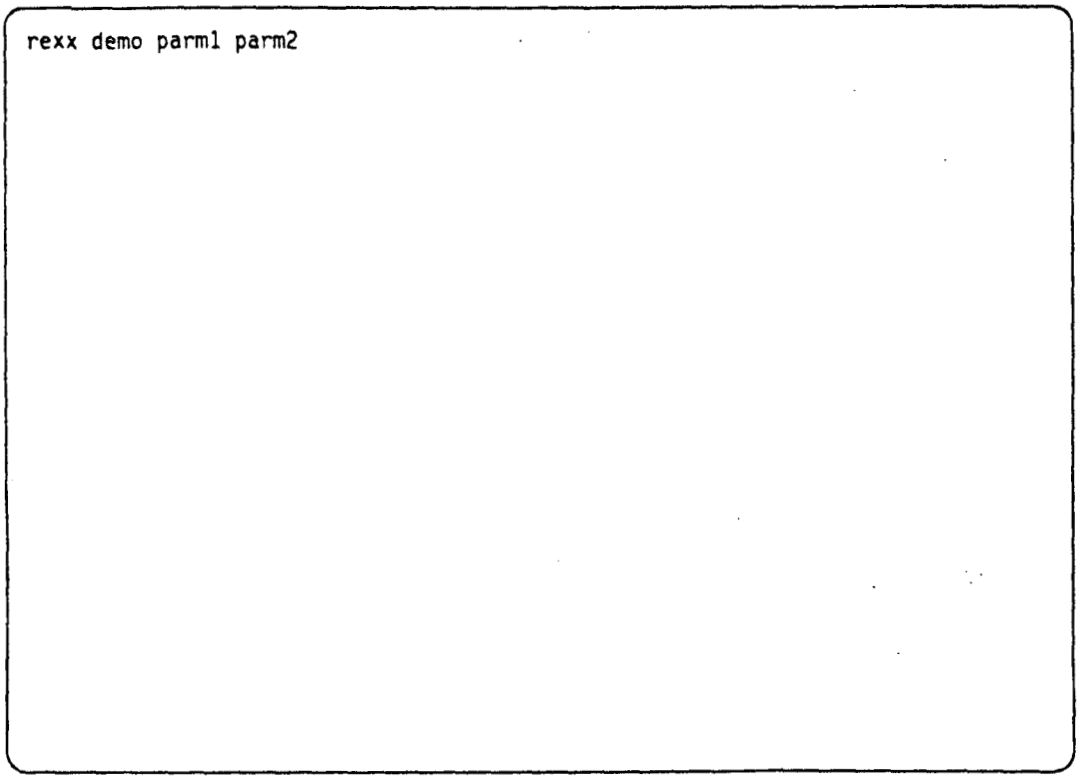
```
EDIT ---- SHRIVER.REXX(DEMO3) - 01.03 ---------------------- COLUMNS 001 072
COMMAND ===>                                                 SCROLL ===> PAGE
****** ***************************** TOP OF DATA ********************************
000010 /* next level 3 */
000020 address mvs
000100 say 'you entered demo3 exec'
000200 'demo4'
****** **************************** BOTTOM OF DATA ******************************



F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(DEMO4) - 01.03 ----------------------- COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
****** ***************************** TOP OF DATA ******************************
000010 /* nest level 4 */
000100 say 'you entered demo4 exec'
000110 address mvs 'demo5'
****** **************************** BOTTOM OF DATA ****************************

F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(DEMO5) - 01.00 ----------------------- COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
****** ***************************** TOP OF DATA ******************************
000200 /* rexx */
000300 say 'you entered demo5 exec'
****** **************************** BOTTOM OF DATA ****************************

F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

Execution with trace off


***DFH2312 WELCOME TO CICS/ESA *** 17:54:50


rexx demo parm1 parm2

```
***-----------------------------------------------------***
*** This is a test REXX program running under MVS CICS ***
***-----------------------------------------------------***

The arguments passed were: PARM1 PARM2

What is your name?









Dave                                                  READ
```

```
***-----------------------------------------------------***
*** This is a test REXX program running under MVS CICS ***
***-----------------------------------------------------***

The arguments passed were: PARM1 PARM2

What is your name?
Dave

Welcome to MVS CICS REXX, Dave

     3 *-* say 'you entered demo2 exec'
       >>>   "you entered demo2 exec"
you entered demo2 exec
     4 *-* address mvs
     5 *-* 'demo3 yyy'
       >>>   "demo3 yyy"
you entered demo3 exec
you entered demo4 exec
you entered demo5 exec
     6 *-* exit
          This is fullscreen output to terminal 04G1
          Now try some fullscreen input
```

```
***-----------------------------------------------------***
*** This is a test REXX program running under MVS CICS ***
***-----------------------------------------------------***

The arguments passed were: PARM1 PARM2

What is your name?
Dave

Welcome to MVS CICS REXX, Dave

     3 *-* say 'you entered demo2 exec'
       >>>   "you entered demo2 exec"
you entered demo2 exec
     4 *-* address mvs
     5 *-* 'demo3 yyy'
       >>>   "demo3 yyy"
you entered demo3 exec
you entered demo4 exec
you entered demo5 exec
     6 *-* exit
The AID key that was pressed = ENTER
         Now try some fullscreen input
test input                                                    MORE
```

```
The cursor was at (Row Col): 24 16
The data that was entered (Row Col Data): 24 2 test input

send a GLOBALV SET and GET commands to the system server
The contents of VAR1 = test data

1000 assignment statements have been executed
2000 assignment statements have been executed
3000 assignment statements have been executed
4000 assignment statements have been executed
5000 assignment statements have been executed
6000 assignment statements have been executed
7000 assignment statements have been executed
8000 assignment statements have been executed
9000 assignment statements have been executed
10000 assignment statements have been executed
11000 assignment statements have been executed
12000 assignment statements have been executed
13000 assignment statements have been executed
14000 assignment statements have been executed
15000 assignment statements have been executed
16000 assignment statements have been executed

                                                              MORE
```

```
17000 assignment statements have been executed
18000 assignment statements have been executed
19000 assignment statements have been executed
20000 assignment statements have been executed

Today's date is Tuesday 20 Aug 1991
The time is 17:59:31
Ready; (5.232298)
```

Execution with trace on

```
rexx demo parm1 parm2
```

```
     3 *-* arg parms
       >>>   "PARM1 PARM2"
     4 *-* parse source . . . . . environm
       >.>   "TSO"
       >.>   "COMMAND"
       >.>   "DEMO"
       >.>   "SYSEXEC"
       >.>   "?"
       >.>   "DEMO"
       >.>   "CICS"
       >>>   "MVS CICS"
     6 *-* say '***-------------------------------------------------------***'
       >>>   "****-----------------------------------------------------****
***-----------------------------------------------------***
     7 *-* SAY '*** This is a test REXX program running under' environm '***'
       >>>   "**** This is a test REXX program running under MVS CICS ****
*** This is a test REXX program running under MVS CICS ***
     8 *-* say '***-------------------------------------------------------***'
       >>>   "****-----------------------------------------------------****
***-----------------------------------------------------***
     9 *-* say


                                                                      MORE
```

```
    10 *-* say 'The arguments passed were:' parms
       >>>   "The arguments passed were: PARM1 PARM2"
The arguments passed were: PARM1 PARM2
    11 *-* say

    13 *-* /* example of REXX standard line-mode input */
    14 *-* say 'What is your name?'
       >>>   "What is your name?"
What is your name?
    15 *-* parse pull name









                                                                      READ
```

```
    10 *-* say 'The arguments passed were:' parms
       >>>   "The arguments passed were: PARM1 PARM2"
The arguments passed were: PARM1 PARM2
    11 *-* say

    13 *-* /* example of REXX standard line-mode input */
    14 *-* say 'What is your name?'
       >>>   "What is your name?"
What is your name?
    15 *-* parse pull name
```

David Shriver READ

```
    10 *-* say 'The arguments passed were:' parms
       >>>   "The arguments passed were: PARM1 PARM2"
The arguments passed were: PARM1 PARM2
    11 *-* say

    13 *-* /* example of REXX standard line-mode input */
    14 *-* say 'What is your name?'
       >>>   "What is your name?"
What is your name?
    15 *-* parse pull name
David Shriver
       >>>   "David Shriver"
    16 *-* say

    17 *-* say 'Welcome to' environm 'REXX,' name
       >>>   "Welcome to MVS CICS REXX, David Shriver"
Welcome to MVS CICS REXX, David Shriver
    18 *-* say

    20 *-* /* example of nesting */
    21 *-* address mvs
    22 *-* 'demo2 xxx'
```

MORE

```
      >>>   "demo2 xxx"
    3 *-* say 'you entered demo2 exec'
      >>>   "you entered demo2 exec"
you entered demo2 exec
    4 *-* address mvs
    5 *-* 'demo3 yyy'
      >>>   "demo3 yyy"
you entered demo3 exec
you entered demo4 exec
you entered demo5 exec
    6 *-* exit
   24 *-* /* example of CICS subcommands */
   25 *-* address cics
   26 *-* 'TERMID'      /* get my CICS terminal id */
      >>>   "TERMID"
   27 *-* outbuf = sba(22 12)||'This is fullscreen output to terminal' termid
      >>>   "?!$This is fullscreen output to terminal 04G1"
   29 *-* /* perform CICS fullscreen output */
   30 *-* 'SEND' outbuf /* do a CICS EXEC CICS SEND */
      >>>   "SEND ?!$This is fullscreen output to terminal 04G1"
   31 *-* outbuf = sba(23 12) || 'Now try some fullscreen input'
      >>>   "?$,Now try some fullscreen input"

                                                                      MORE
```

```
   32 *-* 'SEND' outbuf
      >>>   "SEND ?$,Now try some fullscreen input"
   34 *-* /* perform CICS fullscreen input */
   35 *-* 'WAITREAD'    /* do an EXEC CICS RECEIVE and parse into vars */
      >>>   "WAITREAD"









         Now try some fullscreen input
```

```
    32 *-* 'SEND' outbuf
       >>>   "SEND ?$,Now try some fullscreen input"
    34 *-* /* perform CICS fullscreen input */
    35 *-* 'WAITREAD'    /* do an EXEC CICS RECEIVE and parse into vars */
       >>>   "WAITREAD"
```

```
         Now try some fullscreen input
test input
```

```
    32 *-* 'SEND' outbuf
       >>>   "SEND ?$,Now try some fullscreen input"
    34 *-* /* perform CICS fullscreen input */
    35 *-* 'WAITREAD'    /* do an EXEC CICS RECEIVE and parse into vars */
       >>>   "WAITREAD"
    36 *-* say 'The AID key that was pressed =' waitread.1
       >>>   "The AID key that was pressed = ENTER   "
The AID key that was pressed = ENTER
    37 *-* say 'The cursor was at (Row Col):' subword(Waitread.2,2,2)
       >>>   "The cursor was at (Row Col): 24 12"
The cursor was at (Row Col): 24 12
    38 *-* say 'The data that was entered (Row Col Data):' subword(waitread.3,2
       >>>   "The data that was entered (Row Col Data): 24 2 test input"
The data that was entered (Row Col Data): 24 2 test input
    39 *-* say

    41 *-* /* example of using the system server */
    42 *-* say 'send a GLOBALV SET and GET commands to the system server'
       >>>   "send a GLOBALV SET and GET commands to the system server"
send a GLOBALV SET and GET commands to the system server
    43 *-* address system
    44 *-* 'GLOBALV SELECT GROUP1 SET VAR1 test data'
         Now try some fullscreen input
test input                                                          MORE
```

```
      >>>   "GLOBALV SELECT GROUP1 SET VAR1 test data"
   45 *-* 'GLOBALV SELECT GROUP1 GET VAR1'
      >>>   "GLOBALV SELECT GROUP1 GET VAR1"
   46 *-* say 'The contents of VAR1 =' var1
      >>>   "The contents of VAR1 = test data"
The contents of VAR1 = test data
   47 *-* say

   49 *-* trace 'o'  /* don't want to trace large loop */
1000 assignment statements have been executed
2000 assignment statements have been executed
3000 assignment statements have been executed
4000 assignment statements have been executed
5000 assignment statements have been executed
6000 assignment statements have been executed
7000 assignment statements have been executed
8000 assignment statements have been executed
9000 assignment statements have been executed
10000 assignment statements have been executed
11000 assignment statements have been executed
12000 assignment statements have been executed
13000 assignment statements have been executed

                                                                    MORE
```

```
14000 assignment statements have been executed
15000 assignment statements have been executed
16000 assignment statements have been executed
17000 assignment statements have been executed
18000 assignment statements have been executed
19000 assignment statements have been executed
20000 assignment statements have been executed

Today's date is Tuesday 20 Aug 1991
The time is 18:02:03
Ready; (9.924010)
```

REX EXEC

Source listing

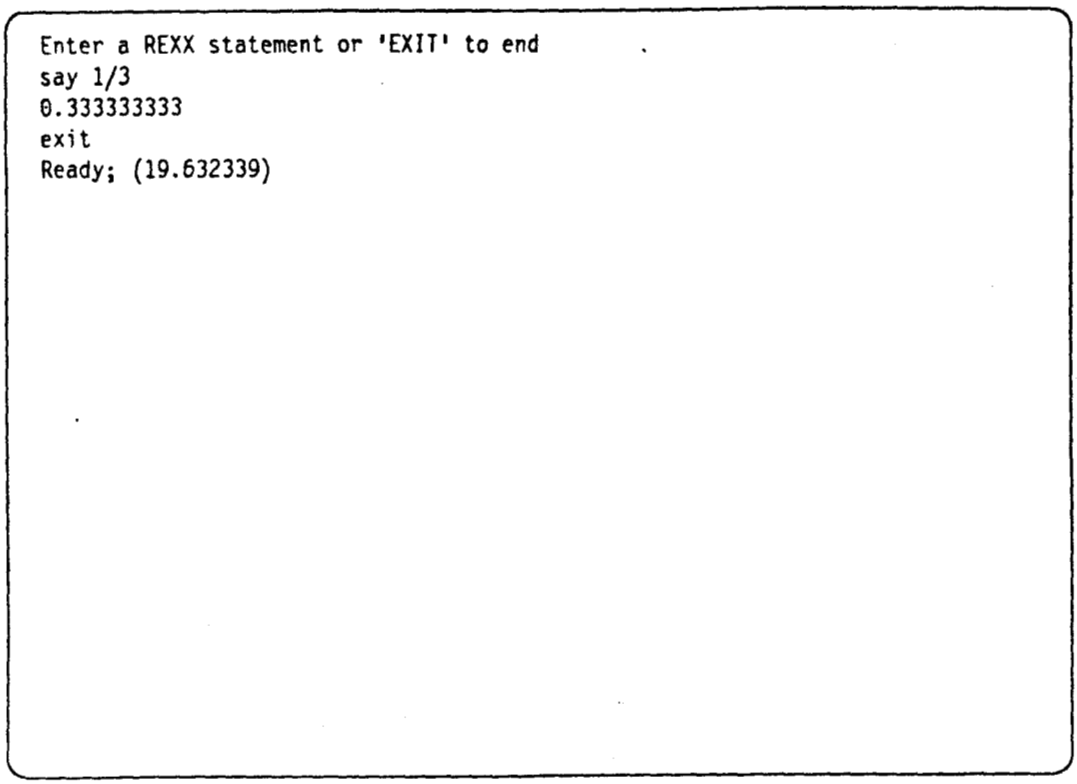
```
EDIT ---- SHRIVER.REXX(REX) - 01.08 ------------------------ MEMBER REX SAVED
COMMAND ===>                                                  SCROLL ===> PAGE
****** ***************************** TOP OF DATA ******************************
000001 /* interpretive execution of REXX statements from the terminal */
000002 TRACE 'O'
000003 parse arg arg
000004 signal on error
000005 signal on syntax
000006 SAY "Enter a REXX statement or 'EXIT' to end"
000007 restart:
000008 DO FOREVER
000009   parse external input
000010   if input = '' then SAY "Enter a REXX statement or 'EXIT' to end"
000011   INTERPRET input
000012   if substr(input,1,1) = "'" then say 'RC =' rc';'
000013 END
000014 EXIT
000015 error:
000016       say 'RC =' rc
000017       signal on error
000018       signal restart
000019 syntax:
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

```
EDIT ---- SHRIVER.REXX(REX) - 01.08 ------------------------ COLUMNS 001 072
COMMAND ===>                                                  SCROLL ===> PAGE
000020       Say 'Syntax error ----  re-enter'
000021       signal on syntax
000022       signal restart
****** ***************************** BOTTOM OF DATA ***************************
F13=HELP     F14=SPLIT    F15=END      F16=RETURN   F17=RFIND    F18=RCHANGE
F19=UP       F20=DOWN     F21=SWAP     F22=LEFT     F23=RIGHT    F24=RETRIEVE
```

Execution

```
rex
```

```
Enter a REXX statement or 'EXIT' to end
say 1/3
0.333333333


exit                                                          READ
```


Enter a REXX statement or 'EXIT' to end
say 1/3
0.333333333
exit
Ready; (19.632339)

REXX: TECHNICAL ISSUES, TODAY AND TOMORROW

MICHAEL SINZ
COMMODORE

REXX

Technical Issues

Today and Tomorrow

Michael Sinz
Senior Systems Engineer
Commodore International - Technology Group

Today

The Good

- ## REXX is a computer language

  REXX is a easy language to learn do to the non-typed, non-declared nature of the language. MFC did a very good job in thinking about what the user of REXX needed rather than how languages are normally written.

- ## REXX is becoming a standard

  The X3J18 group is currently working on a draft ANSI standard for REXX.

- ## REXX is available across platforms

  REXX is now a standard part of a number of operating systems and is available in flavors for most others.

- ## REXX is part of solutions

  REXX is now seen as a standard tool in environments where REXX is installed. It has not only become part of the environment but has proven itself to be very useful. A great many example of this can be seen in the Amiga environment, where REXX has become the tool of choice for systems integration by VARs in many vertical markets.

Today

The Good

- ## REXX is very flexible

  Due to the design of REXX, it has turned out to be very flexible in adapting to more complex systems. For example, on the Amiga, REXX can communicate with any number of applications that have support for REXX. This makes it possible for users and systems integrators to pull together very powerful tools into what looks and acts like one very customized application. This makes the migration into vertical markets much easier and reduces the turn-around time to meet the demands of the changing markets.

- ## REXX has many good points

  After all, it took me two pages just to skim over the key points...

Today

The Bad

- ## REXX is a computer language

  While REXX is a easy language to learn, it is still a "computer language" and that is keeping some people from using it. Many users would easily be able to use REXX for "programming" if it did not feel like programming. A good example of this is the Lotus 1-2-3 macros which business people used all the time but did not realized that they were programming. (And if told it was programming, they suddenly stopped)

- ## REXX is a not up to date

  While REXX has many good points, it is currently not up to the task of some of the issues in today's computing environments. It is not so much that REXX can not be since any implementor of the language can choose to extend it in some ways; rather it is a problem of choosing a model that fits into the REXX model as well as addressing the requirements of complex multi-tasking, multi-user, multi-processor, networked, graphical, object oriented environments. (What a mouth full)

- ## REXX is not yet a standard

  While X3J18 is working hard on getting the standard done, it is not done yet and the various implementations of REXX are not fully interchangeable.

- ## REXX support in applications

  This will happen more as the market starts to demand it and as the utility of REXX becomes a major feature in products. A good example of this happening already is in the Amiga computer where productivity applications are almost required to support REXX due to public demand and feature requirements.

Today

The Ugly

- REXX is *NEVER* ugly...
- Well...
- ...almost never....

The implementation of a good REXX on many platforms is not as simple as the language seems. Part of this is due to the specification of the language and part of it is due to the way REXX is designed to interact with the operating environment of the system. Hopefully the specification of the language will help out, but the close interaction with the system will always be there for the developer to deal with. In addition, without work at getting REXX into new computing technologies such as GUIs, it can be rather "ugly" to code in REXX for such environments.

Tomorrow

- ## REXX and the future

In order for REXX to grow, the direction of the growth needs to be identified first. If the goal is to make REXX into the "user's" programming language, it is important that that goal is what drives development of the language.

- ## Multi-Tasking, Multi-User, & Networks

The current REXX model works great in simple environments. The fact that I/O is very simplistic make it easier for users to learn and use. However, this has also made a number of things rather difficult (if not impossible) to do in complex environments. Issues such as synchronization, semaphores, and shared access are all currently outside of the REXX model. While it would be simple to just use the models of other computer languages, it would be counter to the main goal of REXX: simplicity for the user. This means that a new model for such things as file locking, access control, and synchronization will be needed.

- ## Graphical User Interfaces

The world is moving into GUI environments. The reason for the growth of this interface model is due partly to the fact that computers are more powerful and that users find GUIs easier to learn and use. REXX, as a language, does not address any of these issue directly. External function libraries exist for a number of different GUIs but not having the language contain some fundamental support for GUI operation makes life more difficult for the person writing the REXX program that deals with the GUI. Research at a number of places, most notably IBM, have shown how REXX can be gracefully enhanced to gain these features. However, the amount of work involved for the implementor of the language processor is high.

Tomorrow

- ## REXX and Objects

As operating environments become more object oriented, REXX will need to learn about objects in order to fit in with the environments it is operating in. Last year, IBM showed some of their ideas on how this could be done. Work such as that will need to continue and will need to become standardized such that REXX continues to be a cross-platform language.

- ## REXX as a visual language

This is one of my goals for REXX. REXX has become a user's language. However, it is still very much like a computer language. With the Amiga (and soon to be the many OS/2 2.0 users) REXX has become a staple of application features. On the Amiga, over 140 REXX supporting applications are available with every new application having REXX support due to user demands. REXX has become both a systems integrators best friend and the advanced users power-tool. The next step would be to give this power to users who do not "program" a computer in the traditional sense. A visual interface to REXX programming that can be mastered by the business man and home computer user would be the ultimate goal. In a mature, REXX supporting platform, such a tool would give more users the power to combine their creativity along with the applications they have bought to produce something that is "what they want." Such a tool does not have to replace REXX but would just have to be able to sit on top of REXX. However, such a tool would require more standardization of the way applications support REXX and of the REXX language itself. (I am assuming that due to the complexity of such a tool that it would be "ported" to all the platforms that support REXX in such a way.)

- ## REXX in the future...

With the current growth of REXX as a user's tool and its inclusion as a standard part of a number of operating environments, the future for REXX looks bright. (And REXX developers can be assured of a number of tough problems that will need to be addressed.)

REXX

Going Strong Into the Future.

UNI-REXX: REXX FOR THE UNIX & VMS ENVIRONMENTS

3RD ANNUAL REXX SYMPOSIUM
ANNAPOLIS, MARYLAND
MAY 5, 1992

ED SPIRE
THE WORKSTATION GROUP
ROSEMONT, ILLINOIS

PRESENTATION OUTLINE

PART 1. "MARKETING" REXX FOR UNIX

- THE UNIX MARKET
- MIS AND UNIX
- MACRO LANGUAGES FOR UNIX
- OUR EXPERIENCE SO FAR
- PERL
- UNI-REXX
- RECENT WORK ON UNI-REXX
- PLANS TO IMPROVE ACCEPTANCE
- OTHER PLANS

PART 2. OTHER REXX RELATED ITEMS

- COMBINED REXX AND C DEVELOPMENT PROJECT
- RELATIONSHIP TO PUBLIC DOMAIN REXX'S

"Marketing" REXX

...Not in the "grubby" sense of "hawking" REXX

...But in the sense of "making REXX valuable", through

- improvement,
- application,
- education,
- etc.

The Unix Community

Originally academic and engineering oriented usage

First commercialization involved small business applications (i.e., XENIX)

RISC Price/performance improvements are attracting traditional commercial MIS applications

Lately, the Intel price/performance curve is approaching that of the low end RISC systems, further accelerating wider interest in Unix.

Traditional commercial MIS organizations and Unix

RISC price/performance has become unavoidably attractive

Unix is the only currently available OS for these platforms. (OS/2 and Windows/NT will change this situation)

Unix may continue to be the only truly portable environment for those who seek the flexibility of open systems. (OS/2 and Windows/NT are proprietary technology)

The Unix learning curve is large for existing commercial MIS staffers

These factors are the basis for TWG's product line of mainframe utility software for Unix.

Macro Languages for Unix

Unix includes many "standard" utilities:

- A large number of commands that provide information, access, and control at a very low level
- Reusable "filters" used via "pipes"
- I/O redirection

...making for a very flexible (albeit daunting) environment.

Given this level of complexity, there is certainly a need for macro facilities. As an example, take the case of routine disk space space management at the end-user level.

Standard Unix really only provides the 'du' command, which provides very low level data...

(Output of Unix command "du /usr/export/home/ets" follows...)

```
1668 /usr/export/home/ets/pdr/rexx1
56 /usr/export/home/ets/pdr/rexx2/trip
5 /usr/export/home/ets/pdr/rexx2/code
1316 /usr/export/home/ets/pdr/rexx2
3018 /usr/export/home/ets/pdr
197 /usr/export/home/ets/util
5 /usr/export/home/ets/localterm/a
2 /usr/export/home/ets/localterm/d
9 /usr/export/home/ets/localterm/h
69 /usr/export/home/ets/localterm/i
5 /usr/export/home/ets/localterm/j
3 /usr/export/home/ets/localterm/s
2 /usr/export/home/ets/localterm/u
5 /usr/export/home/ets/localterm/v
13 /usr/export/home/ets/localterm/w
114 /usr/export/home/ets/localterm
1 /usr/export/home/ets/.wastebasket
432 /usr/export/home/ets/rexx
42 /usr/export/home/ets/xedit
24 /usr/export/home/ets/terminfo
3 /usr/export/home/ets/rxf
133 /usr/export/home/ets/lang/SC1.0/man/man1
398 /usr/export/home/ets/lang/SC1.0/man/man3
4 /usr/export/home/ets/lang/SC1.0/man/man5
541 /usr/export/home/ets/lang/SC1.0/man
9 /usr/export/home/ets/lang/SC1.0/include/CC/arpa
9 /usr/export/home/ets/lang/SC1.0/include/CC/debug
14 /usr/export/home/ets/lang/SC1.0/include/CC/hsfs
3 /usr/export/home/ets/lang/SC1.0/include/CC/lofs
12 /usr/export/home/ets/lang/SC1.0/include/CC/lwp
29 /usr/export/home/ets/lang/SC1.0/include/CC/mon
29 /usr/export/home/ets/lang/SC1.0/include/CC/net
```

```
59 /usr/export/home/ets/lang/SC1.0/include/CC/netinet
9 /usr/export/home/ets/lang/SC1.0/include/CC/nettli
15 /usr/export/home/ets/lang/SC1.0/include/CC/nfs
158 /usr/export/home/ets/lang/SC1.0/include/CC/pixrect
13 /usr/export/home/ets/lang/SC1.0/include/CC/protocols
28 /usr/export/home/ets/lang/SC1.0/include/CC/rfs
56 /usr/export/home/ets/lang/SC1.0/include/CC/rpc
58 /usr/export/home/ets/lang/SC1.0/include/CC/rpcsvc
14 /usr/export/home/ets/lang/SC1.0/include/CC/sbusdev
20 /usr/export/home/ets/lang/SC1.0/include/CC/sun
9 /usr/export/home/ets/lang/SC1.0/include/CC/sun3
3 /usr/export/home/ets/lang/SC1.0/include/CC/sun3x
3 /usr/export/home/ets/lang/SC1.0/include/CC/sun4
3 /usr/export/home/ets/lang/SC1.0/include/CC/sun4c
59 /usr/export/home/ets/lang/SC1.0/include/CC/sundev
153 /usr/export/home/ets/lang/SC1.0/include/CC/suntool
142 /usr/export/home/ets/lang/SC1.0/include/CC/sunwindow
210 /usr/export/home/ets/lang/SC1.0/include/CC/sys
3 /usr/export/home/ets/lang/SC1.0/include/CC/tfs
6 /usr/export/home/ets/lang/SC1.0/include/CC/ufs
3 /usr/export/home/ets/lang/SC1.0/include/CC/vm
1430 /usr/export/home/ets/lang/SC1.0/include/CC
1436 /usr/export/home/ets/lang/SC1.0/include
1001 /usr/export/home/ets/lang/SC1.0/cg87
1003 /usr/export/home/ets/lang/SC1.0/cg89
12 /usr/export/home/ets/lang/SC1.0/misalign
52 /usr/export/home/ets/lang/SC1.0/README
9793 /usr/export/home/ets/lang/SC1.0
133 /usr/export/home/ets/lang/man/man1
398 /usr/export/home/ets/lang/man/man3
4 /usr/export/home/ets/lang/man/man5
541 /usr/export/home/ets/lang/man
12537 /usr/export/home/ets/lang
219 /usr/export/home/ets/doc
49 /usr/export/home/ets/symp
16725 /usr/export/home/ets
```

A TYPICAL MACRO PROGRAM WOULD PROVIDE THIS LOW LEVEL INFORMATION IN A MORE MANAGEABLE FORM...

(OUTPUT OF UNIX COMMAND "SPACE /USR/EXPORT/HOME/ETS" FOLLOWS...)

```
FILESYSTEM              KBYTES    USED   AVAIL CAPACITY  MOUNTED ON
/DEV/SD0G               186414  141068   26704    84%    /USR
NUM SIZE(K) % NODE    %FILESYS  ITEM
  1   3018  18.041    2.139   /USR/EXPORT/HOME/ETS/PDR
  2    197   1.178    0.140   /USR/EXPORT/HOME/ETS/UTIL
  3    114   0.681    0.081   /USR/EXPORT/HOME/ETS/LOCALTERM
  4      1   0.006    0.001   /USR/EXPORT/HOME/ETS/.WASTEBASKET
  5    432   2.582    0.306   /USR/EXPORT/HOME/ETS/REXX
  6     42   0.251    0.030   /USR/EXPORT/HOME/ETS/XEDIT
  7     24   0.143    0.017   /USR/EXPORT/HOME/ETS/TERMINFO
  8      3   0.018    0.002   /USR/EXPORT/HOME/ETS/RXF
  9  12537  74.942    8.887   /USR/EXPORT/HOME/ETS/LANG
 10    219   1.309    0.155   /USR/EXPORT/HOME/ETS/DOC
 11     53   0.317    0.038   /USR/EXPORT/HOME/ETS/SYMP
 12  16729 100.000   11.859   /USR/EXPORT/HOME/ETS
SELECT NODE NUMBER FOR REDISPLAY (* FOR SAME) AND OPTIONAL LEVEL
OR 'X' TO EXIT

1

FILESYSTEM              KBYTES    USED   AVAIL CAPACITY  MOUNTED ON
/DEV/SD0G               186414  141068   26704    84%    /USR
NUM SIZE(K) % NODE    %FILESYS  ITEM
  1   1668  55.268    1.182   /USR/EXPORT/HOME/ETS/PDR/REXX1
  2   1316  43.605    0.933   /USR/EXPORT/HOME/ETS/PDR/REXX2
  3   3018 100.000    2.139   /USR/EXPORT/HOME/ETS/PDR
SELECT NODE NUMBER FOR REDISPLAY (* FOR SAME) AND OPTIONAL LEVEL
OR '0' TO GO BACK UP ONE NODE
OR 'X' TO EXIT

0
```

```
FILESYSTEM              KBYTES    USED   AVAIL CAPACITY  MOUNTED ON
/DEV/SD0G               186414  141069   26703    84%    /USR
NUM SIZE(K) % NODE    %FILESYS  ITEM
  1   3018  18.041    2.139   /USR/EXPORT/HOME/ETS/PDR
  2    197   1.178    0.140   /USR/EXPORT/HOME/ETS/UTIL
  3    114   0.681    0.081   /USR/EXPORT/HOME/ETS/LOCALTERM
  4      1   0.006    0.001   /USR/EXPORT/HOME/ETS/.WASTEBASKET
  5    432   2.582    0.306   /USR/EXPORT/HOME/ETS/REXX
  6     42   0.251    0.030   /USR/EXPORT/HOME/ETS/XEDIT
  7     24   0.143    0.017   /USR/EXPORT/HOME/ETS/TERMINFO
  8      3   0.018    0.002   /USR/EXPORT/HOME/ETS/RXF
  9  12537  74.942    8.887   /USR/EXPORT/HOME/ETS/LANG
 10    219   1.309    0.155   /USR/EXPORT/HOME/ETS/DOC
 11     53   0.317    0.038   /USR/EXPORT/HOME/ETS/SYMP
 12  16729 100.000   11.859   /USR/EXPORT/HOME/ETS
SELECT NODE NUMBER FOR REDISPLAY (* FOR SAME) AND OPTIONAL LEVEL
OR 'X' TO EXIT

9

FILESYSTEM              KBYTES    USED   AVAIL CAPACITY  MOUNTED ON
/DEV/SD0G               186414  141070   26702    84%    /USR
NUM SIZE(K) % NODE    %FILESYS  ITEM
  1   9793  78.113    6.942   /USR/EXPORT/HOME/ETS/LANG/SC1.0
  2    541   4.315    0.384   /USR/EXPORT/HOME/ETS/LANG/MAN
  3  12537 100.000    8.887   /USR/EXPORT/HOME/ETS/LANG
SELECT NODE NUMBER FOR REDISPLAY (* FOR SAME) AND OPTIONAL LEVEL
OR '0' TO GO BACK UP ONE NODE
OR 'X' TO EXIT

* 2

FILESYSTEM              KBYTES    USED   AVAIL CAPACITY  MOUNTED ON
/DEV/SD0G               186414  141070   26702    84%    /USR
NUM SIZE(K) % NODE    %FILESYS  ITEM
  1    541   4.315    0.384   /USR/EXPORT/HOME/ETS/LANG/SC1.0/MAN
  2   1436  11.454    1.018   /USR/EXPORT/HOME/ETS/LANG/SC1.0/INCLUDE
  3   1001   7.984    0.710   /USR/EXPORT/HOME/ETS/LANG/SC1.0/CG87
  4   1003   8.000    0.711   /USR/EXPORT/HOME/ETS/LANG/SC1.0/CG89
  5     12   0.096    0.009   /USR/EXPORT/HOME/ETS/LANG/SC1.0/MISALIGN
  6     52   0.415    0.037   /USR/EXPORT/HOME/ETS/LANG/SC1.0/README
  7   9793  78.113    6.942   /USR/EXPORT/HOME/ETS/LANG/SC1.0
  8    133   1.061    0.094   /USR/EXPORT/HOME/ETS/LANG/MAN/MAN1
  9    398   3.175    0.282   /USR/EXPORT/HOME/ETS/LANG/MAN/MAN3
 10      4   0.032    0.003   /USR/EXPORT/HOME/ETS/LANG/MAN/MAN5
 11    541   4.315    0.384   /USR/EXPORT/HOME/ETS/LANG/MAN
 12  12537 100.000    8.887   /USR/EXPORT/HOME/ETS/LANG
SELECT NODE NUMBER FOR REDISPLAY (* FOR SAME) AND OPTIONAL LEVEL
OR '0' TO GO BACK UP ONE NODE
OR 'X' TO EXIT

X
```

Macro Languages for Unix

PERL is fast becoming the de facto standard macro language for Unix, even though REXX is clearly superior in many ways...

- REXX has better programming structures
- Strings and arrays start at 0, not 1
- No way to specify the default value of an associative array (stem)
- Relationship between operators, functions, and precedence is very confusing. For example:

  print (1+1) +1

  prints "2" and throws "3" away as an unused side effect!

- Semicolons required (just like C)
- Arrays, lists, and scalars begin with special characters (@,$,...)
- Brackets required in compound instructions, i.e.,

```
WHILE                                  WHILE
 *                                       STMT;
   STMT;          NOT
 *;
```

- The syntax has far too many special characters and ideosyncracies.

```
/*
 * BRING IN PARMS, HANDLE DEFAULTS
 */
PARSE ARG NODE LEVEL TRACEOPT
TRACE VALUE TRACEOPT
IF NODE="" | NODE="."
THEN DO
  CALL POPEN 'PWD'
  PARSE PULL NODE
  END
IF LEVEL=""
THEN LEVEL=1
```

```
#
# BRING IN PARMS, HANDLE DEFAULTS
#
$NODE = SHIFT @ARGV;
CHOP($NODE = WD
  IF ($NODE EQ '.' || $NODE EQ '');
$LEVEL = SHIFT @ARGV;
$LEVEL = 1
  IF ($LEVEL EQ '');
```

REXX-PERL COMPARISON 2

```
/*
 * FIND APPLICABLE FILE SYSTEM AND SIZE
 */
CALL POPEN 'DF'
DO WHILE QUEUED()>0
  PARSE PULL L
  TEST=WORD(L,DFNAMEWORD.UNAME)
  IF TEST="/"
  THEN DO
       FSNAME=TEST
       FSSIZE=WORD(L,DFSPACEWORD.UNAME)
       IF DFFREEWORD.UNAME<>0
       THEN FSSIZE=FSSIZE-WORD(L,DFFREEWORD.UNAME)
       END
  ELSE IF LEFT(NODE,LENGTH(TEST))==TEST
       THEN DO
         FSNAME=TEST
         FSSIZE=WORD(L,DFSPACEWORD.UNAME)
         IF DFFREEWORD.UNAME<>0
         THEN FSSIZE=FSSIZE-WORD(L,DFFREEWORD.UNAME)
         DO WHILE QUEUED()>0
           PARSE PULL L
           END
       LEAVE
       END
  END
IF DFBLOCKS.UNAME THEN FSSIZE=FSSIZE*BLOCKSIZE.UNAME
```

REXX-PERL COMPARISON 2

```
#
# FIND APPLICABLE FILE SYSTEM AND SIZE
#
OPEN (DF, 'DF *');
WHILE (<DF>)
*
  CHOP;
  @DF = SPLIT;
  $TEST = @DF¢$DFNAMEWORD*$UNAME*-1|;
  IF ($TEST EQ '/')
  *
    $FSNAME = '/';
    $FSSIZE = @DF¢$DFSPACEWORD*$UNAME*-1|;
    $FSSIZE = $FSSIZE - @DF¢$DFFREEWORD*$UNAME*-1|
      IF (@DF¢$DFFREEWORD*$UNAME*-1| != 0);
  *
  ELSE
  *
    IF ($NODE =° /¬$TEST/)
    *
      $FSNAME = $TEST;
      $FSSIZE = @DF¢$DFSPACEWORD*$UNAME*-1|;
      $FSSIZE = $FSSIZE - @DF¢$DFFREEWORD*$UNAME*-1|
        IF (@DF¢$DFFREEWORD*$UNAME*-1| != 0);
      LAST;
    *
  *
*;
CLOSE(<DF>);
$FSSIZE = $FSSIZE * $BLOCKSIZE*$UNAME*
  IF ($DFBLOCKS*$UNAME*);
```

REXX-PERL COMPARISON 3

```
/*
 * DO A DU, BUFFER UP THE LINES, AND GET THE NODE TOTAL
 */
CALL POPEN 'DU' NODE
DO LINE=1 WHILE QUEUED()>0
  PARSE PULL L
  PARSE VAR L COUNT NAME
  NODECOUNTS.NAME=COUNT
  LINES.LINE=L
  END
LINE=LINE-1
NODECOUNT=WORD(LINES.LINE,1)
```

```
#
# DO A DU, BUFFER UP THE LINES, AND GET THE NODE TOTAL
#
OPEN(DU, "DU $NODE |");
$LINE = 1;
WHILE (<DU>)
{
  CHOP;
  @DU = SPLIT;
  ($COUNT, $NAME) = @DU[0..1];
  $NODECOUNTS{$NAME} = $COUNT;
  $LINES[$LINE] = $_;
  ++$LINE;
}
--$LINE;
$NODECOUNT = SHIFT(@DU);
CLOSE(<DU>);
```

Our Experience so far

As of this time

Applications that embed REXX are well received by commercial MIS transiting to Unix

- XEDIT
- ISPF

...REXX usage within these specific environments is highly portable

However...

REXX has not been well received as a macro facility for Unix.

- WHY
- WHAT ARE WE GOING TO DO ABOUT IT

PERL

- Familiar to current Unix System Administrators, who are also software selectors.
- Regular Expression Support
- Many Unix specific functions built-in.
    - -- Ease of Implementation
    - -- Portability
- Free, in Source Form
- Learning curve is as steep (or steeper) than the typical Unix alternatives.
- Questionable support

---

uni-REXX

- Familiar to Commercial MIS, unfamiliar to Unix System Administrators
- Little specific Unix support:
    - -- no regular expression support
    - -- very few Unix-related functions
- Commercial product, whereas most REXX language processors are bundled into the underlying OS.

Recent Work on uni-REXX

The recent improvements to uni-REXX have "played to it's strengths", improving it's usefulness for embedded applications

- Additional APIs
- Completion of standard REXX facilities
- Improved Performance

Recent Work on uni-REXX

Additional API's

- symbol table access exits
  - -- read, write, and drop
  - -- taken only when an uninitialized variable is referenced
  - -- exit can supply/accept a value or allow default processing
  - -- supplied/accepted values make no REXX symbol table entry
- compiler exits
  - -- support language extensions by embedded application
  - -- could support a preprocessor
- multi-threading support

Recent Work on uni-REXX

Completion of standard REXX facilities

- full REXX math
- Language Level 4.00 features

Note that this work has not improved REXX's position with respect to Unix

Plans to Improve Acceptance of REXX for Unix

Lots more Built-In Functions

- Unix specific, ala PERL etc. (Expose entire standard C library?)
- Regular expression support
- Process management & communication
- User interfaces: Curses for sure, possibly X as well
- Database access

Plans to Improve Acceptance of REXX for Unix

Process Management & Communication

We are currently experimenting with functions that will allow REXX to control one or more asynchronous processes via STDIN and STDOUT.

HANDLE=PIPE(COMMAND) - initiates an asynchronous process, with both STDIN and STDOUT piped back to the parent.

RC=PIPEIN(HANDLE,DATA) - receives raw data from a process's STDOUT

RC=PIPEOUT(HANDLE,DATA) - sends raw data to a process's STDIN

RC=PIPESEL(HANDLE1,HANDLE2,HANDLE3) - blocks until data is available on a controlled process's STDOUT, or on the parent task's STDIN.

"RAW" data can be a bit cumbersome (control characters are present,) but this approach allows full control of any application without it's inclusion of a "REXX message port" etc.

Current implementation uses sockets. Production quality will require pseudo-tty's instead.

Plans to Improve Acceptance of REXX for Unix

Base technology free to Academia

- executable only
  - -- no source level portability
  - -- no embedded use (given current technology)
- No support, very limited documentation

Other plans

Improved program documentation (i.e., program listing facilities)

IPC for the REXX API's

- expose APIs to invoked Unix commands
- externalize embedded language processing

REXX Shell

- better integrate the external data queue & Unix command processing
- allow REXX control over the persistent shell environment

Additional embedded applications

Lots of work in the above plans, more than we can fund in the near term.
Effectiveness of these efforts at increasing acceptance is unsure.
Prioritization will be an issue, comments are welcome.

Other Items of Interest from TWG

Combined REXX and C development project for uni-SPF

Relationship with Public Domain REXX Implementations

Combined REXX and C development project for uni-SPF

GOAL - Building upon our base of uni-REXX and uni-XEDIT, add dialog management facilities, "PDF", and the SPF editor.

Works begins on dialog management (in "C") in April, 1991.

Prototype dialog management available 3Q91.

Intensive period of work during 4Q91:

-- "PDF" created using dialog management & REXX

-- XEDIT turned into the SPF editor with REXX macros and supporting "C" code as required.

Production release 1Q92.

Combined REXX and C development project for uni-SPF

SPF project consumed 12 man-months of C coding
4 man-months of REXX coding

Produced 829K bytes of C code
612K bytes of REXX code

Highly productive REXX programming environment was invaluable to rapid prototyping and quick delivery.

Certain REXX code segments exhibit performance problems (mitigated by RISC price/performance characteristics.)

Most REXX code will probably be re-written in C eventually.

Relationship with Public Domain REXX Implementations

Additional improvements are expected in the acceptance of REXX for Unix

Commercial implementation will have it's advantages:

- completeness
- stability
- performance
- documentation
- support

TWG & iX will cooperate in establishing standards where possible

- API interface definitions
- Built-In Function definitions

Any other position would be futile, anyway!

Comments, Please.

Thanks!

PLUNGING INTO PIPES

MELINDA VARIAN
PRINCETON UNIVERSITY

PLUNGING INTO *PIPES*

Melinda Varian

Office of Computing and Information Technology
Princeton University
87 Prospect Avenue
Princeton, NJ 08544 USA

BITNET: MAINT@PUCC
Internet: maint@pucc.princeton.edu
Telephone: (609) 258-6016

REXX Symposium
May 5, 1992

I. INTRODUCTION

CMS Pipelines[1] is the most significant enhancement to CMS since REXX. It introduces into CMS the powerful data flow model of programming that was popularized by UNIX[2] pipes. UNIX pipes were built to work with a byte-oriented file system, but *CMS Pipelines* has so successfully met the challenge of making the pipeline concept work well with a record-oriented file system that *CMS Pipelines* is now being used in MVS, GCS, and MUSIC, as well as in CMS.

There are two primary reasons for discussing *CMS Pipelines* at a REXX Symposium. First, REXX and *CMS Pipelines* work so well together that the example of their synergy may inspire advances in other REXX environments. When *CMS Pipelines* was first being developed, the author of REXX, Mike Cowlishaw, graciously made a critical change to REXX to facilitate the implementation of *Pipes*. The author of *CMS Pipelines*, John Hartmann, has himself said that there would be very little point to *CMS Pipelines* without REXX. Although *CMS Pipelines* does run at the command level (or even with EXEC 2), its real power comes when it is used in conjunction with REXX. Conversely, *CMS Pipelines* magnifies the power of REXX, and that is the second reason for discussing it here. *CMS Pipelines* brings to REXX many of the capabilities that are the subjects of user group requirements for REXX enhancements, and it also augments REXX in other important ways, such as by giving it device independence. I will try today to give you a glimpse of the ways in which REXX and *Pipes* complement one another and of the reasons why CMS REXX users are so excited about *Pipes*.

---

[1] *CMS Pipelines* is part of CMS 8 in VM/ESA 1.1. Customers who are not yet running CMS 8 can order *CMS Pipelines* as a program offering, 5785-RAC, except in the United States, where *CMS Pipelines* is a Programming RPQ (P81059, 5799-DKF). The PRPQ, which includes Mike Cowlishaw's LEXX editor, is in Higher Education Software Consortium Group I-A1.

[2] UNIX is a trademark of AT&T Bell Laboratories.

The Pipeline Concept

A pipeline is simply a series of programs through which data flow, just as water flows through the sections of a water pipe. In a pipeline, a complex task is performed by processing data through several simple programs in an appropriate sequence.

The programs that are hooked together to form a pipeline are called "stages". Each stage in a pipeline reads data from the pipeline, processes them in some way, and writes the transformed data back to the pipeline. Those data are then automatically presented as input to the next stage in the pipeline. The individual programs in the pipeline are independent of one another; they need not know or care which other programs are in the pipeline. They are also device independent; each of them does its own job without concern for where the data came from or for where they are going. The output of any program can be connected to the input of any other program; thus, the programs used to perform one task can be hooked together in a different order to perform a different task. Whenever a new pipeline stage is written, it can immediately be used in conjunction with any previously existing stage.

Pipeline programming involves applying "pipethink" to break a problem into a number of small steps, each of which can then be performed by a simple program. Wherever possible, a pipeline programmer uses existing programs as the stages in a pipeline. Traditionally, programs that run in pipelines are small and have one very well-defined function, but they should also be as general-purpose as possible, to allow re-use. Because they are so small and well-defined, it is possible to make them very reliable. In other words, programs that run in pipelines should be "little gems". *CMS Pipelines* comes with a very rich collection of such gems, well over a hundred built-in programs. *CMS Pipelines* users typically find that most of their applications can be written using only the built-in programs, but if they have a need that is not addressed by a built-in program, they can easily craft their own little gems, preferably in REXX.

II. A *CMS PIPELINES* PRIMER

The Pipe Command

The pipeline concept has not been integrated into command parsing in CMS, as it has in UNIX. Instead, *CMS Pipelines* adds the new CMS command, PIPE:

```
pipe pipeline-specification
```

The argument to PIPE is a "pipeline specification". A pipeline specification is a string listing the stages to be run. The stages are separated by the "stage separator character", which is usually a vertical bar ("|"):

```
pipe stage-1 | stage-2 | stage-3 | stage-4
```

When CMS sees this PIPE command (whether in an EXEC or typed on the command line), it passes control to the PIPE module, which interprets the argument string as a pipeline containing four stages. The pipeline parser locates the four programs and checks for correct syntax in the invocations of any that are built-in programs. If all the stages are specified correctly, the pipeline is executed; otherwise, the pipeline parser issues useful error messages and exits.

Device Drivers

In UNIX, a program can do I/O to a device in exactly the same way it does I/O to a file. Under the covers, the system has "device drivers" to make this work. Because CMS does not provide such device transparency, *CMS Pipelines* has its own device drivers, pipeline stages that connect the pipeline to host interfaces, thus allowing other pipeline stages to be completely independent of host interfaces.

CMS Pipelines provides a large number of device drivers. A very simple pipeline might contain only device drivers. We may as well be traditional and start with this one:

```
pipe literal Hello, World! | console
```

Here, the device driver literal inserts a record containing the phrase "Hello, World! " into the pipeline. The device driver console then receives that record and displays it on the console.

This pipeline reads lines from the console and writes them to the punch:

```
pipe console | punch
```

(It continues reading from the console and writing to the punch until it reaches end-of-file, *i.e.*, until it receives a null line as input.)

As the use of console in these two examples shows, some device drivers can be used for either reading or writing. If they are the first stage in the pipeline, they *read* from the host interface. If they come later in the pipeline, they *write* to the host interface. This pipeline performs a simple echo operation:

```
pipe console | console
```

It just reads lines from the console and writes them back to the console. A similar pipeline performs a more useful task; it copies a file from one tape to another:

```
pipe tape | tape tap2 wtm
```

The first tape stage knows to read, because it can sense that it is the first stage in the pipeline; the second tape stage knows to write, because it can sense that it is *not* the first stage in the pipeline. tap2 and wtm are arguments to the second tape stage. When the pipeline dispatcher invokes the second tape stage, it passes along those arguments, which tape recognizes as instructions to use the CMS device TAP2 and to write a tapemark at the end of the data.

There are several device drivers to read and write CMS files. Some of them will look familiar to you if you know UNIX, but may look rather strange if you do not:

- The < ("disk read") device driver reads a CMS file and inserts the records from the file into the pipeline. Thus, this pipeline copies a file from disk to tape:

```
pipe < fn ft fm | tape
```

- > ("disk replace") writes records from the pipeline to the CMS file specified by its arguments, replacing any existing file of the same name, so this pipeline copies a file from tape to disk:

```
pipe tape | > fn ft fm
```

- >> ("disk append") is the same as >, except that it appends an existing file of the specified name, if any, rather than replacing it. Thus, this pipeline also copies a file from tape to disk, but if the named file already exists, it is appended, not replaced:

```
pipe tape | >> fn ft fm
```

(Note that although <, >, and >> look like the UNIX redirection operators, they are actually the names of programs; like other CMS program names, they must be delimited by a blank.)

An output device driver is not necessarily the last stage of a pipeline. Output device drivers write the records they receive from the pipeline to their host interface, but they also pass those records back to the pipeline, which then presents them as input to the following stage, if there is one. For example, this pipeline reads a CMS file and writes the records to a CMS file, to the console, to the punch, and to a tape:

```
pipe < fn ft fm | > outfn outft outfm | console | punch | tape wtm
```

If you wanted to include that PIPE command in a REXX EXEC, you would need to keep in mind that the entire command is a string, only portions of which should have variables substituted. Thus, in an EXEC you would write that PIPE command something like this:

```
'PIPE <' infn inft infm '| >' outfn outft outfm '| console | punch | tape wtm'
```

That is, you would quote the parts that are not variable, while allowing REXX to substitute the correct values for the variable fields, the filenames.

As PIPE commands grow longer, using the linear form in EXECs becomes somewhat awkward. Most experienced "plumbers" prefer to put longer pipelines into "portrait format", with one stage per line, thus:

```
'PIPE (name DRIVERS)',
   '<' infn inft infm '|',
   '>' outfn outft outfm '|',
   'console |',
   'punch |',
   'tape wtm'
```

You can use the FMTP XEDIT macro, which comes with *CMS Pipelines*, to reformat a PIPE command into portrait format. Note the commas at the ends of the lines; those are REXX continuation characters. This pipeline specification will still be a single string once REXX has interpreted it.

Note also the "global option" name in parentheses immediately following the PIPE command. This gives the pipeline a name by which it can be referenced in a traceback, should an error occur while the pipe is running. (There are a number of other global options, but this is the only one we will meet in this session.)

Once you have the pipeline in portrait format, you can key in comments on each line and then invoke the SC XEDIT macro, which comes with *CMS Pipelines*, to line them up nicely for you:

```
'PIPE (name DRIVERS)',                  /* Name for tracing */
   '<' infn inft infm '|',              /* Read CMS file    */
   '>' outfn outft outfm '|',           /* Copy to CMS file */
   'console |',                         /* And to console   */
   'punch |',                           /* And to punch     */
   'tape wtm'                           /* And to tape      */
```

You will notice that all the device drivers observe the rule that a program that runs in a pipeline should be able to connect to any other program. Although the device drivers are specialized on the side that connects to the host, they are standard on the side that connects to the pipeline.

There are four very useful device drivers to connect a pipeline to the REXX environment:

- var, which reads a REXX variable into the pipeline or sets a variable to the contents of the first record in the pipeline;
- stem, which retrieves or sets the values in a REXX stemmed array;
- rexxvars, which retrieves the names and values of REXX variables; and
- varload, which sets the values of the REXX variables whose names and values are defined by the records in the pipeline.

All four of these stages allow you to specify which REXX environment is to be accessed. If you do not specify the environment, then the variables you set or retrieve are from the EXEC that contains your PIPE command. But you may instead specify that the variables are to be set in or retrieved from the EXEC that called the EXEC that contains your PIPE command or another EXEC further up the chain, to any depth. For example, this pipeline:

```
'PIPE stem parms. 1 | stem parms.'
```

retrieves the stemmed array "parms" from the environment one level back (that is, from the EXEC that called this EXEC) and stores it in the stemmed array "parms" in this EXEC. (If these two stages are reversed, then the array is copied in the opposite direction.)

rexxvars retrieves the names and values of all exposed REXX variables from the specified REXX environment and writes them into the pipeline, starting with the source string:

```
'PIPE rexxvars 1 | var source1'     /* Get caller's source. */
'PIPE rexxvars 2 | var source2'     /* And his caller's.    */

Parse Var source1 . . . fn1 .
Parse Var source2 . . . fn2 .

Say 'I was called from' fn1', which was called from' fn2'.'
```

In this example, rexxvars is used twice, once to retrieve the variables from the EXEC that called this one and once to retrieve the variables from the EXEC that called that one. In each case, a var stage is then used to store the first record produced by rexxvars (the source string) in a variable in this EXEC, where it can be used like any other REXX variable.

Another very useful group of stages issue host commands and route the responses into the pipeline. Among these "host command processors" are:

- cp, which issues CP commands;
- cms, which issues CMS commands with full command resolution through the CMS subcommand environment, just as REXX does for the Address CMS instruction; and
- command, which issues CMS commands using a program call with an extended parameter list, just as REXX does for the Address Command instruction.

Each of these stages issues its argument string as a command and then reads any records from its input stream and issues those as commands, too. The command responses are captured, and each response line becomes a record in the pipeline. For example, in this pipeline:

```
'PIPE cp query dasd | stem dasd.'
```

the cp stage issues a CP QUERY DASD command and writes the response into the pipeline, where the stem stage receives it and writes it into the stemmed array "DASD", setting "DASD.0" to the count of the lines in the response.

There are a great variety of other device drivers, for example:

- xedit, which writes records from an XEDIT session to the pipeline or *vice versa;*
- stack, which reads or writes the CMS program stack;
- sql and ispf, which interface to SQL and ISPF;
- qsam, which reads MVS files (and writes them under MVS);
- storage, which reads or writes virtual machine storage; and
- subcom, which sends commands to a subcommand environment.

The list of device drivers goes on and on, and it continues to grow.

Other Built-in Programs

Pipelines built only of device drivers do not really show the power of *CMS Pipelines* (although they may be quite useful, especially as they often out-perform the equivalent native CMS commands). There are dozens of other *CMS Pipelines* built-in programs. Most of these are "filters", programs that can be put into a pipeline to perform some transformation on the records flowing through the pipeline.

Using Pipeline Filters: A simple pipeline consisting of a couple of device drivers wrapped around a few filter stages provides an instant enhancement to the CMS command set. Once you have had some practice, you will find yourself typing lots of little "throwaway" pipes right on the command line.

Many *CMS Pipelines* filters are self-explanatory (especially as many of them behave just like the XEDIT subcommand of the same name). For example, this pipeline displays the DIRECTORY statement from a CP directory:

```
pipe < user direct | find DIRECTORY | console
```

The find filter selects records using the same logic as the XEDIT FIND subcommand.

This pipeline displays all the occurrences of the string "GCS" in the *CMS Pipelines* help library:

```
pipe < pipeline helpin | unpack | locate /GCS/ | console
```

The unpack filter checks whether its input is a packed file and, if it is, does the same unpack operation that the CMS COPYFILE and XEDIT commands do. The locate filter selects records using the same logic as the XEDIT LOCATE subcommand.

This pipeline tells you how many words there are in one of your CMS files:

```
pipe < plunge script a | count words | console
```

A slightly more elaborate pipeline tells you how many *different* words there are in that same file:

```
pipe < plunge script a | split | sort unique | count lines | console
```

split writes one output record for every blank-delimited word in its input; sort unique then sorts those one-word records and discards the duplicates, passing the unique records on to count lines to count. count writes a single record containing the count to its output stream. console reads that record and displays it on the console.

This pipeline writes a CMS file containing fixed-format, 80-byte records to a tape, blocking it in a format suitable to be read by other systems:

```
pipe < gqopt fortran a | block 16000 | tape
```

This pipeline writes a list of the commands used with "SMART" (RTM) to a CMS file:

```
pipe literal next| vmc smart help| strip trailing | > smart commands a
```

literal writes a record containing the word "next". The vmc device driver sends a help command to the SMART service machine via VMCF and writes the response to the pipeline. It then reads the single record from its input and sends a next command to the SMART service machine, again writing the response to the pipeline. strip trailing removes trailing blanks from the records that pass through it, thus turning the blank lines in the response from SMART into null records. > reads records from its input, discards those that are null, and writes the others to the file SMART COMMANDS A.

And here is a pipeline I especially like; it would be typed on the XEDIT command line:

```
pipe cms query search | change //INPUT / | subcom xedit
```

In this pipeline, the cms device driver issues the CMS QUERY SEARCH command and routes the response into the pipeline; the change filter (which works like the XEDIT CHANGE subcommand) changes each line of the response into an XEDIT INPUT subcommand; and then subcom sends each line to XEDIT, which executes it as a command. This is a very easy way to incorporate the response from a command into the text of a file you are editing.

The Specs Filter: Now, let's look at one of the less obvious filters, specs. specs selects pieces of an input record and puts them into an output record. It is very useful and not really as complex as it looks at first. Its syntax was derived from the syntax for the SPECS option of the CMS COPYFILE command, but it has long since expanded far beyond the capabilities of that option:

- The basic syntax of specs is:

  specs input-location output-location

  with as many input/output pairs as you need.

- The input location may be a column range, such as "10-14". "10.5" means the same thing as "10-14". "1-*" means the whole record. "words 1-4" means the first four blank-delimited words. The input may also be a literal field, expressed as a delimited string, such as "/MSG/", or it may be "number", to get a record number.

- The output location may be a starting column number, or "next", which means the next column, or "nextword", which leaves one blank before the output field.

- A conversion routine, such as "c2d", may be specified between the input location and the output location. The specs conversion routines are similar to the REXX conversion functions and are applied to the value from the input field before it is moved into the output field.

- A placement option, "left", "center", or "right", may be specified following the output location; for example, "number 76.4 right" puts a 4-digit record number right-aligned starting in column 76.

```
/* PIPEDS EXEC:                  Find lrecl of an OS dataset */

Parse Upper Arg dsname fm

'PIPE (name PIPEDS)',
   'command LISTDS' fm '( FORMAT |',     /* Issue LISTDS.    */
   'locate /' dsname '/ |',              /* Locate file we want.  */
   'specs  word 2  1 |',                 /* Lrecl is second word. */
   'console'                             /* Display lrecl.        */
```

PIPEDS EXEC is a simple example of using specs. PIPEDS displays the logical record length of an OS dataset. The command stage issues a CMS LISTDS command with the FORMAT option and routes the response into the pipeline, where locate selects the line that describes the specified dataset, *e.g.*:

U 6447 PO 02/25/80 RES342 B SYS5.SNOBOL

specs selects only the second word of that line, the logical record length ("6447"), and moves it to column 1 of its output record, which console then reads and displays.

```
pipe < cms exec a | specs 1-27 1 8-27 nextword | > cms exec a
```

This is another simple example of using specs. The arguments to specs here are two pairs of input-output specifications. The first input-output pair ("1-27 1") copies the data from columns 1-27 of the input record to columns 1-27 of the output record. The second input-output pair ("8-27 nextword") copies the data from columns 8-27 of the input record to columns 29-48 of the output record; that is, a blank is left between the first output field and the second output field. So, this pipeline would be used to duplicate the filenames in a CMS EXEC created by the EXEC option of the CMS LISTFILE command. (This pipeline is almost 500 times as fast as the XEDIT macro I used to use to do this same thing.)

Augmenting REXX: People often start in gradually using *CMS Pipelines* in EXECs, first just taking advantage of the built-in programs that supply function that is missing or awkward in REXX. Here is a function that has been implemented a zillion times in REXX or Assembler:

```
'PIPE stem bananas. | sort | stem bunch.'
```

That sorts the values in the stemmed array "bananas" and puts them into the array "bunch".

Here is an example of using specs to augment REXX (which has no "c2f" function):

```
'PIPE var cpu2busy | specs 1-* c2f 1 | var cpu2busy'
```

The device driver var picks up the REXX variable "cpu2busy", which contains a floating-point number stored in the System/370 internal representation (*e.g.*, '4419B600'x), and writes it to the pipeline. specs reads the record passed from var and converts it to the external representation of the floating-point number (6.582E+03), and then the second var stage stores the new representation back into the same REXX variable, allowing it to be used in arithmetic operations.

Another function *CMS Pipelines* brings to REXX programmers is an easy way to process all the variables that have a given stem. In the example below, rexxvars writes two records into the pipeline for each exposed variable. One record starts with "n " and contains the variable's name; the other starts with "v " and contains its value. The find stage selects only the name records for variables with the stem "THINGS". specs removes the "n ", and stem puts the names of the "THINGS" variables into the stemmed array "vars", where they can be accessed with a numeric index. (The buffer stage prevents the stem stage from creating new variables while rexxvars is still loading the existing variables.)

```
'PIPE',                     /* Discover stemmed variables:  */
   'rexxvars |',            /*   Get all variables.         */
   'find n THINGS.|',       /*   Select names of THINGS.    */
   'specs 3-* 1 |',         /*   Remove record type prefix. */
   'buffer |',              /*   Hold all records.          */
   'stem vars.'             /*   Names of THINGS into stem. */

Do i = 1 to vars.0
   Say vars.i '=' Value(vars.i)
End
```

rexxvars has many other uses; for example, you might wish to use it in a syntax error routine to dump all exposed variables to a file for debugging. The combination of rexxvars and varload provides such capabilities as saving the state of an EXEC and later restoring it.

varload uses the information in its input records to set REXX variables. The input to varload consists of records that contain a delimited string specifying a variable name, followed by the value to which the variable is to be set. The canonical example of using varload and rexxvars is a pair of EXECs written by Jim Colten, of the University of Minnesota, with contributions by Chuck Boeheim and Michael Friendly. The first one is called to save all CP settings:

```
/* CPQSET EXEC: Load CP SET values into REXX stem.          */

'PIPE (name CPQSET)'        ,
   '  cp query set'         ,    /* Get QUERY SET output.    */
   '| split ,'              ,    /* Split into settings.     */
   '| specs /=CPVAR./ 1'    ,    /* Build up stem name,      */
   '    word 1 next'        ,    /*   delimiters, and        */
   '    /=/ next'           ,    /*   value for VARLOAD.     */
   '    word 2-* nextword' ,
   '| varload 1'                /* Create vars for caller.  */
```

The cp stage issues a CP QUERY SET command and routes the response into the pipeline, where the split stage splits the records at the commas, thus producing one record for each CP setting. The specs stage converts these records into the format required by varload: a delimited string containing the name (in this case, of the form "=CPVAR.xxx="), followed by the value. varload 1 receives these records and loads the specified variables into the caller's environment. The companion EXEC performs the inverse operation:

```
/* CPRESET EXEC: Restore CP variables from REXX stem.       */

'PIPE (name CPRESET)'       ,
   '  rexxvars 1'           ,   /* Get caller's variables. */
   '| drop 1'               ,   /* Drop source line.       */
   '| spec 3-* 1'           ,   /* Join name & value,      */
   '    read 3-* nextword'  ,   /*   removing type prefix. */
   '| find CPVAR.'        || ,  /* Only our stem.          */
   '| nfind CPVAR.0'        ,   /* Discard the counter.    */
   '| nlocate /ECMODE/'     ,   /* SET ECMODE is BAD!      */
   '| spec /CP SET/ 1'      ,   /* Make into CP command,   */
   '    7-* nextword'       ,   /*   removing stem name.   */
   '| cp'                   ,   /* Let CP do reSET.        */
   '| console'                 /* Display any messages.   */
```

Replacing EXECIO: EXECIO is usually the first thing to go when one learns *CMS Pipelines*. Anything that can be done with EXECIO can be done with *CMS Pipelines*, generally faster and always more straightforwardly. (And replacing EXECIO with a pipeline makes it easier to port an EXEC between CMS and MVS.) Let's look at a few EXECIO examples from various IBM manuals, along with the equivalent pipelines:

- These both read the first three records of a CMS file into the stemmed array "X" and set the value of "X.0" to 3:

```
'EXECIO 3 DISKR MYFILE DATA * 1 ( STEM X.'

'PIPE < myfile data * | take 3 | stem x.'
```

- These both issue a CP QUERY USER command in order to set a return code (without saving the response):

```
'EXECIO 0 CP ( STRING QUERY USER GLORP'
+++ RC(1045) +++

'PIPE cp query user glorp'
+++ RC(45) +++
```

- These both put a blank-delimited list of the user's virtual disk addresses into the REXX variable "used":

```
Signal Off Error
'MAKEBUF'
Signal On Error
theirs = Queued()
'EXECIO * CP ( STRING Q DASD'
used = ''
Do While Queued() > theirs
   Pull . cuu .
   used = used cuu
End
'DROPBUF'

'PIPE cp q dasd | specs word 2 1 | join * / / | var used'
```

The EXECIO case comes from the REXX *User's Guide*. Admittedly, it is rather old-fashioned code; nevertheless, its eleven lines make up an all too familiar example of manipulating the CMS stack. In the pipeline, the cp device driver issues the CP QUERY DASD command and routes the response into the pipeline. specs selects the second word from each input record and makes it the first (and only) word in an output record. join * joins all these records together into one record, inserting the delimited string in its argument (a blank) between the values from the individual input records. And var stores this single record into the variable "used".

Pipeline Programs: After a while, you will find yourself not just augmenting your EXECs with small pipes, but also writing EXECs that are predominantly pipes, such as REACCMSG EXEC:

```
/* REACCMSG EXEC:       Notify users to re-ACCESS a changed disk */

Parse Arg vaddr .

'PIPE (name REACCMSG)',
   'cp q links' vaddr '|',              /* Issue CP QUERY LINKS     */
   'split at , |',                      /* Get one user per line    */
   'strip |',                           /* Remove leading blanks    */
   'sort unique 1-8 |',                 /* Discard duplicates       */
   'specs /MSG/ 1',                     /* Make into MSG commands   */
      'word 1 nextword',                /* Fill in userid           */
      '/Please re-ACCESS your/ nextword',
      'word 2 nextword',                /* Fill in virtual address */
      '/disk./ nextword |',
   'cp'                                 /* Issue MSG commands       */
```

REACCMSG is used to send a message to all the users linked to a particular CMS disk to let them know that they should re-ACCESS the disk because it has been changed. It uses built-in programs we have seen before, but in a slightly more sophisticated manner: split receives the response from the CP QUERY LINKS command:

```
PIPMAINT 320 R/O, MAINT    420 R/O, TDTRUE   113 R/O, Q0606    320 R/O
Q0606    113 R/O, SERGE    420 R/O
```

and splits those records into multiple records by breaking them up at the commas between items; strip removes the leading blanks; and sort unique sorts the records on the userid field in the first eight columns and discards any duplicates, so that each user will be sent only one message. This example shows a more elaborate use of specs than before, but it is not difficult to understand if you keep in mind that specs's arguments are always pairs of definitions for input and output. This specs stage has been written in portrait format with each input-output pair on a separate line. You will note that the input definitions in three of the five pairs here are for literals. The first input-output pair puts the literal "MSG" into columns 1-3 of the output record; the second pair puts the userid from the first word of the input record ("word 1") into columns 5-12 of the output record; and so on. Then as each record flows from the specs stage to the cp stage, cp issues it as a CP MSG command.

The next example is a simple service machine that uses the starmsg device driver to connect to the CP *ACCOUNT system service, so that it can monitor attempts to LOGON to the system with an invalid password. Each time CP produces an accounting record, this starmsg stage receives that record via IUCV and writes it to the pipeline (prefacing it with an 8-byte header). The locate stage discards all but the "Type 4" records, which are the ones that CP produces when the limit of invalid LOGON passwords is reached. specs formats a message containing a literal and three fields from the accounting record, which console then displays. (Note the stage separators on the left side here. This is a widely used alternative portrait format.)

This pipeline runs until you stop it by using the haccount immediate command, which *CMS Pipelines* sets up for you when it establishes the connection to the *ACCOUNT system service. starmsg can also be used to connect to several other CP system services, including *MSG and *MSGALL.

```
/* HACKER EXEC:          Display Type 4 Accounting Records. */

'CP RECORDING ACCOUNT ON LIMIT 20'

'PIPE (name STARMSG)',
   '| starmsg *account',          /* Connect to *ACCOUNT.      */
   '| locate 88 /4/',             /* Only Type 4 records.      */
   '| specs',                     /* Format warning message:   */
   '     /Hacker afoot? / 1',     /*    literal,               */
   '     9.8 next',               /*    ACOUSER,               */
   '     37.4 nextword',          /*    ACOTERM@,              */
   '     79.8 nextword',          /*    ACOLUNAM.              */
   '| console'                   /* Display on console.       */

If Userid() <> 'OPERACCT'
Then 'CP RECORDING ACCOUNT OFF PURGE QID' Userid()
```

The next example may be a bit arcane, but it can be very useful; it reads a file containing textual material of arbitrary content and record length and produces a file containing the same text formatted as Assembler DC instructions for use, say, as messages:

```
/* MAKEDC EXEC:   Reformat text into Assembler DC statements */

Parse Arg fn ft fm .          /* File to be processed.       */

"PIPE (name MAKEDC)",
   "<" fn ft fm "|",          /* Read the file.              */
   "change /&/&&/ |",         /* Double the ampersands.      */
   "change /'/''/ |",         /* Double the single quotes.   */
   "specs",                   /* Reformat to DC statement:   */
   "   /DC/ 10",              /*    literal "DC" in col 10;  */
   "   /C'/ 16",              /*    literal "C'" in col 16;  */
   "   1-* next",             /*    entire record next; and  */
   "   /'/ next |",           /*    terminate with quote.    */
   "asmxpnd |",               /* Split to continuations.     */
   ">" fn "assemble a"        /* Write the new file.         */
```

The two change filters double any ampersands or quotes in the text. For each input record, specs builds an output record that has "DC" in column 10 and "C" in column 16, followed by the input record enclosed in single quotes. asmxpnd then examines each record to determine whether it extends beyond column 71; if so, it breaks the record up into two or more records formatted in accordance with the Assembler's rules for continuations. And finally, > writes the reformatted records to a CMS file. Thus, if the input file were to contain the line:

The PACKAGE file records have ' &1 &2 ' in columns 1-7 and a filename,

then these two records would appear in the output file:

```
DC    C'The PACKAGE file records have '' &&1 &&2 '' in columns*
       1-7 and a filename,'
```

Selection Filters: There are many more *CMS Pipelines* filters to learn, but I want to mention one class in particular, the selection filters:

between	frlabel	nfind	outside	unique
drop	inside	nlocate	take	whilelab
find	locate	notinside	tolabel	

The selection filters are used to select certain records from among those passing through the pipeline, while discarding all others. A cascade of selection filters can quickly select the desired subset of even a very large file. I routinely use pipelines to filter files containing tens (or even hundreds) of thousands of records to select the records I need for some purpose.

One simple example is a filter I use with the NETSTAT CLIENTS command. NETSTAT CLIENTS produces hundreds of lines of output, several lines for each user who has used TCP/IP since the last IPL. The first line of the response for each user begins with the string "Client:" followed by the userid; and one of the other lines begins with the string "Last Touched:". Usually, when I issue a NETSTAT CLIENTS command, I need to see only these two lines for each of four servers. The eight lines I want are easily isolated using two selection filters:

```
/* STATPIPE EXEC:      Display "Last Touched" for BITFTPn. */

'PIPE',
   'command NETSTAT CLIENTS |',
   'between   /Client: BITFTP/ /Last Touched:/ |',
   'notinside /Client: BITFTP/ /Last Touched:/ |',
   'console'
```

The command stage issues a NETSTAT CLIENTS command and routes the response into the pipeline. The between filter selects *groups* of records; its arguments are two delimited strings, describing the first and last records to be selected for each group. So, the between stage here selects groups of records that begin with a record that begins "Client: BITFTP" and that end with a record that begins "Last Touched:". notinside then further refines the data by selecting only those records that are *not* between a record that begins with "Client: BITFTP" and a record that begins with "Last Touched:". That leaves us with only those two lines for each client I am interested in, the ones whose userids start "BITFTP".

You will likely find that many of your pipelines process the output of CP or CMS commands or CMS or MVS programs. The output from UNIX commands and programs is generally designed to be processed by a pipe, so it tends to be essentially "pure data", with few headers and trailers. With CP, CMS, and MVS output, however, you generally need to winnow out the chaff to get down to the data. Although I cannot go over the selection filters in detail today, they are easy to use and quite powerful, so you should not hesitate to process listing files that were designed to be read by humans and that have complicated headers and trailers and carriage control. It is very easy to write a pipe that reads such a file and pares it down to the bare data.

LIST2SRC EXEC is an example of really using the selection filters; I will leave the detailed interpretation of LIST2SRC as an exercise for you. Basically, LIST2SRC reads a LISTING file produced by Assembler H and passes it through a series of selection filters, winnowing out the chaff in order to reconstruct the original source file. Although this is a "quick & dirty" program (and not quite complete), it is a good example of "pipethink", of solving a complex problem by breaking it up into simple steps:

```
/* LIST2SRC EXEC:        Re-create the source from a LISTING file */
Signal On Novalue

Parse Arg fn .

'PIPE (name LIST2SRC)',
   '| <' fn 'listing *',          /* Read the LISTING file       */
   '| mctoasa',                  /* Machine carriage ctl => ASA */
   '| frlabel -  LOC',           /* Discard to start of program */
   '| drop 1',                   /* Drop that '-  LOC' line too */
   '| tolabel - POS.ID',         /* Keep only up to relocation  */
   '| tolabel -SYMBOL',          /*    dictionary or cross-ref  */
   '| tolabel 0THE FOLLOWING STATEMENTS',  /*    or diagnostics  */
   '| outside /1/ 2',            /* Drop 1st 2 lines on each pg */
   '| nlocate 5-7 /IEV/',        /* Discard error messages      */
   '| nlocate 41 /+/',           /* Discard macro expansions    */
   '| nlocate 40 /                 /',  /* Discard blank lines  */
   '| specs 42.80 1',            /* Pick out source "card"      */
   '| >' fn 'assemble a fixed'   /* Write new source (RECFM F)  */
```

Subroutine Pipelines

Once you have been using *CMS Pipelines* for a while, you may find that there are some sequences of stages that you use often:

pipe stage-a | **stage-b** | **stage-c** | **stage-d** | stage-e

pipe stage-x | **stage-b** | **stage-c** | **stage-d** | stage-y

In that case, it is time to move those stages into a subroutine pipeline, polish them a bit, generalize them a bit, and create your own little gem:

```
/* MYSUB REXX */

'CALLPIPE *: | stage-b | stage-c | stage-d | *:'
```

Then whenever you need the function performed by your subroutine, you simply use its name as a stage name ("mysub" in this case):

```
pipe stage-a | mysub | stage-e

pipe stage-x | mysub | stage-y
```

The subroutine may look a bit mysterious, but it is simply a pipeline stage written in REXX. If we look at it again in portrait format, it can be demystified quickly:

```
/* MYSUB REXX:                    Generic subroutine pipeline  */

'callpipe',                           /* Invoke pipeline       */
   '*: |',                            /* Connect input stream  */
   'stage-b |',
   'stage-c |',
   'stage-d |',
   '*:'                               /* Connect output stream */

Exit RC
```

There are just a few things one needs to understand about subroutine pipelines:

1. The *CMS Pipelines* command callpipe says to run a subroutine pipeline; callpipe has the same syntax and the same options as the PIPE command itself.

2. Those "*:" sequences are called "connectors". The connector at the beginning tells the pipeline dispatcher to connect the *output* from the previous stage of the calling pipeline to the *input* of the first stage of this subroutine pipeline, stage-b. The connector at the end says to connect the *output* from the last stage of this subroutine pipeline, stage-d, to the *input* of the next stage in the calling pipeline.

3. When you use REXX to write an XEDIT subroutine, the default subcommand environment is XEDIT. Similarly, when you use REXX to write a *CMS Pipelines* subroutine, the default subcommand environment executes *CMS Pipelines* commands. Thus, if you wish to issue CP or CMS commands in your subroutine, you will need to use the REXX Address instruction.

4. When you use REXX to write an XEDIT subroutine, the subroutine has a filetype of XEDIT, but when you use REXX to write a *CMS Pipelines* subroutine, the filetype is *not* PIPE. It is REXX.

5. Arguments passed to a subroutine are available to the REXX Parse Arg instruction.

Let's look at an example of a real subroutine pipeline, HEXSORT, which sorts hexadecimal numbers. An ordinary sort does not work for hexadecimal numbers *(i.e.,* base 16 numbers, expressed with the "numerals" 0-9, A-F), because the EBCDIC collating sequence sorts A-F before 0-9. This handy little subroutine pipeline sorts hexadecimal data correctly by using the trick of temporarily translating A-F to characters higher in the collating sequence than 0-9 (which are F0-F9 in hexadecimal):

```
/* HEXSORT REXX:          Hexadecimal sort,   0123456789ABCDEF  */

Parse Arg sortparms                          /* Get parms, if any     */

'callpipe (name HEXSORT)',                   /* Invoke pipeline       */
   '*: |',                                   /* Connect input stream  */
   'xlate 1-* A-F fa-ff fa-ff A-F |',        /* Transform for sort    */
   'sort' sortparms '|',                     /* Sort w/caller's parms */
   'xlate 1-* A-F fa-ff fa-ff A-F |',        /* Restore               */
   '*:'                                      /* Connect output stream */

Exit RC
```

The arguments to the xlate stages here are a column range, "1-*", which means the entire record, followed by pairs of character ranges specifying "to" and "from" translations. Records flow in from the calling pipeline through the beginning connector; they are processed through the xlate, sort, and xlate stages; and then they flow out through the end connector back into the calling pipeline. If the caller specifies an argument, that argument is passed to the sort stage to define a non-default sort operation. Here is a typical invocation:

```
'PIPE stem mdisk. | hexsort 7.3 | stem mdisk.'
```

That sorts an array of minidisk records from a CP directory into device address order. (The device addresses are hexadecimal numbers in columns 7-9 of the minidisk records.)

Of course, it is not necessary to put these operations into a subroutine. You could simply use the xlate-sort-xlate sequence in all your pipelines, whenever you need to do a hexadecimal sort, but it is much better to hide such complexity. Once you have this subroutine built, you can invoke it by name from any number of pipelines and need never think about the problem again.

Furthermore, by building the subroutine with a simple, well-defined interface and at the same time making its function as generic as possible, you create a piece of code that can be used over and over again. Here is another example of invoking HEXSORT:

```
'PIPE cp q nss map | drop 1 | hexsort 33-44 | > nss map a'
```

That issues a CP QUERY NSS command, drops the header line from the response, and sorts the remaining lines to produce a list of saved systems in memory address order. (The virtual memory addresses are hexadecimal numbers starting in column 33 of the response lines.)

A subroutine pipeline is often the cleanest way to package a function that you have implemented with *CMS Pipelines*. If you make it a subroutine pipeline, then the people you give it to can easily invoke it from their own pipes.

Writing REXX Filters

The time will come when you have a problem that cannot be solved by any reasonable combination of *CMS Pipelines* built-in programs. You will need to write a filter of your own, preferably in REXX. A REXX filter is similar to the simple subroutine pipelines we have just been looking at. It has a filetype of REXX; its subcommand environment executes *CMS Pipelines* commands; it is invoked by using its name as a stage in a pipeline; and it can receive passed arguments.

You will find writing your own pipeline filters in REXX to be very easy once you understand the basics. When I am writing a filter, I always start with this dummy filter that does nothing at all except pass records through unchanged:

```
/* NULL REXX:                          Dummy pipeline filter  */
Signal On Error

Do Forever                               /* Do until EOF      */
   'readto record'                       /* Read from pipe    */
   'output' record                       /* Write to pipe     */
End

Error: Exit RC*(RC<>12)                  /* RC = 0 if EOF     */
```

There are only a few new things one needs to learn to understand this REXX filter:

1. The *CMS Pipelines* command readto reads the next record from the pipeline into the specified REXX variable ("record" in this case).

2. The *CMS Pipelines* command output writes a record to the pipeline. The contents of the record are the results of evaluating the expression following the output command (again, in this case, the value of the REXX variable "record").

3. The pipeline dispatcher sets return code 12 to indicate end-of-file. A readto command completes with a return code of 12 when the stage before it in the pipeline has no more records to pass on to it. An output command completes with a return code of 12 when the stage following it in the pipeline has decided to accept no more input records.

So, this filter, NULL, reads a record from the pipeline and writes it back to the pipeline unchanged. It keeps on doing that until an error is signalled, *i.e.*, until a non-zero return code is set. That causes a transfer to the label "Error" in the last line of the EXEC. The most likely non-zero return code would be a return code 12 from the readto command, which would indicate end-of-file on the input stream, but the output command could get return code 12 instead, or there could be a real error. If the return code is 12, then before exiting the filter sets its own return code to 0 to indicate normal completion. Any other return code is passed back to the caller.

The effect of including the NULL filter in a pipeline:

pipe stage-a | **null** | stage-b | stage-c

is simply to make the pipeline run a bit slower. But once you understand NULL, you can quickly go on to writing useful filters, such as REVERSE, which reverses the contents of the records that pass through it:

```
/* REVERSE REXX:          Filter that reverses records  */
Signal On Error

Do Forever                              /* Do until EOF     */
   'readto record'                      /* Read from pipe   */
   'output' Reverse(record)             /* Write to pipe    */
End

Error: Exit RC*(RC<>12)                 /* RC = 0 if EOF    */
```

We can make that example slightly more complex, to illustrate one more concept that you will need when writing filters. This filter reverses only the even-numbered lines passing through it:

```
/* BOUSTRO REXX:   Filter that writes records boustrophedon */
Signal On Error

Do recno = 1 by 1                        /* Do until EOF      */
   'readto record'                       /* Read from pipe    */
   If recno // 2 = 0                     /* If even-numbered  */
      Then record = Reverse(record)      /*    line, reverse  */
   'output' record                       /* Write to pipe     */
End

Error: Exit RC*(RC<>12)                  /* RC = 0 if EOF     */
```

Each stage in a pipeline runs as a "co-routine", which means that it runs concurrently with the other stages in the pipeline. It is invoked once, when the pipeline is initiated, and remains resident. So, when BOUSTRO is ready for another record, it calls upon the pipeline dispatcher by doing a readto. The dispatcher may then decide to dispatch some other co-routine, but it will eventually return control to this one, which will continue reading and writing records until an error is signalled. Thus, when you are writing a *CMS Pipelines* filter, you need not worry (as I did at first) about where to save local variables, such as "recno" here, between "calls" to your filter. Your filter is called only once and then runs concurrently with the other stages in the pipeline. There is nothing special that your filter needs to do in order to run concurrently with the other stages; the pipeline dispatcher takes care of all that for you.

I would like to show one more example of a simple REXX filter, AVERAGE, which illustrates the point that your filter can decide not to write a record back to the pipeline for every record it reads from the pipeline. AVERAGE first reads all the input records; then, when it gets end-of-file on its input, it calculates the contents of a single output record and writes that to the pipeline:

```
/* AVERAGE REXX:                    Filter that averages input  */
Signal On Error

acum = 0                                    /* Initialize        */

Do nobs = 0 by 1                            /* Do until EOF      */
   'readto record'                          /* Read from pipe    */
   Parse Var record number .                /* Get number        */
   acum = acum + number                     /* Accumulate        */
End

Error:  If RC = 12                          /* If EOF, then      */
   Then 'output' Format(acum/nobs,,2)       /*    write average  */
Exit RC*(RC<>12)                            /* RC = 0 if EOF     */
```

Differences from UNIX Pipes

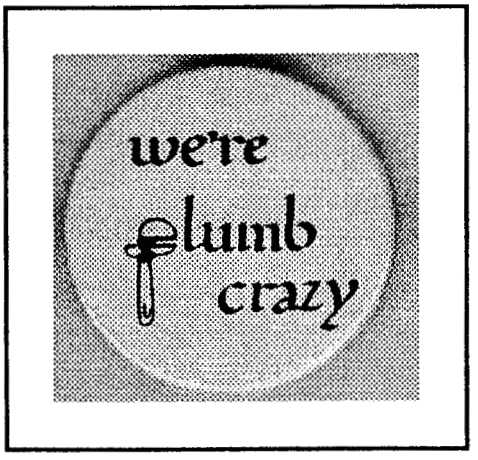
That is as many examples of using pipelines in CMS as we have time for right now. I have pointed out some of the differences between the UNIX and CMS implementations of pipelines. You may have noticed some of the others:

- As you would expect, *CMS Pipelines* is record-oriented, rather than character-oriented.
- *CMS Pipelines* implements asynchronous input, immediate commands, and dynamic reconfiguration of pipeline topology.
- *CMS Pipelines* implements multi-stream pipelines. These networks of interconnected pipelines allow selection filters to split a file into streams that are processed in different ways. The streams can then be recombined for further processing.
- Most *CMS Pipelines* stages run unbuffered; that is, they process each input record as soon as it is received and pass it on to the following stage immediately. (Of course, some pipeline stages, such as sort, must, by their nature, be buffered.) Running the stages unbuffered is necessary to allow records flowing through a multi-stream pipeline to arrive at the end in a predictable order. It can have the advantage of greatly reducing the virtual memory requirements. Thus, *CMS Pipelines* can often be used to perform operations that cannot be done with XEDIT because of virtual memory constraints.
- *CMS Pipelines* runs a pipeline only after all its stages have been specified correctly.
- *CMS Pipelines* programs can co-ordinate their progress via "commit levels" and can stop the pipeline when a program encounters an error.
- When the *CMS Pipelines* PIPE command completes, it sets its return code to the worst of the return codes set by the stages in the pipeline.

To sum up the differences between UNIX pipes and *CMS Pipelines*, let me quote a colleague of mine who said recently, "You know what I *really* miss in UNIX? *CMS Pipelines*!"

Advanced Topics

I have had time to give you only a flavor of *CMS Pipelines*. I have barely alluded to multi-stream pipelines, a very powerful extension to the basic pipeline concept with which you will want to become familiar. I also have not mentioned that *CMS Pipelines* can be run under GCS, TSO, and MUSIC. *CMS Pipelines* can now be ordered with MUSIC, and although it is not officially supported for TSO and GCS, it contains device drivers developed specifically for those environments.



III. WHY YOU SHOULD TAKE THE PLUNGE NOW

I have become convinced that any CMS user who writes REXX programs should learn to use *CMS Pipelines* as soon as possible. By the time I had been using *CMS Pipelines* for a few months, it had "saved my life" twice. In one case, I almost missed my plane to SHARE, due to a last-minute problem, but I was able to write a pipe to solve that problem before dashing out the door just in time. Then, a few weeks later, the systems in our SSI complex went into "yo-yo mode" after a service machine went into a loop creating spool files; I was finally able to get out of the problem by quickly keying in a command-line pipe to purge those files before the systems crashed again. *CMS Pipelines* can do the same sorts of things for you.

CMS Pipelines is a Powerful Application Enabler

CMS Pipelines makes CMS programmers more productive, so programs get written that would not get written without *CMS Pipelines*. (And programs that use *CMS Pipelines* are often much faster than if they had been written some other way.)

To give you a feeling for the variety of ways *CMS Pipelines* can be used, I will list a few of the ways I have used it so far myself:

- To analyze many kinds of data, including system accounting data, system performance data, and logs from service machines. Because *CMS Pipelines* is such a powerful tool, I find myself doing more thorough analyses and getting the answer down to a single page more often than I used to.
- To mend our system accounting data (more times than I care to admit).
- To charge for the use of our UNIX systems. This required writing a couple of Assembler-language filters, which turned out to be easy to do.
- To merge MVS RMF data into my VMAP ACUM files, so that I could plot the CPU utilization of our SPMODE native processor along with the utilization of the processors that the CP monitor knows about.
- To implement a full-blown service machine, with timer, IUCV, and I/O interrupts all handled without a line of Assembler code.
- To write a simple image enhancement program.
- To merge a PC database with a mainframe database.
- To augment and circumvent SES. (This seems to be a rapidly growing trend.)
- To build numerous tools to help me in my daily work, such as an XEDIT macro that understands CMS UPDATE control files and can pipe the next update onto the current file in the XEDIT ring.

I reached some sort of new plateau the first time I used *CMS Pipelines* to write a pipe to write a pipe. To celebrate that, I used *CMS Pipelines* to do this:

```
// EXEC PGM=PIPE,
//    PARM='literal Hello, World | change /World/Mom/ | console'
```

(If you put *CMS Pipelines* into an MVS loadlib, it figures out where it is and struggles on. In this case, when the console stage finds no console to write to, it uses a WTP macro.)

You Can Get Lots of Help in Learning CMS Pipelines

There are several good sources for learning *CMS Pipelines* and for getting assistance if you have questions:

***CMS Pipelines Tutorial* (GG66-3158)**: This Washington Systems Center Bulletin provides an excellent introduction to *CMS Pipelines*, and I strongly recommend it to anyone who wants to plunge into *"Pipes"*. My advice is to read this manual and work the exercises at the end of each section. Then make a conscious effort to use *CMS Pipelines* in your daily work. Before long, you will find that "pipethink" has become second nature.

***CMS Pipelines User's Guide* (SL26-0018)**: This is a rather awe-inspiring manual—300 pages without an ounce of fat on it. It contains a good tutorial and is also the reference manual and the messages manual for *CMS Pipelines*. Read the "Syntax Notation" chapter before using it as a reference manual. (The same information can be obtained by typing the command pipe help

syntax.) Incidentally, if you have only the "-00" version of this manual, you should order the "-01" version, which is substantially expanded and improved.

"Pipe help": *CMS Pipelines* provides help files that can be used with the CMS HELP command, but using the pipe help command is less painful. One especially nice feature of the pipe help command is that if you issue it with no arguments, it gives you help for the last error message that *CMS Pipelines* issued, while pipe help 1 gives you help for the one before that, and so on.

VMSHARE: The VMSHARE electronic conference has several active files dealing with *CMS Pipelines*, notably Memo Pipeline and Prob Pipeline. If you ask a *CMS Pipelines* question on VMSHARE, you will almost certainly get it answered within hours. (Inside IBM, Pipeline Forum on IBMVM is another good place to get help with *CMS Pipelines*.)

CMSPIP-L: The BITNET discussion list for *CMS Pipelines*, CMSPIP-L, is a good place for asking questions if you do not have access to VMSHARE. Several "master plumbers" participate in the list. CMSPIP-L is housed at Marist College (MARIST on BITNET or vm.marist.edu on the Internet) and at the Institute for Medical Computer Science of the University of Vienna (AWIIMC12 on EARN or awiimc12.imc.univie.ac.at on the Internet). If you can contrive to get electronic mail into BITNET/EARN or the Internet, you can subscribe to this list by sending mail to LISTSERV at one of these two sites. The body of your mail should contain the command:

```
SUBSCRIBE CMSPIP-L your name
```

The LISTSERVs at Marist and Vienna maintain archives of the discussions from the list, as well as an archive of useful pipes. You can get a list of what the nearest archive has available by sending its LISTSERV mail containing the command:

```
GET CMSPIP-L FILELIST
```

Pipedemo: Chuck Boeheim, of SLAC, has written a wonderful program called Pipedemo which "animates" a pipeline to illustrate the flow of data from stage to stage. You can download Pipedemo from Note Pipedemo on VMSHARE or order it from LISTSERV. Pipedemo is also available on the VM Workshop Tools Tape for 1991.

To use Pipedemo, you simply write a normal pipeline specification but change PIPE commands to pipedemo and change callpipe commands to rexx pdcall. Running a few *CMS Pipelines* examples through Pipedemo is an excellent way to get a deeper understanding of how pipelines work. Pipedemo can also be a big help in understanding why one of your pipelines is not working. Pipedemo is itself a pipeline, of course, and is well worth reading as an example of skillful use of *CMS Pipelines*.

CMS Pipelines Explained: This new paper by John Hartmann, the author of *CMS Pipelines*, provides many extremely useful insights into how *"Pipes"* works. I strongly recommend it.

ESA Manuals: New *CMS Pipelines* manuals will soon be issued for ESA 1.1. The new *Pipelines User's Guide* (SC24-5609) is essentially an updated version of the *Tutorial*; it reflects the changes in message numbers and the HELP facility that were required when *"Pipes"* was incorporated into CMS 8. There is also a completely new *CMS Pipelines* reference manual for CMS 8, *Pipelines Reference* (SC24-5592). However, I cannot recommend that book for any but the most casual users of *CMS Pipelines*. Even if you are running CMS 8, I suggest that you order the PRPQ manual. (Unfortunately, if you are on CMS 8, you will be stuck with help files based on the new *Reference*, unless you also order the PRPQ and load the help files from there.)

CMS Pipelines Pays Back Your Investment Quickly

CMS Pipelines is an extremely powerful facility with very rich function. There is a lot to learn. After not quite two years of using *CMS Pipelines*, I still frequently find myself saying, "Wow! I didn't know that!" or "I never thought of using it *that* way!" Although I am still a long way from having completely mastered *CMS Pipelines*, it has, nevertheless, been making my life easier since the day I installed it.

You do not have to understand all of *CMS Pipelines* to benefit from using it. The learning curve, though long, is not steep. You do not need to read the entire *User's Guide* before starting to use *CMS Pipelines*; indeed, you do not need to read the entire *Tutorial* before starting. I can guarantee that if you spend two or three hours reading the first few sections of the *Tutorial* and working the exercises, you will learn enough about *CMS Pipelines* that you will never again need to use EXECIO.

I recall that when I started learning REXX, there were several pleasant surprises:

- Programming in REXX was more fun than programming in other languages I had used.
- Because the REXX language was so powerful, I could write more programs and I could go further with them than I would have had the time for otherwise, so I ended up providing richer function and handling error conditions better.
- Even more pleasant was finding that my REXX programs tended to work correctly once I got them correct syntactically (or soon after that), which had seldom been my experience with programs I had written in other languages. The structure of the REXX language was disciplining my thinking so that I was programming not only more easily but also better.

I see these same effects even more strongly when I combine REXX with *CMS Pipelines*:

- Programming with *"Pipes"* is even more fun. It has restored my delight in CMS. I cannot imagine going back to not having *CMS Pipelines*. As my colleague Serge Goldstein was heard to exclaim a few weeks after we got it, "I can't do *anything* without *Pipes*!"
- The power of *CMS Pipelines* allows me to write programs that I would not have found the time (or the stamina) to write before. I am writing more programs and giving them richer function.
- My *CMS Pipelines* programs have fewer bugs. The processes of applying "pipethink" and of visualizing the flow of data through my pipelines make me a better programmer.

If you will give it a try, I think you will find, as I have, that *CMS Pipelines* is a tool for unclogging the brain.

THE IMPLICATIONS OF MULTIMEDIA FOR TRAINING IN THE '90S

P. JOSEPH VERTUCCI
THE ALIVE CENTER OF AMERICA

File Copy

Session:

The Implications of Multimedia for Training in the '90's

Dr. P. Joseph Vertucci

Chief Executive Officer

ALIVE Centers of America, Inc.

Fairlawn, Ohio

IASA Annual Conference

Dallas, Texas

May 31 - June 3, 1992

Television has had a dramatic impact on the adult population. By the time a person graduates from high school, they have been exposed to over 20,000 hours of television, that is high impact visuals and audio. In contrast, that same person has been exposed to approximately 14,000 hours of classroom instruction. Multimedia brings the impact of television to the training environment.

While motivating adult learners is a very complex issue, part of the solution resides in the application of adult learning principles to multimedia instruction. Documented research shows that when interactive multimedia is employed over every other style, stand up instruction, computer based training (CBT), or video based training, learners prefer interactive multimedia in 97% of the cases. That means almost 33 to 1 prefer interactive multimedia to other training approaches. Preference of learning approach also equates to increased performance and results. Documented studies show that as more senses are incorporated into the learning environment, retention increases dramatically. Traditional computer based training, that is reading a computer screen, is very similar in task to reading a book. Documented research shows that a person remembers only 10% of what is read, 20% of what is heard, 30% of what is seen, 50% of what is seen and heard and 80% of what is experienced. Interactive multimedia simulates experience to such an extent that it has been documented in over thirty research studies to replace the actual experience.

While the issue of individual motivation is complex, multimedia has demonstrated through numerous studies to be a major factor for increased performance, reduced time on task and increased employee productivity. Multimedia programs ensure student motivation and successful program completion. Well-known Adult Learning Principles enhance and complement multimedia in this respect.

Adult Learning Principles

The Adult Learning Principles include the following:

- Project vs. subject centered focus
- Immediate application of learning
- Capitalizing on learner's previous experience
- Learner vs instructor centered focus
- Self-directed vs dependent focus
- Active participation in the learning process
- Whole-part-whole sequence of learning
- Association of material
- Integrated thinking
- Recognition of individual learning rates and styles
- Maximizing time on task
- Regular checking of understanding
- Appropriate and meaningful instructional cues

• Feedback on results with positive reinforcement

The following examples illustrate how these principles can be incorporated into training applications.

Project vs. subject centered focus

Adults are problem oriented, thus training must be problem centered. Classroom training is predominantly subject centered. By focusing on problems, adults are challenged to use their experience in finding solutions to problems.

We have created training programs to teach high school coaches how to accurately diagnose knee injuries; to teach sales representatives for orthopedic implant manufacturers how to enter product orders into the company's computerized order-entry system; and to teach retail store employees how to recognize potential shoplifters. These are just a few examples that illustrate how multimedia learning applications are being used today.

Immediate application of learning

Training programs can allow immediate application to the learner's work environment. In this respect, the programs may be regarded as a modern approach to the older but successful on-the-job-training concept. Whether the training is aimed at teaching product knowledge of caskets or orthopedic implants, the learner can employ the skill and knowledge just acquired to enhance job performance.

Capitalizing on learner's previous experience

By employing pretest and branching techniques, training programs recognize the value of the learner's previous experience. Such programs are designed to permit learners to progress at their own

pace, to focus on material they do not know and to bypass material which is already known.

Learner vs. instructor centered focus

Training programs can be designed to focus on the learner. The learner can select the subject, topics within the subject and pace. By using built-in navigational controls, the learner can move forward or backward, access a glossary or bibliography or review course maps.

Self-directed vs. dependant focus

Learning is self-directed. The learner is not dependent on a group pace, but controls his own pace. We have developed a proprietary menu system which allows the learner to access information quickly and effortlessly.

Active participation in the learning process

Training programs can involve the user. The learner is an active participant in the learning process. Regardless of whether the subject deals with executive, management, sales, industrial or medical training, the learner makes decisions and is branched to different sections of the course based on these decisions.

Whole-part-whole sequence of learning

Programs can relate information into context. The learner is introduced to concepts using whole-part-whole sequencing. The learner's ability to quickly grasp the material is substantially increased by first learning a small concept, then relating that concept to the whole.

Association of material

Learning generally does not exist in isolated settings and frequently the same material can be used for multiple applications. Product knowledge is very much related to both sales and technical training. More often a logical association exists between information and its use in various other parts of an enterprise. This information is a valuable corporate asset. Our programs recognize the investment involved in capturing and maintaining this information by organizing this material modularly. Modularity minimizes the expense of updating information or of extracting this information for use in other applications.

Integrated thinking

Training programs can employ integrated/holistic thinking. Through navigation and mapping, individual learners can determine their current position and assess their progress throughout the course.

Recognition of individual learning rates and styles

Training programs can be designed to recognize that individuals have different learning styles as well as learning rates. Programs offer visual, audio, and conceptual stimulus. Programs are designed to stimulate the learner by rewarding correct answers to exercises and quizzes. By the same token, a benevolent, non-judgmental response is provided in response to incorrect answers.

Maximizing time on task

Studies have shown a 30% to 60% reduction of time on task using interactive multimedia. Learners proceed at their own pace and access information as needed. Learners move forward and backward

through the material in accordance with their personal style and educational needs.

Regular checking of understanding

Our training programs incorporate periodic, regular checks of the learner's comprehension. Programs can be customized by the training administrator to require 100% mastery, or any other specified level of accomplishment. This can be accomplished by employing remediation techniques that return the learner to material not mastered, feedback on responses to reinforce success, and other appropriate learning strategies.

Appropriate and meaningful instructional cues

Adult learners require appropriate and meaningful instructional cues. Our programs employ icons, images and audio feedback appropriate and sensitive to the audience.

Feedback on results with positive reinforcement

In addition to the audio and visual feedback used to reinforce the learner's understanding of material, learners are remediated into appropriate course material to further enhance the learning experience.

Overall Program Structure

In addition to utilizing the Adult Learning Principles in designing multimedia training applications, there are specific overall design concerns that can effect learner motivation and retention. These include:

- Skill and Drill: the repetition of an exercise insuring the learner's understanding and proficiency. This is usually followed by a self-test.
- Tutorial: personalized company assistance promoting understanding of a particular concept; one-on-one is the best and most expedient method.

- Gaming: Know Your Product game, a method of involving competition under specified rules; and,
- Simulation: the method of teaching allowing a learner to manipulate a particular environment.

Multimedia technology uses numerous learning aids imbedded in programs to facilitate learning. These include:

- Icons and buttons to permit the student to navigate through the course. We use forward and backward navigational tools, a looping tool, ability to return to the main menu, help screens and other features as required by the content;
- Course maps that permit the learner to assess where they are in the course;
- Glossary and bibliographies to permit the learner to access definition of terms;
- Remediation programming to loop the learner through material that has not been mastered;
- Randomized question pools for mastery tests;
- Help features that include how the system operates;
- Pretests to assess the learner's current level of understanding and knowledge with branching on results to permit the learner to move quickly through material that they have previously mastered;
- Periodic exercises to verify learner retention;
- Comprehensive post tests to measure performance and mastery;
- Tracking of learner progress through use of a database manager;
- Bookmarking capabilities to permit the learner to leave a program and return at a later time to exactly where they left;

- Course objectives stated at the beginning of the course and at the beginning of each new section;
- A summary screen that lets the learner review material before taking the mastery test.

Additional features are added to courses as required. Each multimedia course can be designed to employ these key concepts while also addressing the specific requirements of the content. In this respect, content can be made easy to understand by incorporating the following features:

- High level of student motivation by using graphics, digital effects, audio and text in appropriate educational strategies;
- Random visual and audio accessibility;
- Consistency of instruction that guarantees all students receive a high quality presentation;
- Dual track, stereo audio with music and professional voice over as required;
- Subject mastery that can be adjusted by the instructor. Mastery levels can be set at 100% if required and can be modified by the instructor as appropriate;
- Feedback and reinforcement using immediate and automatic feedback and reinforcement including visual, graphic and audio;
- Zoom feature to let the learner examine in detail material and concepts;
- Self-paced learning permitting each learner to master the course at their own learning rate; and,
- Round the clock availability allowing the learner to take the material on demand.

Further educational concerns include graphic quality, testing and evaluation, on-the-job applicability and obsolescence issues.

Graphics Characteristics

Multimedia training programs employing AVC use VGA graphics resolution, but with enhanced resolution to 8 bit graphics with 256 colors. The resolution is 320x480 lines of resolution. Picture and image clarity and quality is unmatched. Even though the images are superb, the image size is approximately one tenth of the size of comparable industry standard images. Because of this, the learning process benefits in several significant ways. First, we are able to provide close representation of a company's products on the computer image. Because of the high image quality, the learner does not experience a believability problem. We are also able to provide greater detail that is further exploited to enhance the learning process. Secondly, because we can support stereo audio, the learner has multi-sensory exposure further enhancing the learning environment. The combination of high quality image and low memory requirements means that we can pack our programs with more images and use less memory. We use a five to one ratio, that is five images for one tiff or targa formatted image. More visuals means greater representation of products and enhanced learning.

Student Testing and Evaluation Procedures

Students are tested against the behavioral learning objectives prescribed by the program. We recommend that mastery levels be set at 100%. Because of the inherent capabilities of multimedia, mastery at 100% is still accomplished in less time than with conventional approaches. We employ several testing strategies as previously documented. Student performance is tracked by the system. The instructor has the ability to review student performance and recommend additional strategies to assure success. Testing is accomplished in a variety of approaches. These include:

- Exercises after each section, chapter or major content function;
- Module tests from a randomized pool of questions; and

- Mastery test covering all major course objectives.

Final mastery is compared to initial pretest results to measure overall program effectiveness.

On-the-Job Application of Learning

Multimedia training programs can incorporate not only skill and drill and tutorial strategies, but also game and simulation strategies. Our programs can challenge the learner to apply the principles, skills and concepts that have been learned in realistic situations. This takes the multimedia user where CBT programs could never take them, into the realm of experience. Multimedia with realistic image, voice, noise and sound offer capabilities that no other methodology can approach. The difference between interactive multimedia and computer based training is similar to being in the driver's seat of a race car (interactive multimedia) versus reading about the experience (computer based training). We have known that experience is the best teacher. Now with interactive multimedia, we can afford to put the learner in the driver's seat!

How to Upgrade Proposed Software

Images, audio as well as story files can be upgraded and distributed on a company's current network. When an image or audio file or story is created, updated or modified, it can be automatically loaded onto the distributed system via modem, network software or floppy disk. The file is copied onto the local system. The new file replaces the old one and the user is virtually shielded from the process. There are no expensive charges, no complicated re-editing problems and no reliance on outside production services. Cost for upgrading can be handled on an hourly rate or on an as needed basis.

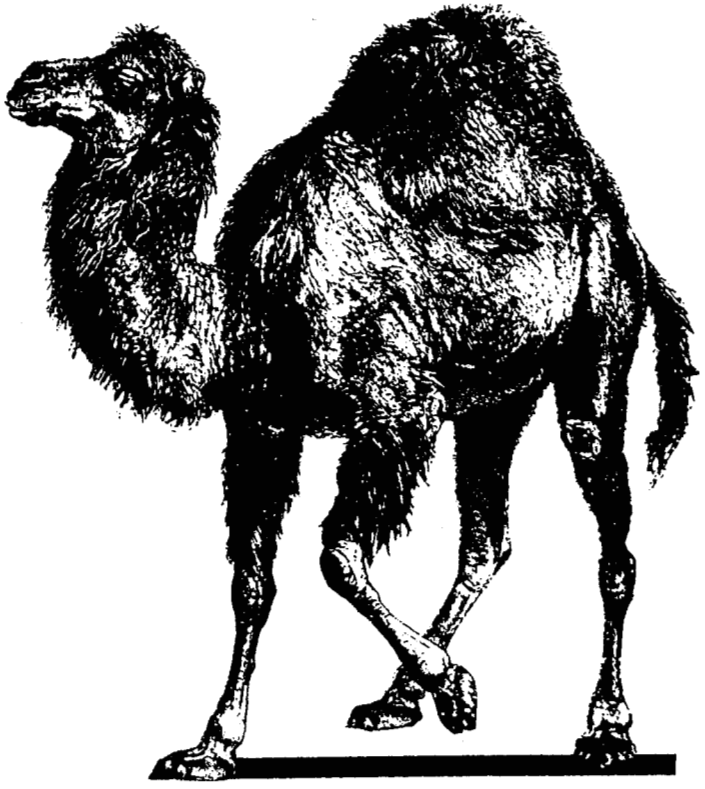
REXX, PERL, AND VISUAL BASIC

BEBO WHITE
STANFORD LINEAR ACCELERATOR CENTER

REXX

and (not vs.)

Perl



Bebo White
SLAC
REXX Symposium
Annapolis, MD
May 4, 1992

M.F. COWLISHAW

THE REXX LANGUAGE

A PRACTICAL APPROACH TO PROGRAMMING

SECOND EDITION

UNIX Programming

Programming

perl

Larry Wall and Randal L. Schwartz

O'Reilly & Associates, Inc.

Caveats

- I am a REXX bigot, but the cards weren't stacked against Perl; I am not a Perl expert (much less bigot);
- the most important thing about comparing these languages is determining how well they support their environment; this is largely implementation-dependent;
- I have never used REXX and Perl on the same system;
- this talk started out as "REXX vs. Perl" - but they really aren't competitors;
- I like Perl; it makes Unix far more "approachable" for me;
- I think that some of the features of Perl can contribute to the development of REXX;

REXX and Perl Have a Similar Background

BOTH-

- were developed largely by an single individual;
- were developed for a particular operating system and strongly utilize features of that system;
- have their roots in a "popular" high level programming language;
- have "natural typing";
- emphasize string processing;
- provide a strong built-in function library;
- emphasize readability and an understandable block structure;
- have useful debugging capabilities;

Perl Names

BLATZ - a filename or directory "handle"
$BLATZ - a scalar variable
@BLATZ - a normal array
%BLATZ - an associative array
&BLATZ - a subprogram
***BLATZ** - everything named BLATZ

- does not harken back to EXEC, EXEC2 or Batch;
- does increase the readability /understandability of a program;
- allows program entities to be associated in a subtle way;
- eliminates part of a "style controversy";

Perl Lists

- an ordered list of scalars;
- can be like an array, or "user-defined types";
- can be fully dynamic;
- incorporates some of the capabilities of Parse; for example -
  - **@ARGV** consists of **$ARGV[0]** to **$ARGV[$#ARGV]**
  - **($name,$address) = split(/:/,<NAMES>)**

Perl "Gotchas"
(for REXX users)

- the default value of a variable is the null string;
- a value is TRUE if it isn't the null string, 0 or "0";
- there are different comparison operators for numerics and strings;
- some operators are borrowed from sed, awk, and various Unix utilities;

Some General Conclusions

- REXX is easier to learn and more readable; REXX is more accessible to a greater audience;
- Perl's syntax is harder to learn and read (unless you're a big C fan); appeals to "hackers";
- Perl is an excellent interpreted shell script/systems language, but not a common embedded macro language for Unix;
- Perl is more consistent with a "Unix mindset" than REXX;
- Some Perl operations are very arcane (e.g., ++i, i++);
- Perl has many more redundancies than REXX;
- Perl has better support for aggregate types than REXX; both languages lack support for non-trivial datatypes;
- Perl is more compact for some things (e.g., string processing); compactness <----> safety?

- Perl has an extensive collection of pattern matching operators; REXX relies more heavily on PARSE;
- Perl has built-in file feature operators; where REXX relies on OS;
- Perl has a package mechanism which REXX lacks;
- REXX is more extensible than Perl;

Can REXX Learn From Perl?

- Associative arrays are very "CMS-like"; can be weakly implemented by the REXX ABBREV;
- Perl lists allow for a *for each* construct;
- Perl makes extensive use of the <STDIN>, <STDOUT>, <STDERR> streams; REXX LINEIN, LINEOUT capabilities not always implemented;
- PIPELINES can add some Perl capabilities to REXX;

REXX APPLICATIONS IN AUTOMATED OPERATIONS

PETE ZYBRICK
FUTURESYS, INC.

REXX Applications in Automated Operations

Pete Zybrick

FutureSys, Inc.
20 Dogwood Trail
Kinnelon, NJ 07405
(201) 492-2777

I. Overview

1. What is Automated Operations? The progressive minimization of computer operator intervention by
    1. Replacing the need for intervention whenever possible by the design and implementation of hardware/software problem determination and correction processes.
    2. Increase problem determination and correction efficiency by filtering and combining only the critical system status information, eliminating redundant and trivial information.

2. Automation Types
    1. Reactive - Event/Response
    2. Proactive - Question/Answer
    3. Administrative/Management

II. Why use REXX

1. Good
    1. PARSE Instruction, especially Literal String
    2. Relatively simple to use/debug/maintain
    3. Relatively easy to create structured code
    4. Function libraries

2. Bad
    1. Simplicity has been oversold by vendors
    2. Unskilled programmers can write bad code in any language
    3. Simplicity masks potential errors
    4. CLIST programmers rarely take advantage of REXX features
    5. Reliance on environment for global variables, poor variable sharing between procedures

III. Features and AO Application

1. Subcom (Host Command Environment Table) - Creating an Environment

    1. Advantages
        1. Speed - commands are directly targeted
        2. No changes to REXX itself are required

    2. Disadvantages
        1. Development - must be written in lower level language, initialization exit configured (MVS) or DLL created (OS/2)
        2. Programmer must remember to use ADDRESS both initially and when switching environments (ie. ADDRESS MVS "EXECIO..." and ADDRESS NETVIEW "GETMLINE..."

2. Shared Variable Interface

    1. Advantages
        1. Large blocks of variables can be created with one command/function
        2. Same basic processing sequence and control block structure on different platforms

    2. Disadvantages
        1. Uses more storage than the stack
        2. Programmers usually forget to DROP, possibly causing storage problems

3. Function Libraries

    1. Advantages
        1. Speed development time and consistency
        2. Can be written in lower level language for improved performance
        3. Can accept and return very large plists
        4. Third party vendors and SHARE

    2. Disadvantages
        1. Definition of requirements
        2. Someone has to write/maintain the functions
        3. Will anyone know they are there?

4. External Programs

    1. Advantages
        1. Can be REXX or load module. Load modules can use the Shared Variable Interface
        2. Interface to external products
        3. Command response/screen capture

    2. Disadvantages
        1. Search time (for load modules, faster to use Subcom and ADDRESS)
        2. Poor global variable handling forces large values to be passed/duplicated between programs

IV. Suggested Methods

Objectives:

1. Keep it simple
2. Minimize redundant coding/maintenance

1. Centralized Routines
    1. Objectives
        1. Maximize the capabilities of the most skilled programmers to produce common 'black box' routines to simplify the most difficult tasks
        2. Maintenance - if the program is broken, it is fixed in one place

    2. Example: NetView returns command responses asynchronously, if at all. Even experienced programmers can have a conceptual problem with async events. Create an external function to serialize command execution/response under NetView, returning the responses on the stack.

```
/* REXX - LINKSTN */
call stkmsgs    ,
     "D NET,ID=someappl,E"  ,     ,
          "IST097I IST075I"  , "IST314I"
. . .  read from stack and process messages  . . .
exit


/* REXX - STKMSGS */
parse arg CmdText , TrapMsgs , EndMsg
"TRAP AND SUPPRESS MESSAGES" TrapMsgs
CmdText
"WAIT 5 SECONDS FOR MESSAGES"
"MSGREAD"
getresps: do while 'EVENT'() = "M"
     "GETMSIZE MAXMLWTO"
     getmlwto: do mlcnt = 1 to maxmlwto
          "GETMLINE CURML" mlcnt
          queue curml
          if 'WORD'(curml,1) = EndMsg then leave
        end /* getmlwto */
     "WAIT CONTINUE"
     "MSGREAD"
   end /* getresps */
return /* stkmsgs */
```

2. Literal String Parsing

Objectives:

1. Parse messages based on text fields to extract variable-length values.

Example: The NetView TSOUSER command describes the status of a TSO user. Display the TSO (application name) and LU of a particular user.

a. Command Format:

"TSOUSER tsologonid"

b. Output:

IST097I DISPLAY ACCEPTED
IST075I VTAM DISPLAY - NODE TYPE = TSO USERID
IST486I NAME=TSOPJZ, STATUS=ACTIV,DESIRED...
IST576I TSO TRACE=OFF
IST262I APPLNAME=TSOA, STATUS = ACTIV
IST262I LUNAME=A01T1234, STATUS=ACTIV
IST314I END

c. Program:

```
/* REXX */
parse upper arg tsoid .
call 'STKMSGS' "TSOUSER' tsoid , "IST097I IST075I",  ,
      "IST314I"
do queued()
     parse pull MsgID MsgText
     if MsgID = "IST262I" then do
          parse var MsgText hdr"= "name", STATUS= "status
          if hdr = "APPLNAME" then do
            TSOName = name
            TSOStatus = status
           end
          if hdr = "LUNAME" then do
            LUName = name
            LUStatus = status
           end
       end
  end
```

3. Global Variables - Logical/Stem/Associative Arrays

Objectives:

1. Simplify the status setting and determination of a particular subsystem
2. Can be used to drive a graphic status panel (ie. subsystem name in green if up, yellow if brought down cleanly, red if crashed, etc.)

Example: Set status variables for group of CICS's. Retain the time each CICS was last brought up or down. There is nothing 'CICS-unique' about this example - any subsystem on any platform can be substituted (just the type of global variable handling would have to change).

a. Executed during System Initialization

```
/* REXX */
AllCICS = "PROD01 PROD02 ... PRODxx"
"GLOBALV PUTC ALLCICS"
CICSUp. = 0
do until AllCICS = ""
      parse var AllCICS CurrCICS AllCICS
      "GLOBALV PUTC CICSUP."CurrCICS
      call 'STRTCICS' CurrCICS
  end
```

b. Start a given CICS region (ie. STRTCICS PROD01)

```
/* REXX */
parse upper arg CurrCICS
  .  .  .
/* Current CICS brought up OK */
CICSUp.CurrCICS = 1
CICSDtTm.CurrCICS = 'DATE'("U") 'TIME'()
"GLOBALV PUTC CICSUP."CurrCICS "CICSDTTM."CurrCICS
```

c. Stop a given CICS region (ie. STOPCICS PROD01)

```
/* REXX */
parse upper arg CurrCICS
  .  .  .
/* Current CICS brought down OK */
CICSUp.CurrCICS = 0
CICSDtTm.CurrCICS = 'DATE'("U") 'TIME'()
CICSWhyDown.CurrCICS = "Stopped by" 'OP'()
"GLOBALV PUTC CICSUP."CurrCICS "CICSDTTM."CurrCICS ,
      "CICSWHYDOWN."CurrCICS
```

d. Restart CICS due to some error (ie. RSTCICS PROD01, probably called from NetView Message Automation Table after hit on abend message)

```
/* REXX */
parse upper arg CurrCICS AbendInfo
CICSUp.CurrCICS = 0
CICSDtTm.CurrCICS = 'DATE'("U") 'TIME'()
CICSWhyDown.CurrCICS = "Abended:" AbendInfo
"GLOBALV PUTC CICSUP."CurrCICS "CICSDTTM."CurrCICS ,
      "CICSWHYDOWN."CurrCICS
/* Restart Current CICS */
  . . .
```

e. Status of CICS regions

```
"GLOBALV GETC ALLCICS"
do until AllCICS = ""
      "GLOBALV GETC CICSUP."CurrCICS  ,
            "CICSDTTM."CurrCICS "CICSWHYDOWN."CurrCICS
      select
            when CICSUP.CurrCICS then
                  say "UP  " CurrCICS
            when ^CICSUp.CurrCICS  &
                     CICSWhyDown.CurrCICS <> "" then
                  say "DOWN" CurrCICS  CICSWhyDown.CurrCICS
            when ^CICSUp.CurrCICS  &
                     CICSWhyDown.CurrCICS = "" then
                  say "DOWN" CurrCICS "Never Started"
            otherwise say  "Unknown" CurrCICS
         end
   end
```

4. Log Processing

Objectives:

1. Perform filtering and summary information against log files (ie. MVS system log, VM operator console log, NetView log, etc.).

Example 1: Create a subset of a large log file. Scan an entire log and write only VTAM messages to another dataset.

```
/* REXX */
/* Scan a log and filter messages */
/* Delete/Erase the Output File */
/* if MVS/NetView, ALLOCATE here */
ReadLoop: do until ExecioRC <> 0
      "EXECIO *nnnnn DISKR <InputFile>"
      ExecioRC = rc
      PullLoop: do queued()
            /* Message ID starts in 10 */
            /* Save only VTAM (IST) Messages */
            parse pull . 10 MsgID 13 1 MsgRec
            if MsgID = "IST" then queue MsgRec
        end /* PullLoop */
      /* if any matches on IST then write */
      if queued() > 0 then
            "EXECIO" queued() "DISKW <OutputFile>"
  end /* ReadLoop */
/* Close files here */
```

Example 2: Display a summary of message occurances

```
/* REXX */
/* Scan a log and sum by message id */

/* if MVS/NetView, ALLOCATE here */

UniqueMsg = ""
GotMsg. = 0
SumMsg. = 0
TotMsgs = 0
ReadLoop: do until ExecioRC <> 0
      "EXECIO nnnnn DISKR ...."
      ExecioRC = rc
      TotMsgs = TotMsgs + queued()
      PullLoop: do queued()
            /* Message ID is in cols 10-19 */
            parse pull . 10 MsgID 20 .
            SumMsg.MsgID = SumMsg.MsgID + 1
            if ^GotMsg.MsgID then do
                  UniqueMsg = UniqueMsg || MsgID" "
                  GotMsg.MsgID = 1
               end
         end /* PullLoop */
   end  /* ReadLoop */

/* Close the log file here */

/* Display Msgid # % */
do until UniqueMsg = ""
      parse var UniqueMsg MsgID UniqueMsg
      Pct = 100 * (SumMsg.MsgID/TotMsgs)
      say 'LEFT'(MsgID,12) 'RIGHT'(SumMsg.MsgID,8) ,
        'FORMAT'(Pct,3,0) || "%"
   end
```

5. Screen Image Parsing

Objectives:

1. Parse screen images to isolate critical information

Example: The following screen image was trapped into one variable, SCREEN. Extract the CPU utilization for the displayed applications.

```
          Performance Stuff
before
before
App          Util
= = = = = = = =|= = = = = = = =| ...
ME           22
YOU          15
= = = = = = = =|= = = = = = = =| ...
after
after
```

```
/* REXX */
GotHdr = 0
do while Screen <> ""
      parse var Screen 1 Line 81 Screen
      parse var Line 1 Hdr 8 1 SubSys 10 UtilCPU 15 .
      select
            when ^GotHdr & Hdr = '= = = = = = = =' then
                  GotHdr = 1
            when GotHdr & Hdr = '= = = = = = = =' then
                  leave
            when GotHdr then say SubSys UtilCPU
            otherwise nop  /* 'Before' stuff */
        end
  end
```

6. Table Driven Automation

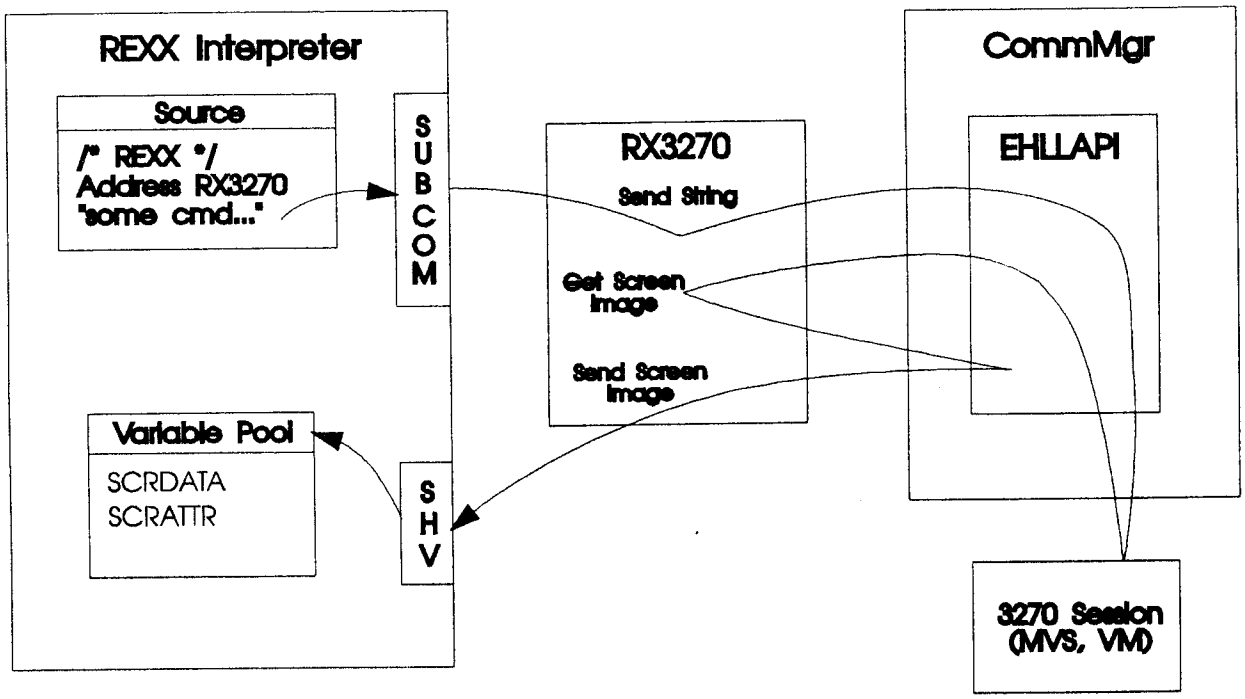
7. Testing and Simulation

8. Selective/Blanket Restart Enable/Disable

9. System/NCP/etc. Generation File
   Scanning/Parsing/Comparing

V. OS/2 CommMgr as an AO Tool

1. REXX is supplied with OS/2
2. CommMgr uses EHLLAPI to allow session management, namely:
   1. Issuing text strings to a 3270 session
   2. Retrieving 3270 screen images
3. REXX API's support Environments, Shared Variable Interface, Function Libs
4. REXX3270 tool:



VI. Indirect Benefits

1. Table driven status/recovery routines allow ownership of resources to be rapidly moved to alleviate performance/failure considerations

2. Disaster Recovery
    1. A 'disaster' table can exist which contains only critical devices mapped to the ownership of critical systems
    2. A 'snapshot' program can display/query critical system components/values on a periodic basis and save this info into a table. After and disaster and recovery, a display/query job can be run to verify critical component availability and differences.

3. Job Automation. Experience/confidence gained during AO implementation can be extended to automating nightly job cycles, replacing JCL with REXX to allow for more intelligent and automatic job monitoring/restart/correction.

VII. The Future...

1. Dynamic Configuration Management. Access external matrix switches to reconfigure devices from one system to another 'on the fly', both for performance and failure recovery purposes.

2. Enterprise Automation

3. DMS?

4. NetWare?

5. ???

The programs/ideas in this document are in the public domain. Use them in any manner. Most were written to run under NetView and/or MVS, but should, with minor changes, run anywhere. Be careful - I either clipped them out of larger programs or wrote them from memory based on projects I worked on in the past - typos are probable. More importantly, to keep things concise, I removed all the error handling code. If you have any questions, feel free to call/fax me at (201) 492-2777. I'm always willing to help and curious to hear how different sites implement automated operations.

Thanks,
Pete Zybrick

)ebbie Audette
MS America Ltd.
'lymouth Meeting, PA 19462-0905
!15-834-4623

lim Babka
BM GO9/20
?.O. Box 6
Endicott, NY 13760
607-752-1613
)abka@gdlvm7.vnet.ibm.com

Rick Berge
/MSG, INC.
1604 Spring Hill Road
Vienna, VA 22182-2224
703-506-0500
fdb@alumni.caltech.edu

Gurnie Bowden
2704 Loyola Lane
Austin, TX 78727

Gordon Callan
2040 Wooden Glen Way
Los Altos, CA 94024

Jerry Campbell
Amoco Corporation
501 WestLake Park Blvd.
P.O. Box 3092
Houston, Texas 77253
713-556-7036
ZSLCIZ@HOU.AMOCO.COM

Steven Carroll
EBASCO
89-20 55th Ave. Apt. 7A
Elmherst, NY 11373
718-446-8973

Anders Christensen
University of Trondheim
Computing Center
Trondheim, Norway
+47-7-593004
anders@lise.unit.no

Dean Clark
InFonet Corp.
11700 Montgomery Rd.
Beltsville, MD 20705
301-937-0500

Creswell Cole
Amdahl
1350 East Arques Ave.MS205
Sunnyvale, CA
408-746-4877
chappy@pswd.amdahl.com

Mike Cowlishaw
IBM United Kingdom Labs., Ltd
Hursley Park
Winchester, Hampshire S021 2JN
UK

Cathie Dager
SLAC MS97
P.O. Box 4349
Stanford, CA 94309
415-926-2904
cathie@slacvm.slac.stanford.edu

Charles Daney
Quercus Systems
P.O. Box 2157
Saratoga, CA 95070-0157

Chip Davis
Amdahl Corp.
10420 Little Patuxent Parkway
Columbia, MD 21044-3598
CHIP.DAVIS@AMAIL.AMDAHL.COM.

Steve Demion
30 Sawmill River Rd.
MS HOB13
Hawthorne, NY 10532

Kenneth R. Down
ORACLE
2075 Sutter St. #523
San Francisco, CA 94115-3131
415-506-2778
Kdown@ORACLE.COM

Deryl Duncan
IBM
1902 Willowcrest
Denton, TX 76205
214-280-6739

Larry Dusold,
Telecom. & Sci. Comp. Support
FDA/C F.S.A. N.
200 C St. S.W.
Washington, DC 20204
Bitnet: LRD@FDACFSAN

Frank Esposito
P.O. Box 140125
Brooklyn, NY 11214-0002
718-946-6148
FPEOC2CUVMB

Carl Feinberg
Relational Architects
33 Newark Street
Hoboken, NJ 07030
201-420-0400

Janice Fitch
Electronic Data Systems
3044 West Grand Boulevard
GM Building Room 7-119
Detroit, MI 48202
313-556-4451

Bob Flores
CIA
Rm 2V29
Washington, DC 20505
703-874-5174
RAF4@PSUVM.PSU.EDU

Nancy Flynn
Computer Sciences Corp.
Applied Technology Division
16511 Space Center Blvd.
Houston, TX 77058
713-280-2434

Dave Fraatz
3M Center
Bldg. 220-3W-01
St. Paul, MN 55144

Forrest Garnett
IBM
2500 Huston Court
Morgan Hill, CA 95037
408-997-4619
Garnett@sisvm28.vnet.ibm.com.

Kathleen Garvey
Borland
1800 Green Hills Road
Scotts Valley, CA 95067-0001

Eric Giguere
University of Waterloo
Computer Systems Group
Waterloo, Ontario, Canada N2L 361

Gabe Goldberg
VM Systems Group, Inc.
1604 Spring Road
Vienna, VA 22182-2224

Anthony Green
Ruddock & Assoc.
74 McGill Street
Toronto, Ontario
Canada M5B 1H2
416-340-0887
green@roboco.vocp

Linda Green
IBM
G93/6C12
P.O. Box 6
Endicott, NY 13760

Billy Guthrie
SIDNEY
5727 Holly Hill Circle
Dallas, TX 75231
214-750-8112

Rainer Hauser
IBM Research Div. Zurich
Saumerstrasse 4
CH-8803 Ruschlikon
Switzerland
rig@zyrucg.ibm.com

Paul Heaney
Delphi Software, Ltd.
Fleming Court, Flemming Place
Mespil Road
Dublin 4, Ireland
Ireland 602877

Earl Hodil
Chicago-Soft, Ltd.
420 S. Winsome Ct.
Lake Mary, Fl 32746
407-834-7530

Marc Irvin
MVI
100-01 Hope Street
Stamford, CT 06906
203-327-4361
498-9279pmcimail.com

Don Jones
IBM
1902 Willowcrest
Denton, TX 76205
214-280-6458

Jeff Karpinski
Towers Perrin
Centre Square East
1500 Market St.
Philadelphia, Pa 19102-4790
215-246-6003

Andrew J. Katz
IBM
13 Dufief Ct.
Gaithersburg, MD 20878
301-571-7842

Thomas W. Kema
Bell South Telecom.
Rm S-304
1876 Data Drive
Birmingham, AL 35124
205-988-1504

William Kohlstrom
Kohl International
400 N. Fourth St., Suite 1012
St, Louis, MO 63102-2636
410-664-1961
MCI333-1002

Terry Kong
National Library of Medicine
Bldg. 38A Rm. B1W08G
8600 Rockville Pike
Bethesda, MD 20894

Lee Krystek
Boole and Babbage
8000 Commerce Parkway
Mount Royal, NJ 08054

Jimmy Lee
Metropolitan Life
One Madison Ave. Area 9-C
New York, NY 10010-3690

Linda Littleton
214 Computer Bldg.
Penn State University
Univ. Park, PA 16802

Terry Masemore
IBM
800 N. Frederick
Gaithersburg, MD 20879
301-240-7607

David McAnally
Motorola, Inc.
Corporporate Computer Services
8220 E. Roosevelt R7142
Scottsdale, AZ 85257
602-441-5296
ACUS02@WACCVM.CORP.MOT.COM

Glenn McPeters
RD1, Box 6390
Underhill, VT 05489

Pat Meehan
IBM Ireland
ECFORMS Dev.Team
4 Burlington Road
Dublin 4, Ireland
353-1-603744
MEEHANP@DUBVM1.VNET.IBM.COM

John Milburn
IBM
800 N. Frederick
Gaithersburg, MD 20879
301-240-7275

Neil Milsted
iX Corporation
575 W. Madison St. N0. 3610
Chicago, IL 60606
312-902-2149
NFNM@WRKGRPP.COM

Stan Murawski
635 S. 16th St.
San Jose, CA 95112-2372
408-288-6759
CISMAIL 70444.55

Chuck Nelson
830 McCandless
Wichita, KS 67230

Edward G. Nilges
P.O. Box 16
Kingston, NJ 08528
egnilges@pucc

Eric Nothman
8417 Fenway Rd.
West Bethesda, MD 20817

Robert O'Hara
Lotus Development Corp.
One Rogers Street
Cambridge, MA 02142

F. Scott Ophof
Consultant
269 Hall Avenue
Windsor, Ontario
Canada N9A 2L5
519-253-7534
ophof@server.uwindsor.ca

Walter Pachl
IBM Austria
Lassallestrasse 1
A-1020 Vienna, Austria
PACHL @ VABVM1.VNET.IBM.COM

Steve Price
IBM
GO9/20M
P.O. Box 6
Endicott, NY 13760
607-754-9653
pricesg@gdlvm7.vnet.ibm.com

Brian Rodbell
RMS TECHNOLOGIES, INC.
NASA GSFC Code 520.9
Greenbelt, MD 20771
301-286-2098
Z8blrAsspa.gsfc.nasa.gov

Ed Root III
Motorola, Inc.
Corporate Computer Services
8220 East Roosevelt R7142
Scottsdale, AZ 85257

Roger Root
2953 Tillinghast Trail
Raleigh, NC 27613
919-846-7014
compuserve?

Anthony Rudd
Robert-Schumannstrasse IIA
W-8510 Fuerth
Germany

Albert Sayers
MILBANK, TWEED
12 Woodland Dr.
Port Washington, NY 11050
212-530-8920

Kurt D. Scherer
CSC
4522 Bennion Rd.
Silver Spring, MD 20906
301-794-1030

Paul Schobert
IBM
1920 Willowcrest
Denton, TX 76205

Sally Schor
IBM
1920 Willowcrest
Denton TX 76205
214-280-6487
mks@cbmvax.commodore.com

Gary Schramm
4912 Green Road
Raleigh, NC 27604

Colleen K. Seine
IBM
5601 Executive Blvd.
Irving, TX 75038

David Shriver
IBM MS 01-03-50
5 W. Kirkwood Blvd.
Roanoke, TX 76299-0001

Michael Sinz
Commodore-Amiga, Inc
1200 Wilson Drive
West Chester, PA 19380
215-431-9382
mks@cbmvax.commodore.com

Steve Siperas
3M-HIS
100 Barnes Road
Wallingford, CT06906
Siperas@hsi.com

Phil Smith
VM SYSTEMS GROUP, INC.
1604 Spring Hill Road
Vienna, VA 22182-2224
703-506-0500

Ed Spire
The Workstation Group
6300 River Road
Rosemont, IL 60018

Joe Vertucci
The Alive Centers of America, Inc.
3250 W. Market St.
Suite 202
Fairlawn, OH 44333

Bebo White
SLAC MS97
P.O. Box 4349
Stanford, CA 94309
bebo@slacvm.slac.stanford.edu

Tony Stephenson
U. S. Dept of Agriculture
Econ Mgmt. Staff/Personnel Division
1301 New YorkAve NW
Washington, DC 20005
202-219-0573
MAINT3@ERS.BITNET

Al Villarica
SYRACUSE UNIV.
104 Roney Lane
Syracuse, NY 13218
315-442-9198
RVILLARI@CAT.SYR.EDU

Michael Wright
Mobil Room 4A-508
3225 Gallows Rd.
Fairfax VA 22037
703-846-3930

Bernie Style
Systems Center
1800 Alexander Bell Dr.
Reston, VA 22091

Tony Walsh
Lotus Development Corp
One Rogers Street
Cambridge, MA 02142

James Youngdale
IBM M/D B025
10401 Fernwood Rd.
Bethesda, MD 20817
301-571-7520
JWYOUNG@BETASUM2

Dave Sutter
4912 Green Road
Raleigh, NC 27604

Chi-Ching Wang
10413 Quietwood Dr.
North Potomac, MD 20878

Kathryn Youngdale
IBM M/D D072
10401 Fernwood Road
Bethesda, MD 20817
301-571-2872
KYOUNG@BETASUM2

Philippe Taymas
Westin Ghost Energy Systems, Inc.
73 Rue De Stalle
B1180-Brussels, Belgium

Tom Wassel
IMS America Ltd.
Plymouth Meeting, PA 19462-0905
215-834-4447

Peter Zybrick
Future Systems
20 Dogwood Trail
Kinnelon, NJ 07405
201-492-2777

Anh Te
Towers Perrin
Centre Square East
1500 Market St.
Philadelphia, PA 19102-4790

Howard Weatherly
Computer Task Group
3347 Eastern Ave. NE
Grand Rapids, MI 49505-2576
616-3263-7634
cis:71327,1575

Melinda W. Varian
38 Gordon Way
Princeton, NJ 08540

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

Paul Verba
Relational Architects
33 Newark St.
Hoboken, NJ 07030

David Wescott
State of California
Health & Welfare Agency Data Center
1651 Alhambra Blvd. MS 710
Sacramento, CA 95816-7092
HWJ.DWESCOTT@TS3.TEALLE.CA.
GOV

ANNOUNCING

The REXX Symposium
for Developers and Users

San Diego, CA
May 18–20, 1993

- 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@slacvm.slac.stanford.edu
FAX: (415) 926-3329

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

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

Jim Weissman
(415) 688-6737
jhw@cup.portal.com
FAX: (415) 688-7269

Register and make travel arrangements by April 1 with:

Village Travel
(800) 245-3260
FAX: (415) 326-0245