SLAC-R-447
SLAC–447
CONF–9405161--
UC–405
(M)

PROCEEDINGS OF THE REXX SYMPOSIUM FOR DEVELOPERS AND USERS

May 1–3, 1994
Boston, Massachusetts

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

Program Committee

Cathie Dager of SLAC, Convener
Forrest Garnett of IBM
Pat Ryall

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

PROCEEDINGS OF THE REXX SYMPOSIUM
FOR DEVELOPERS AND USERS

TABLE OF CONTENTS

Summary

The fifth annual REXX Symposium for Developers and Users convened in Boston, Massachusetts, May 2–4. The fifty-seven attendees came from Australia, Austria, England, Norway, Canada and many US states.

This conference has become the premier event for exchanging REXX technical information, and people were impressed with how much REXX has spread since the last Symposium. This year we welcomed implementations for new platforms and continued growth in numbers of users and in importance of uses.

One of the most popular sessions was "Object-Oriented Extensions," given by Simon Nash of IBM. Also IBM gave the first public demonstration of their Object REXX for Windows. And the attendees continued the Symposium tradition of contributing software and making diskettes available for all.

The Symposium served as a springboard for the REXX Language Association (RexxLA) which will help promote the use of the language. RexxLA held its first public meeting in conjunction with this year's REXX Symposium.

Next year the Symposium will be held at the Stanford Linear Accelerator Center.

1994 Steering Committee:

Cathie Burke Dager
Forrest Garnett
Pat Ryall

IBM AIX REXX/6000 and IBM REXX for NetWare

Tom Brawn
IBM

IBM AIX REXX/6000
&
IBM REXX for NetWare

Thomas J. Brawn
Object REXX Development
IBM Endicott, NY
Internet id: tombrawn@vnet.ibm.com

IBM AIX REXX/6000

- Available 12/93
- Program number 5764-057
  - REXX/6000 Reference guide SC24-5708
    - *comments on reference guide can be sent to pubrcf@gdlvme.vnet.ibm.com*
- Supports AIX release 3.2 and up
- "AS-IS" release
  - In use internally at over 100 IBM locations for over 2 years

Additional details...

- Port of IBM's OS/2 REXX kernel to the AIX platform
  - SAA Level 2
    - *I/O functions (charin, charout, stream, etc)*
    - *System exits*
  - Default command environment
    - *address ksh 'ls -Fc'*
  - Access to AIX environment variables
    - *value('PATH',,'ENVIRONMENT')*

Additional Functions

- Directory
  - Works just like OS/2 cmd
- RexxRegisterFunction/RexxRegisterFunctionPkg
  - For registering functions from an application or functions from a function package
  - Similar API's for Subcom handlers and Exit handlers
  - SysAddFuncPkg, SysAddCmdPkg, etc. functions allow easy access to API functions
- SysCreatePipe
  - Creates an "unnamed pipe" for communication between two processes

More functions...

- SysFork()
  - Interface to fork command
- SysGetpid()
  - Returns process ID
- SysWait()
  - Waits on child process to end

IBM REXX for NetWare

- Available 3/94
- Program number 5764-075
  - IBM REXX for NetWare Reference Guide SH24-5286
    - ➡ *comments on reference guide can be sent to pubrcf@gdlvme.vnet.ibm.com*
- Supports NetWare release 3.11, 3.12, or 4.0
- "AS-IS" release

REXX for NetWare

Additional details

- Port of IBM's OS/2 REXX kernel to Novell's NetWare platform
  - SAA Level 2
    - *I/O functions (charin, charout, stream, etc)*
    - *System exits*
  - Default command environment
    - *Address netware 'load monitor'*

REXX API's

- RexxRegisterSubcomNLM
  - registers a subcommand handler from a NetWare loadable module
- RexxRegisterSubcomAdr
  - registers a subcommand handler from within an application routine
- Similar API's for registering external functions and exit handlers

NetWare specific utilities

- RxConsolePrint
  - Display a line of text on the system console
- RxDosCopy
  - Copies a file from the DOS partition to the NetWare partition
- RxDosPresent
  - Queries whether DOS is in memory
- RxGetNumberOfVolumes
  - Queries the number of volumes on the local server
- RxGetVolumeName
  - Given a volume number, get the volume name
- RxQueryNLM
  - Return NLM handle if the NLM is loaded

Industry excitement

- Tritus & IBM REXX/6000
  - Use REXX/6000 to extend the Tritus SPF editor
- Knozall & IBM REXX for NetWare
  - Use the REXX for NetWare scripting language module to extend and tailor the functionality of Knozall's NLMAuto Professional and NLMerlin products.

Trademarks

Object REXX—What's New?

Tom Brawn
IBM

Object REXX - What's new?

Tom Brawn
Object REXX Development
IBM Endicott, NY
Internet id: tombrawn@vnet.ibm.com

Background

- Work began in 1988
- Prototyped since 1989
- Complete rewrite of interpreter
- Language architecture "in progress" and subject to change
- Significant enhancements over the past 18 months
- Limited beta on OS/2 and Windows 4/94

Why Object REXX?

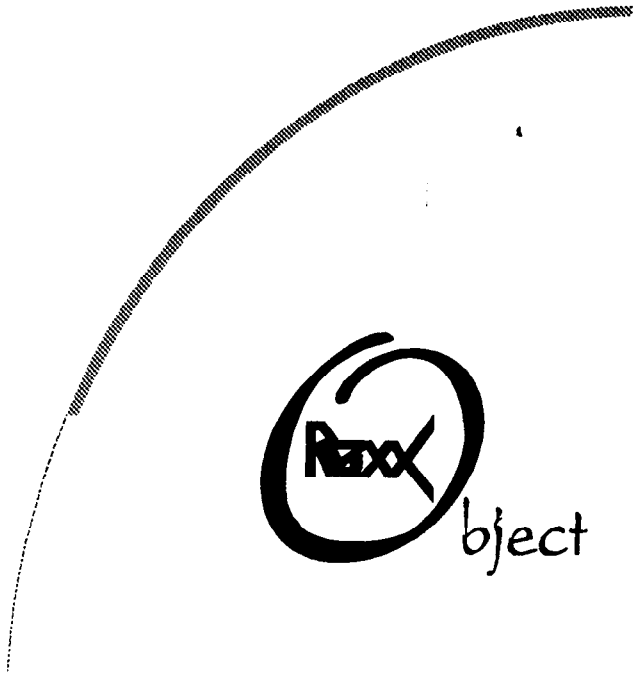
- Remove limitations of current REXX language
- Bring the power of OO programming to REXX
- Bring the usability of REXX to OO programming
- Extend REXX usage
  - windowing,

What's New in Object REXX?

- Objects
  - Everything in Object REXX is an object
- Methods
  - Everything that happens in Object REXX is a method
- Messages
  - Everything that happens in Object REXX is caused by a message

New REXX changes...

- Expressions in stems
- Parse enhancements
- Countstr/Changestr functions
- Extended Do syntax
- Date conversions



Expressions in stem references

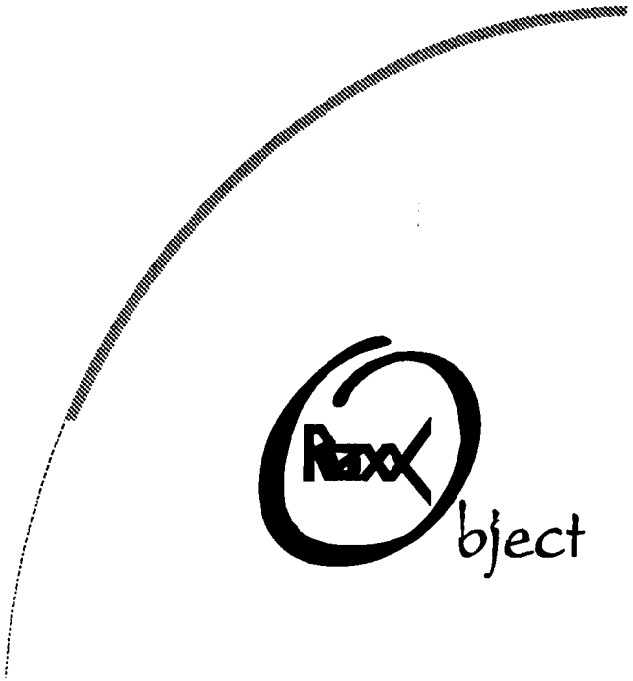
- Allow expressions on right side of stem .'s
- Use [ ] to contain stem expressions
- Example:

Before...

```
tom =index + 1
chris = index + 2
if employee.tom.chris ...
```

Is written...

```
if employee.[index+1].[index+2] ...
```



Parse enhancements

- parse caseless...
  - Parse template match without regard to case
  - Example:

```
parse caseless value 'Out To Lunch' with first ' to ' last
say first ==>Out
say last ==>Lunch
```

- parse lower...
  - Translate to lower case, then parse
  - Example:

```
parse lower value 'Out To Lunch' with first ' to ' last
say first==>out
say last==>lunch
```

countstr/changestr

- countstr(needle, haystack)
  - Returns count of needle in haystack
  - Example:

    countstr('is','This is a test of counstr') ==> 2

- changestr(needle,haystack,new)
  - Returns copy of haystack in which new replaces all occurences of needle
  - Example:

    changestr('1','101100','X') ==> 'X0XX00'

Extended DO

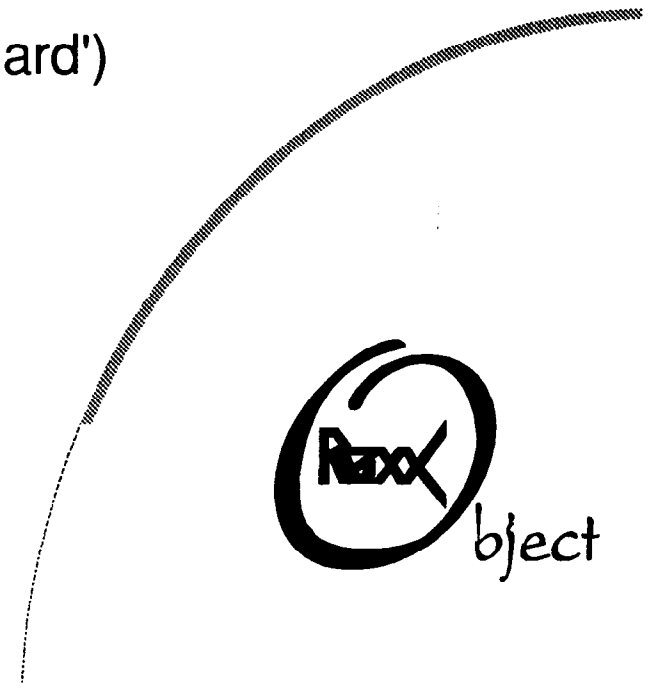
- Adds ability to iterate over stems
- Example:

```
fred.1='Tom';fred.2='Chris';fred.3='Alex';
do tail over fred.
  say fred.tail
end

==>Alex
==>Chris
==>Tom
```

Date Conversions

- Convert date to standard formats
- Example:

```
say date('usa',19931225,'standard')
==>12/25/93

say date('european',19931225,'standard')
==>25/12/93
```



Techniques for Performance Tuning REXX Interpreters—A Case Study of Regina

Anders Christensen
Sintef Runit

Techniques for Performance Tuning Rexx Interpreters—A Case Study of Regina

Anders Christensen
<anders.christensen@runit.sintef.no>

The Rexx Symposium for Developers and Users
Boston, May 1–4, 1994

Abstract

This article describes some of the techniques and methods used for optimizing the Regina interpreter, a REXX interpreter written in C, originally for Unix systems. The methods described first may be regarded as optimalization techniques in isolation, but they are also prerequisites for the last technique described here: the creation and maintenence of shortcut pointers from the parse tree to the variable structure.

1 Introduction

When tuning a program like a REXX interpreter for improved speed, a number of general techniques are used. Some of these are interesting in themselves, but not very specific to REXX interpreters. The scope of this text is to present some of the techniques that are closely related to the datastructures and operations of REXX interpreters.

2 Datatyping Variables

The fact that REXX is a typeless language is often described as one of its major advantages. Thus, it might be a great surprise to learn that one of the techniques boosting the performance the most, was introducing typed variables. Another technique was introducing typed expressions, which is described in the next section.

Internally, a Regina variable can hold either a string value, a numeric value, or both. When setting a variable, either a string or numeric value is set, depending on the context. Whenever the value of a variable is retrieved, it can be retrieved as either a string or a number. If a string value is retrieved for a variable currently holding only a numeric value, that value is converted so the variable holds both data type formats and then the string value is returned.

To understand the difference between these two formats, it might be instructive to look at their definitions in Regina.

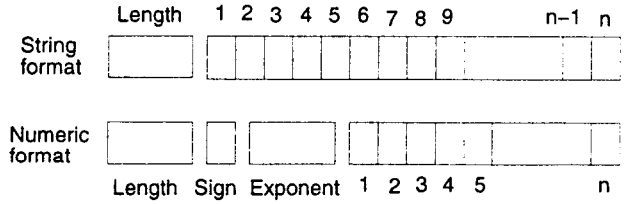


Figure 1: Storage formats for variables in Regina

For the string format, the values "2", " 2 ", and "2E0" are different, but for the numeric format, these are identical. The string is simply a sequence of characters, having a specific length. The numeric format is a sequence of decimal digits, to which there are connected three pieces of information: the length (number of digits), the sign, and the exponent (a native integer).

Consider the REXX statements:

	REXX statement	Numeric	String
1	`foo = 1+1`	2	N/A
2	`bar = foo || '.'`	2	'2'
3	`foo = (foo * 3) || ' '`	N/A	'3 '
4	`say foo*3`	3	'3 '

- After the first line, `foo` contains the numeric value 2, while its string value is not set. Note that it is not undefined, it can easily be converted from the numeric format, if necessary.
- In the second line, the string value of `foo` is retrieved, which means that the numeric value is converted. After the second line, both a numeric and a string value are stored for the `foo` variable.
- In the third line the numeric value of `foo` is retrieved and used in an expression which results in a string value. At the end of that statement, the `foo` variable is set to a new string value, and the numeric value becomes unset.
- In the fourth line the numeric value of `foo` is retrieved. However, at that point the `foo` variable have only a string value, so when retriving the value, the current string value is expanded to a numeric value. After the fourth line both a string and a numeric value are set.

Why maintain this double accounting? It turns out that variables set to a string value are very rarely used in numeric expressions . And vice versa, when a variable is set to a numeric value, it is seldom used in a string context; except for output statements, which tend to be slow anyway.

Based on these two observations, it makes sense to have two parallel, highly optimized sets of functionality for operating variables: one for numeric values and one for string values. Since the conversion between them are rather rare, the more time-consuming code for conversion between the two formats does not significantly increase the total execution time.

As a future extension, the scheme may be expanded to handle boolean variables too. However, it may turn out that the increased complexity this requires (six conversion types as opposed to only two above) may not justify the increase in speed. The use of boolean variables are much less widespread than string and numeric variables; and besides, boolean variables can be emulated through numeric variables.

In addition, the native floating point numbers could be used. It beats REXX numbers in speed, but it is difficult to avoid loosing accuracy wrt the definition of REXX arithmetics.

3 Construction of a Parse Tree

In order to explain what comes next, we need to know the format in which Regina stores a parsed REXX program. As an example, consider the following REXX code:

```
if ('xxx'/=bar) & (bar*foo>1000) then
   exit
```

Regina converts this sequence of tokens to a parse tree, the expression in the if-clause is shown in figure 2. The conversion between a sequence of tokens and a parse tree is described in most text books on compiler construction. As an aside note: parse trees are often considered to be incompatible with the customary way REXX programs are stored internally—a list of tokens. However, a static tree can easily be converted to a list of tokens. The difference lays in generating

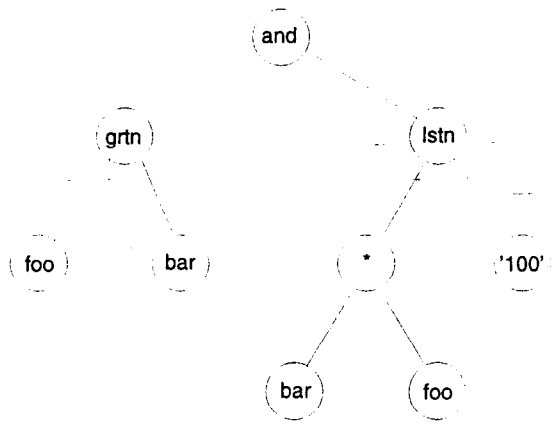


Figure 2: A parse tree built by Regina

a parse tree, which requires a more thorough analysis than a simple conversion of the source code to a list of tokens.

The most obvious approach for executing the code represented by the parse tree would be to traverse the parse tree, and for each binary operator ("=/", "&", "*", etc) first traverse the left subtree, then the right subtree, and in the end apply the operations to the two strings obtained from the traversals.

It is possible to add some optimalizations here:

`bar =/ 'foo'`
: We know that this must always be a non-numeric comparison, this there is no need to try anything but a normal string comparison straight away.

`2*bar`
: In this expression, we are only concerned with the numeric value of `bar`, so we retrieve its value in numeric "mode", as described in section 2.

`(a>b)&(c<d)`
: Here, each of the two pairs of parentheses can result in either "1" or "0". Thus, we use the native integer format of the computer to signify the values, rather than using the Regina string or numeric format.

4 Datatyping Expressions

Using these techniques, the dataformat of the data transmitted from a subtree to its parent node depends on the context. For instance, consider the parse tree shown in figure 2. After adding the datatypes, the new parse tree is shown in figure 3.

5 Hash Tables to Store Variables

Regina uses hash tables to store the variables defined at any given point during the execution of a REXX script. This technique can make the retrieval of a variable a constant-time operation, if given a well balanced hash table. However, once the hash table becomes full, the efficiency drops.

One of the key points with hash tables is to choose the correct size. If the size is too small, the handling of overflow adds a large overhead. If the table is too big, the extra work of initialization and deallocation adds unnecessary overhead. One solution is to have only one huge hash table for the whole interpreter, in which case the work of initialization and deallocation of the hashtable is done only once. However, this requires some extra overhead for insertion and deletion of variables.

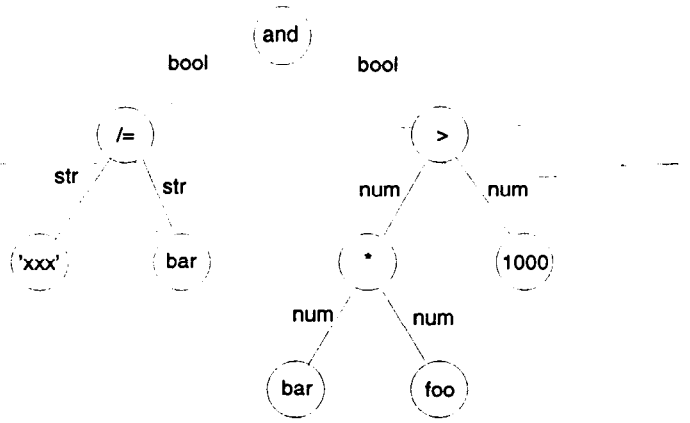


Figure 3: Parse tree with datatypes of transmitted results

Among other things, it makes the operations of deleting all tails of a particular stem a bit more complicated.

Another solution is to use dynamic hashing, where a small hash table is used initially, and the table is expanded when it is filled. The advantage of this technique is illustrated by the fact that the interpreter has no way of predicting the number of tails used by a routine at the entry of the routine. (Except that it may cache the number of tails used at earlier calls.)

Figure 4 shows how Regina stores its variables. There is one hash table for each subroutine having a PROCEDURE clause, and within each such hash table, there is another hash table for each stem in use.

6 Shortcut Pointers from Parse Tree

A well-known technique for optimizing computer code is to cache any value for which you may have need later. Regina makes use of this several places. For instance, whenever Regina executes a CALL clause or a function call for the first time, it must determine which routine to call. If the destination routine is an internal or built-in function, it is cached by setting to pointer in the parse tree to point to it.

7 Shortcut Pointers to the Variable Structure

Whenever a REXX clause refers to a variable name, the value of that variable must be retrieved from the variable structure. This involves some navigating, which can be time-consuming. However, it often turns out that multiple invocations of the same variable reference in a clause navigate through the variable structure only to end up at the same variable box. Thus, it may be advantageous to cache the result of the most recent navigation for each variable reference of the program. This means storing a pointer in the parse tree, pointing into the hash table of the variable structure.

Consider the following trivial code:

```
foo = 1
do 1000
   foo = foo + 1
   end
```

If we restrict the analysis to the contents of the loop, the variable `foo` is set 1000 times and its value is retrived 1000 times. I.e., navigating the variable structure 2000 times.

Then we add functionality for caching the result of each navigation. Neither retrieving nor changing the value of a variable are operations which change the identity of the box in the variable

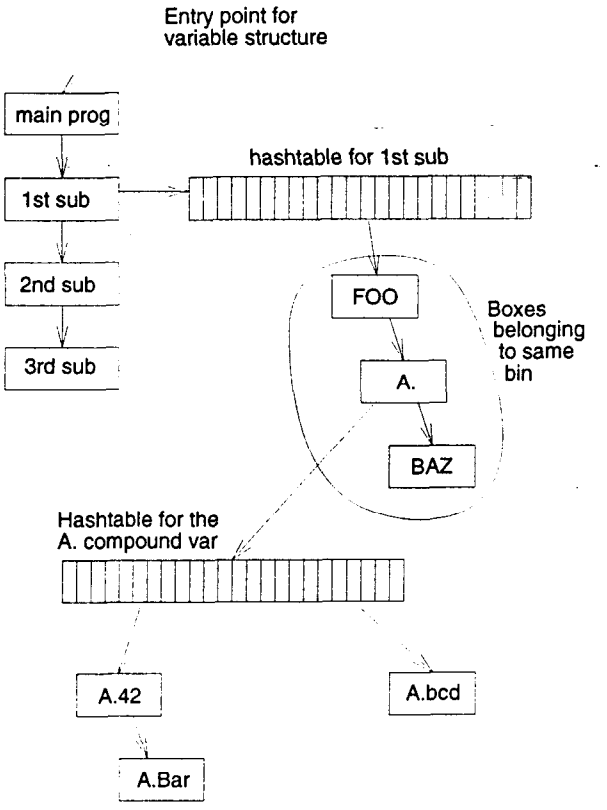


Figure 4: The structure of variables in Regina

structure where the variable is stored. Thus, if we can cache a pointer to the variable, the number of walks through the hash table structure drops from 2000 to 2.

On the other hand, the cost of this is caching the pointer after each navigating walk, unless it was already cached. And the cost of verifying that the shortcut pointer is still valid. In particular the latter of these introduces a number of subtle points. Consider the following code:

```
/* first example */
call foo
exit

foo: do i=1 to 2
   say i
   if i=1 then do
      procedure
      i = 1
      end
   end
```

In this example (which is only allowed for TRL1—not TRL2) the variable `i` in the `SAY` clause refers to different variables during the first and the second iterations of the loop. This is due to the execution of the `PROCEDURE` clause during the loop, which changes the scope of the `i` variable. Thus, the shortcut pointers cached during the first iteration must be tested during the second iteration, and the fact that they are invalid must be detected.

This is achieved using a generation number, which is identical to the number of currently nested functions having executed the `PROCEDURE` clause. Whenever a new `PROCEDURE` clause is executed, the generation number is incremented, and whenever a `RETURN` clause is executed for a

routine which have—during its course—executed a PROCEDURE clause, the generation number is decremented.

To verify the validity of a shortcut pointer, the current generation number is recorded in the box pointed to by the pointer. Whenever a recorded pointer is to be validated, it is considered invalid if the current generation number is greater than the number recorded in the box pointed to by the shortcut pointer (i.e. a PROCEDURE clause has been executed since this pointer was made, invalidating the pointer). In this case, the recorded shortcut pointer is attempted deallocated, and the variable is located using the standard procedure—the new location is of course cached if the current generation number is greater than the recorded number.

The next example shows a function.

```
/* second example */
say bar(3)
exit

bar: procedure
   parse arg i
   if i=1 then
      return 1
   else
      return bar(i-1)*i
```

Here, the last clause in the routine is executed twice, as a result of the recursion. However, due to the rules for evaluation of REXX expressions, the retrieval of the i variable at the end of the last clause is executed twice: first at end of the second invocation of bar, and then at the end of the first invocation of bar. (Note: i is referred to after the recursion itself.)

According to the rules outlined above, the shortcut pointer is cached at the end of the second call to bar (the first recursive call). Thus at the end of the first call to bar, this cached value is picked up, but the generation number does not match (the recorded generation number is greater than the current generation number), so the shortcut pointer is discarded and the variable is located using the standard procedure (i.e. since the pointer was made, the routines in which it was made has been terminated).

There is another, less subtle point here, too. All variables local to the second (recursive) call to bar are discarded when that routine returns. Thus the shortcut pointers appear to point to undefined memory! This is easily fixed by maintaining a counter with each variable box. Whenever a shortcut pointer is set to point to that box, the counter is incremented; and whenever a shortcut pointer is removed from pointing to a variable box, the counter is decremented. As soon as this counter mechanism is in place, a variable box can be marked for deletion, and retained until all shortcut pointers to it have been killed.

```
/* third example */
do i=1 to 2
   call bar
   end
exit

bar: procedure expose i
   if i=1 then
      foo = ''
   say foo
   return
```

Here, the second invocation of bar finds the cached pointer in the SAY clause, but the pointer is invalid, even though the generation number is correct. To handle this case, variables discarded during the execution of the RETURN clause are not immediately discarded if there are any shortcut

pointers pointing to it (as recorded by the shortcut counter field). Instead, it is suspended until all shortcut pointers point elsewhere, at which time the box is deallocated. In the meantime a flag is set for the variable box, so that the interpreter can discover that the box is invalid if it tries to dereference the shortcut pointer.

```
/* fourth example */
do i=1 to 4
   if i=2 then do
      drop i
      i = 2
      end
   end
```

The code of the fourth example, as shown above, illustrates why the delete flag is necessary. The variable box created at the start of the loop is dropped during the loop, so a mechanism is necessary to detect that the box is invalid at the start of the next iteration.

8 Algorithms

The two algorithms shown are the the central for the correct operation of the shortcut pointers in Regina. The first algorithm is shown in figure 5, and describes how to access (retrieve or update) the value of a variable. To be effective, it requires that the code has been executed at least once before, so that shortcut pointers have been created.

```
foo is a variable reference to access
 if exists a shortcut pointer for var  then
   if points to a variable not deleted  then
     if the generation number is correct  then
       retrieve/set the value
       return
     else
       decrement counter
       remove shortcut pointer
   else
     decrement counter
     remove shortcut pointer
 if counter=0  then
   delete/deallocate variable box
access variable "the hard way"
cache the found box in the shortcut pointer in the parse tree
increment counter
return
```

Figure 5: Retrieving/setting value of variable reference in the parse tree

The second algorithm is used to delete variables during the execution of the `RETURN` clause from a routine which had its own "private" `PROCEDURE` clause. It will always detach the variable boxes, but it will only deallocate the space if there are no shortcut pointers pointing to the box (as recorded by the counter in the box).

```
for each local variable
  disconnect it from variable system
   if counter is greater than 0  then
     mark variable box as deleted
   else
     deallocate variable box
```

Figure 6: Deleting local variables at return from routine

9 Why So Complicated?

Most computer languages keep track of their variables in much easier ways, so why introduce this complexity for REXX? Because of the enormous degree of freedom in REXX. REXX does not have compile-time routines, it has "only" run-time routine entry and exit points! Therefore, it is virtually impossible to bind a given clause to a particular "routine" at parse time. The possibilities of "`SIGNAL ON`" and "`INTERPRET`" ensure that control can pass from virtually any clause to virtually any label in a REXX program.

Thus, the techniques used for most compilers and some interpreters, which allow them to bind the variable references in the source code to specific locations at compile- or parse-time do not work for REXX, and more elaborate systems, like the one described above, are called for.

REXX/imc
A REXX Interpreter for UNIX

Ian Collier
Oxford University

Work in progress

REXX/imc

A Rexx interpreter for Unix

Ian Collier

available from `rexx.uwaterloo.ca`

in `/pub/freerexx/imc`

Abstract

Since 1989 I have been working on a Rexx interpreter for Unix in my spare time (what little I get). It was first released to the public in August 1992 and has had many improvements since then. In my presentation I will demonstrate the most recent enhancements and some of the language extensions that I have added to the interpreter, a few of which are connected with the work of the X3.J18 standardisation committee. I hope to show some of the ways in which REXX/imc can interface with its environment; this will include the use of Unix-specific built-in functions, the writing of external function libraries, and the application interface with programs such as THE (an editor based on KEDIT written by Mark Hessling). If time allows, I will take a brief look at the internals of the interpreter, showing the basic blocks of which it is built, and giving a short explanation of how it performs a task such as evaluating a Rexx expression.

Unfortunately, since my 'real' job is to write a D. Phil. thesis, I have not been able to enhance REXX/imc as much as I would have liked for this presentation. However, work is still in progress to turn REXX/imc into an efficient and fully integrated programming language on Unix.

History

- May 1989: Work begins!
- Jan 1991: Interpreter has most language constructs except the stack, but no I/O functions.
- May 1992: REXX/imc is not ready in time for the Symposium.
- Aug 1992: REXX/imc release 1.2 released.
- Sep 1992: Release 1.3
- May 1993: Release 1.4 announced at the Symposium.
- Jun 1993: Release 1.5, the first level 4.00 release of REXX/imc.
- Sep 1993: Release 1.5a, with some bug fixes.
- May 1994: Release 1.6 is presented at the Boston Symposium.
- 1994– : ?

Because REXX/imc is a spare-time project, work on it has been characterised by bursts of activity and long periods of slow development. Even though the interpreter was functional in 1991, it was not released until August 1992. In fact it is interesting to note that REXX/imc was already capable of running a program to calculate π to many decimal places by October 1989, although it had no functions.

The period between the 1992 Symposium and the initial release of REXX/imc was spent in implementing the file I/O functions and in documenting the source—even the few comments that are dotted around now were almost entirely absent before this period!

Many of the changes between versions of REXX/imc have been bug fixes—thanks to Anders Christensen who spent time running his trip tests on REXX/imc, and to everyone who reported a bug.

The main changes in release 1.5 were the addition of language level 4.00 features (`SIGNAL ON` with the `NAME` keyword, `CALL ON`, `CONDITION()`, `STREAM()` and so on), the command line flags, and the `OPTIONS` options.

The main changes in release 1.6 are the addition of an API and the improvement of function handling.

Things planned for the future include, but are not limited to, the following (not in any particular order): implementing speedups (in at least three areas: improving the variable table, improving the arithmetic and implementing a pre-parsing process), improving tracing, adding a Unix system call library, adding `OPTIONS` to control the language extensions and to move towards the language standard, adding extensions as proposed by the Rexx Extensions committee, adding a 'stems' library, completing the API, adding an API which can be called by other processes even after Rexx has started, and anything else which people suggest...

Files

librexx.so.1.6	204800
rexx	5712
rxmathfn.exec	6743
rxmathfn.rxfn	57344
rxmathfn.rxlib	57
rxque	8016
rxstack	6600
const.h	16140
functions.h	16423
globals.h	6165
rexxsaa.h	5678
calc.c	49157
globals.c	8683
interface.c	37875
main.c	4896
rexx.c	97258
rxfn.c	77118
rxmathfn.c	8061
rxque.c	8610
rxstack.c	6051
shell.c	8228
util.c	80214
rexx.info	33568
rexx.ref	155257
rexx.summary	12627
rexx.tech	33320

The file **librexx.so.1.6** is the main library file which contains all the routines necessary for an application to use the SAA API of REXX/imc. On the SunOS system, this is a dynamically loaded shared library, which means that an application which uses the library does not need to include a copy of the library within its object code, thus saving disk space. This can be seen from the fact that the program **rexx**, which is the interpreter itself, is only a 6K file! This program is merely an interface between the command line and the API library, and is compiled from the source file **main.c**.

The programs **rxque** and **rxstack** are for the Rexx stack, which will be discussed later.

The file **rxmathfn.rxlib** is a function dictionary for the REXX/imc mathematical functions, which are implemented in Rexx as **rxmathfn.exec** and in C as the object file **rxmathfn.rxfn**.

As shown opposite, REXX/imc comes with about 430K of source.

The four major documentation files shown opposite are **rexx.info**, which is my attempt at a tutorial for Rexx, **rexx.ref**, which is a complete reference on the language features of REXX/imc, **rexx.summary**, which is a 'reference card' on REXX/imc, and **rexx.tech**, which gives details to the application programmer or any programmer who is interested in the internals of REXX/imc. There are also several minor documentation files, not shown here, which give details about the current release, the change history, the installation instructions, etc.

Invocation

rexx [options] [program] [arguments]

where options are:

-<option> – any option from '**OPTIONS**';

-v – print version;

-s <string> – execute the string as a program;

-t <trace> – turn tracing on;

-i – enter interactive trace mode;

-x – run a Unix-executable Rexx program.

- The **OPTIONS** instruction's most useful option for using on the command line is the **tracefile=f** option, which redirects tracing output to a file.
- The **-t** option can be followed by any Rexx trace setting, which allows you to trace a program without altering it.
- The **-v** option can be used alone (in which case the interpreter does nothing except print its version) or with other options (in which case it prints its version and then runs a program).
- The **-x** option is usually used for programs which invoke themselves on Unix by having a '**#!**' or a shell instruction on the first line. REXX/imc will treat the first line of the program as a comment, and will not append anything to the program name.
- If no program name is given, or if the program name is '**-**', then the program will be read from the standard input.

Added Features

- `stem.(expression)`
- `stem.'string'`
- `SELECT expression`

  `WHEN value THEN instruction`

  `...`

  `END [SELECT]`
- `PROCEDURE HIDE`
- `PARSE VALUE x,y,z WITH p1,p2,p3`
- Any non-zero number is true
- `OPTIONS 'SETRC'` for setting `RC` after I/O operations
- `*,*` trace prefix for continued lines
- Extra tracing for `SIGNAL ON x` when `x` is an undefined label
- Features from CMS
  - `PARSE NUMERIC`
  - `JUSTIFY()`
  - `LINESIZE()`

Of these enhancements, one, namely the `*,*` trace prefix, is as a result of a decision of the X3.J18 standardisation committee, and one other, namely the compound variable with an expression as part of its tail, has been provisionally accepted by the extensions committee. More substantial enhancements based on meetings of these committees (such as date/time conversion functions) were planned but have been delayed.

The `PROCEDURE HIDE` instruction really means 'procedure expose everything-except-the-following', and its use is not strongly reccommended at present.

The `OPTIONS 'SETRC'` instruction makes all I/O (including `SAY` and `PARSE PULL` set the variable `RC` to indicate the success or otherwise of the operation, in order to allow this to be checked without calling `STREAM`. It also causes a `SIGNAL ON ERROR` if that is appropriate. This option was added in order to preserve backward compatibility with a previous version of REXX/imc which had neither `STREAM` nor `SIGNAL ON NOTREADY`.

The 'extra tracing' extension prints out a traceback including the `SIGNAL ON` instruction and the cause of the error whenever the target label for the trap is not found. For example, the program:

```
signal on novalue
call test
exit
test: say xyz
```

produces this traceback:

```
      +++ No-value error on XYZ
    1 +++   signal on novalue
    4 +++  say xyz
    2 +++ call test
Error 16 running test.exec, line 1: Label
not found
```


Features for Unix

- The TRL I/O functions
- Pre-defined streams: `stdin stdout stderr`
- The `STREAM` commands: `close fdopen fileno flush ftell open pclose popen`
- Functions: `CHDIR GETCWD SYSTEM USERID`
- Access to the Unix environment via the `VALUE` built-in function
- Access to Unix error messages via the `ERRORTEXT` built-in function
- Subcommand environments `UNIX` and `COMMAND`
- The stack daemon
- The function interface

- REXX/imc offers a variety of file access functions via the function call `STREAM(stream,'C',command)`. The `open` command allows any file to be attached to a stream in either read or read/write mode. The `popen` command starts a Unix command and attaches it to the named stream for reading or writing. The `fdopen` command allows Rexx to access any Unix file number as a stream. The file number of any Rexx stream is given by the `fileno` command. The `ftell` command gives the file pointer which was set by the last access on the named stream.
- The `SYSTEM` function runs a shell command and returns its output as a string.
- Environment variables may be examined and/or set using the `VALUE` function with a third argument of `'ENVIRONMENT'`. Note, however, that changes made to the environment will be lost when the Rexx interpreter finishes.
- The function call `ERRORTEXT(n+100)` gives the nth Unix error message, such as 'No such file or directory', which is message number 2.
- The subcommand environment `UNIX` passes each command to a Bourne shell. The `COMMAND` environment passes each command to a small built-in shell which tokenises and executes the command directly, which is usually much faster than invoking a shell for each command.

The REXX/imc Stack

- `rxque` is the stack daemon
  - it runs as a separate process
  - it is created and destroyed automatically by the interpreter
  - it may be run as a server for a whole session
- `rxstack` is a stack client
  - `rxstack [-fifo|-lifo]` copies standard input to the stack
  - `rxstack -string x` stacks one entry
  - `rxstack -print` copies stack contents to standard output
  - `rxstack -pop` copies one entry to standard output
  - `rxstack -num` prints the number of stacked entries
- REXX/imc is also a stack client
  - `queue x` stacks an entry in FIFO order
  - `push x` stacks an entry in LIFO order
  - `queued()` tells the number of stacked lines
  - On SunOS, REXX/imc can transfer stack contents to the keyboard buffer.

The program `rxque` forks off a stack daemon and prints out its process number and socket name in the form of two environment variables. The format of the output is as either a Bourne shell command or (with the flag `-csh`) a c-shell command. `rxque` may be given the name of a socket to create, in which case the output is just the process number.

The stack daemon is usually started by REXX/imc and killed with signal 15 when the Rexx program finishes. REXX/imc checks for the presence of a stack daemon by looking for environment variable RXSTACK. If a stack exists, then it uses that instead of creating one. Queued entries may then persist between programs:

```
% eval `rxque -csh`
% ls -al | rxstack
% rexx -s "say queued()"
45
% rexx -s "pull .; parse pull a; say a"
drwx------  5 imc       1024 May  2 16:00 .
% kill $RXSTACKPROC
```

On some systems, REXX/imc can be compiled with the preprocessor symbol `STUFF_STACK` defined. REXX/imc can then pretend to cause persistent changes to the environment:

```
% rexx -s "queue 'cd /tmp'"
cd /tmp
% % pwd
/tmp
```


Application Programming Interface

The following SAA API functions are implemented:

- RexxStart
- RexxVariablePool (except requests RXSHV_EXIT and RXSHV_PRIV)
- RexxRegisterSubcomExe
- RexxDeregisterSubcom
- RexxQuerySubcom
- RexxRegsiterExitExe

  with exits: RXCMDHST RXSIODTR RXSIOSAY RXSIOTRC RXSIOTRD RXINIEXT RXTEREXT
- RexxDeregisterExit
- RexxQueryExit
- RexxRegisterFunctionExe
- RexxDeregisterFunction
- RexxQueryFunction

More will be added later.

Release 1.6 of REXX/imc is the first to have an API. The functions have been modelled on those of OS/2. It should be possible to compile a Rexx-aware application—such as Mark Hessling's editor 'THE'—with REXX/imc without altering it (as long as it uses only the functions which are currently supported).

In order to use the API, an application includes the C header file `rexxsaa.h` supplied with REXX/imc, which will declare the functions opposite and the associated constants and datatypes. When the application is compiled, it is linked with the library file which is created when REXX/imc is compiled. This file will be either `librexx.a`, in which case the code from REXX/imc will be included in the application's object file (*static* linkage), or `librexx.so.1.6`, in which case only a reference to the library file will be included in the application's object file (*dynamic* linkage).

If linkage is dynamic, it will be possible to upgrade to a later release of REXX/imc without recompiling the application, just by copying the new library into the same directory as the old one.

External Functions

External functions or libraries for REXX/imc can be written

- in Rexx
- using the SAA API
- using REXX/imc hooks
- as a Unix program

Writing an external function in Rexx or with the SAA API is the same as for any other interpreter.

A function may be compiled and linked as a dynamically loaded object called `*.rxfn` with the `*` replaced by the function's name (by which it will be called by a Rexx program). When REXX/imc searches for external functions, it searches for such a file first. If the file is found, it is linked in and called as if it were built-in. The function must retrieve its arguments from the REXX/imc calculator stack and place the result (if any) there.

A `*.rxfn` file may contain several functions, all of which will be registered when the file is first loaded.

A function library using the SAA API may be compiled as a `*.rxfn` file in order to make a library which is portable but which can be called by an already-running program. To do this, the library is augmented by an initialisation function which takes no parameters and returns no result, but which uses the SAA API to register all the other functions in the library. Before calling any of the functions, the Rexx programmer must call the initialisation function.

If a function cannot be found, then a Unix program having the same name as the function is searched for. The program can be in any language supported by Unix, such as C, perl or shell script. It will be 'exec'ed with the arguments in `argv[]` and the function name in `argv[0]`, and it should print out the result (if any) on its standard output followed by a newline character.

Many functions can be aliased to one function library by supplying a text file called `*.rxlib` (where `*` is the basename of the function library) which lists the names of all the functions in the library. The library can be a `*.rxfn` file, a Rexx file or a Unix program. If it is Rexx, then it can find out which function is being called using `parse source`.

Interpreting a program

1. Read command line parameters (`main()`)
2. Load program from disk (`load()`)
3. Tokenise program (`tokenise()`)
4. Enter main loop (`interpreter()`)
   (a) Fetch the next token.
   (b) If `NOP` then do nothing
   (c) If `SAY` then print an expression
   (d) If `RETURN` then return an expression
   (e) If `IF` then read and test an expression
   ...
   (f) If program has ended then return, else go to (a).
5. Clean up and finish.

Tokenising a program means (in the case of REXX/imc):

- rejecting invalid characters and unmatched quotes
- removing comments, null clauses and excess blanks
- Concatenating lines which are continued with a ','
- translating un-quoted text to upper case
- recognising keywords (like `NOP`, `SAY`, `IF` and so on)
- organising the program as a list of clauses (each end-of-line, ';', or `THEN` starts a new clause. In addition, labels, `THEN`, `ELSE` and `OTHERWISE` are clauses by themselves)
- making a label table

Keywords are recognised based on what has appeared since the start of the current clause. For example, `THEN` is only allowed when the current clause started with `IF`. Keywords are stored as negative character codes (defined in `const.h`). This makes them easy to recognise: during the main loop, instead of asking, "Are the next three characters 'say'?" we can ask, "Is the next character equal to the constant `SAY` (which is -128)?" It also makes it clearer for the expression evaluator when to stop; the code `WHILE` (-88) is obviously not part of an expression, whereas the word `while` could be a variable name.

The tokenised list of clauses is stored in an array `prog[]`, which also gives other information such as the line number and address of the clause within the source.

The main loop is relatively trivial; it is executing the individual instructions such as `DO` and evaluating the expressions which is the difficult part...

Internal data structures

- the source (`source`)
- the tokenised program (`prog`)
- the label table (`labelptr`)
- the calculator stack (`cstackptr`)
- the program stack (`pstackptr`)
- the signal stack (`sgstack`)
- the variable table (`vartab`) and pointer list (`varstk`)
- the work space (`workptr`)

- The source and tokenised program are each kept in a linear stretch of memory, pointers to which are held in the arrays `source` and `prog` respectively. The label table is stored in a linear stretch of memory which is organised as a kind of linked list.
- The calculator stack is a space to store a list of intermediate values during calculations.
- The program stack records information about the control structures that are currently open (such as `DO` groups and function calls). It stores the variable name, step and limit and/or the `FOR` counter of a `DO` instruction, and it stores all the saved state which must be restored on return from a function call.
- The signal stack holds information about which conditions are currently trapped or delayed, and it also holds the data for the `CONDITION` function. It has one entry for each `INTERPRET` or function call currently active.
- The variable table is a linear stretch of memory which is divided into sections by `varstk`. Each section contains the variables for an active `PROCEDURE` or external function call (apart from the workspace, this is the only one of the above structures which persists across external function calls). Within each section the variables are stored in a tree structure. Exposed variables contain a pointer to another section where the 'real' copy of the variable is to be found.
- The work space is a temporary area for all sorts of calculations. It is cleared after interpreting each instruction.

Example: DO

1. Store the current clause number on the stack.
2. Fetch next token. If clause has ended then finish.
3. Flag the stack entry as 'repetitive'.
4. If the token is `FOREVER`, skip past it.
5. Otherwise, try and fetch a symbol and '='. If found then:
   (a) Store the symbol name on the stack.
   (b) Fetch an expression and assign it to the symbol.
   (c) Search for `TO`, `BY` and `FOR` expressions and store them on the stack.
   (d) If the limit is already passed then `LEAVE`.
6. If that failed, try to evaluate an expression and store it on the stack.
7. Store the pointer to any `WHILE` or `UNTIL` on the stack.
8. If `WHILE` is found and the following expression is false then `LEAVE`.

`DO` and `END` have been chosen to illustrate how the program stack works.

Most of the work of `DO` is to find out what sort of `DO` clause this is and to set up an entry in the program stack which describes the `DO` clause. The information needed is:

- where to come back to
- whether there is a symbol and if so, what are its name, and its step and limit values
- whether there is a counter or `FOR` value, and if so, how many iterations are left
- where the `WHILE` or `UNTIL` can be found, if any

`DO` also has to check to make sure the loop is to be executed at least once.

Example: END

1. Fetch the top stack entry. If none exists, complain.
2. If the entry is not from `DO` or `SELECT`, complain.
3. If the entry is not flagged 'repetitive' then
   (a) Delete the top stack entry
   (b) finish.
4. Fetch the pointer to any `WHILE` or `UNTIL`. If `UNTIL` is found and the expression following it is true, go to 3(a).
5. If a symbol name is stored, add the step to it and compare with the limit. If the limit is passed, go to 3(a).
6. Decrement any `FOR` counter. If it is zero, go to 3(a).
7. Fetch the pointer to any `WHILE` or `UNTIL`. If `WHILE` is found and the expression following it is false, go to 3(a).
8. Fetch the stored clause number and jump to the following clause.

Even though the `END` instruction contains no information (although it might contain a symbol name, details of which have been skipped here), it can be interpreted because the information is all on the program stack. Interpreting the stacked data is relatively straightforward.

Example: expressions

There is a stack of values and a stack of operations.

1. Stack an 'end marker' operation with priority 0.
2. Search for a value:
   - If the next token is a unary operation, stack it and repeat 2.
   - If it is '(' then evaluate the expression inside, check for ')' and go to 3.
   - If it is a quote, collect a string.
   - Collect a symbol name.
   - If the token after the string or symbol is '(' then call a function, otherwise stack its value.
3. Search for the 'current' operator:
   - If the next token is a keyword, ')', ',' or the end of the clause then the operator is an end marker.
   - Otherwise, if it is not a binary operator then the operator is an implicit concatenation.
4. Perform operations:
   - If the top stacked operator and the current operator are both end markers, then finish.
   - If not, and the top stacked operator has a priority no less than that of the current operator, perform the stacked operator and go to 4.
   - Otherwise stack the current operator and go to 2.

The function which performs the above algorithm is called **scanning**.

This is a variant of a well-known algorithm to turn an expression in infix notation into one in reverse polish notation (sometimes described by analogy with a railway track with a siding, the siding being the operation stack). REXX/imc evaluates the reverse polish expression as it is created. The calculator stack is the stack which reverse polish notation requires.

The unary operations each operate on the top value on the calculator stack, replacing it with the result. The binary operations each operate on the top two values, replacing them with the result. It is clear that at step 4 of the above algorithm it is always true that the number of values on the calculator stack is one more than the number of stacked binary operations. Since each stacked binary operation reduces the size of the calculator stack by one item, this means that when the stacked operations have all been performed there is precisely one element left on the calculator stack. This is the result.

Arguments to functions and expressions within parentheses are evaluated by calling **scanning** recursively.

Interesting Corners of REXX

Mike Cowlishaw
IBM

Interesting Corners of REXX

REXX Symposium

Mike Cowlishaw

IBM UK Laboratories
Hursley

May 1994

Outline

- Instructions
- Built-in Functions
- Miscellaneous
- Questions?

Multi-way CALL

```
var='FRED'
call jumper var, firstarg
    ¦
    ¦
jumper: procedure
signal value arg(1)
    ¦
    ¦
fred:     /* do whatever */
    ¦
return    /* to the CALL */
```

DO FOREVER—can be clearer

```
do forever
     ¦
     ¦
   if something then leave
     ¦
   end /* forever loop */
```

DROP—extra state for a variable

```
drop var
do i=1 to howmany
     ¦
   if whatever then var=somevalue
     ¦
   end i

if symbol('var')\='LIT' then say 'Found!'
```

NUMERIC FORM and FUZZ

With *NUMERIC FORM ENGINEERING*:

```
var=1234
say var*1e10     var*1e11     var*1e12
```

→

```
     12.34E+12    123.4E+12    1.234E+15
```

... and don't forget *NUMERIC FUZZ* for fuzzy comparisons.

PARSE

Most implementations have variable column patterns:

```
namecol=pos('Name', header)
do i=1 to entries
  parse var entry.i =(namecol) name.i
  end i
```

Use "." placeholder to strip blanks:

```
line='modemspeed     = 9600 '
   |
parse var line key . '=' value .
```

More PARSE

Use relative patterns to include strings in results:

```
parse var line pre 'START' 'SLIP' +0 post
if pre='' & post='SLIP' then do
   /* found 'start ... slip' */
   end
```

or...

```
parse var line pre 'WAIT' +0 key num post
if pre='' & key='WAIT' & post='' then do
   /* found 'wait [num]' */
   end
```

Parsing field-oriented data

```
/* Set up template matching structure */
/* (perhaps read from a file).        */
template='socsecnum    + 9',
         'name         +40 -40',
         '  last       +20',
         '  first      +20',
         'balance      + 4'
   ¦
record=charin(myfile, 80)
interpret 'parse var record' template
balance=c2d(balance, 4)
```

Using PARSE for POS/SUBSTR

```
/* Change all "old" to "new" in string */
/* If "old" is null,"new" is prefixed  */
Change: procedure
  parse arg string, old, new
  if old=='' then return new||string
  out=''
  do while pos(old, string)\=0
    parse var string prefix (old) string
    out=out||prefix||new
    end
  return out||string
```

PROCEDURE EXPOSE lists

Lists can be very useful with *PROCEDURE EXPOSE*:

```
errors='sigl rc CleanupFlag'
shared='masterlist. CurName CurCount'
     ¦
     ¦
subfunction:
  procedure expose (errors) (shared)
     ¦
  return
```

TRACE

Don't forget:

```
trace Labels
```

... lets you check the flow in a program

and...

```
trace Intermediates
```

... lets you check expression evaluation in detail.

Note: The TRACE instruction is completly ignored during interactive tracing—but the TRACE() built-in function is not.

Function names in quotes

System-dependent function names can be useful:

```
say  'e:\tools\testit.cmd'()
```

or...

```
say  'EXEC PROFILE'()
```

... and they work with CALL, too.

Built-in Functions

ABBREV allows default match to null string:

```
say abbrev('PRINT', 'PRINT')
say abbrev('PRINT', 'PRI'   )
say abbrev('PRINT', ''      )
```

... all say **1**

CENTER can be spelled properly, too:

```
say centre('goal kick', 25)
```

More Built-in Functions

COMPARE is often overlooked:

```
alpha='abcdefghijklmnopqrstuvwxyz'
say compare(alpha, 'abcdefghijklmnoqp')
```

... says 16

DATE lets you find the day-of-the week as a number:

```
say date('Base')//7
```

... says 0 for Monday, 1 for Tuesday, *etc.*

INSERT and OVERLAY

Powerful, when you need them:

```
say insert('needle', 'haystack', 3)
```

... says "hayneedlestack"

and...

```
say overlay('12:30', 'It is hh:mm', 7)
```

... says "It is 12:30"

Removing character(s) from a string

Use *SPACE* (with a little help from *TRANSLATE*):

```
string='Hoppy floppy'
string=translate(string, 'p', ' p')
string=space(string, 0)
string=translate(string, 'p', ' p')
```

... sets STRING to "Hoy floy"

For multiple characters, use (for example):

```
string=translate(string, 'a', ' aeiou')
string=space(string, 0)
string=translate(string, 'a', ' a')
```

Testing for parity

Use *SPACE* — with a little help from *X2B*, *TRANSLATE*, and *LENGTH:*

```
bits=x2b('C7')  /* 11000111 */
ones=translate(bits, ' ', '0')
ones=length(space(ones, 0))
parity=ones//2
```

... sets PARITY to "1"

SUBSTR, LEFT, and RIGHT

SUBSTR or *LEFT* can take a pad character:

```
say substr('Fred', 1, 8, '?')
say left('Fred', 8, '?')
```

... both say "Fred????"

RIGHT can pad on the left, or return rightmost characters:

```
say right(12, 6, 0)
say right('e:\extra.cmd', 3)
```

... says "000012" and "cmd"

TRANSLATE

As well as character substitution, *TRANSLATE* can be used to reformat (lay out) strings:

```
in='abcdefgh'
pattern='gh.ef.abcd'
say translate(pattern, '19940827', in)
```

... says "27.08.1994"

VERIFY

VERIFY can look for "the odd one in", as well as "the odd one out":

```
say verify('123.456', '0123456789')
```

... says "4"

but...

```
say verify("It's 1994", '13579', 'Match')
```

... says "6"

And finally...

```
/* Shuffle the numbers in range 1->max */
shuffle: procedure
  signal off novalue
  max=arg(1)
  out=''
  do i=1 to max
    sub=random(i,max)
    out=out substr(ar.sub,4)
    if sub=i then iterate
    ar.sub=ar.i
    end
  return out
```

IBM Views on REXX

James Crosskey
IBM

IBM Views on REXX

James Crosskey
REXX Development
IBM Endicott, NY
Internet id: crosskey@gdlvm7.vnet.ibm.com

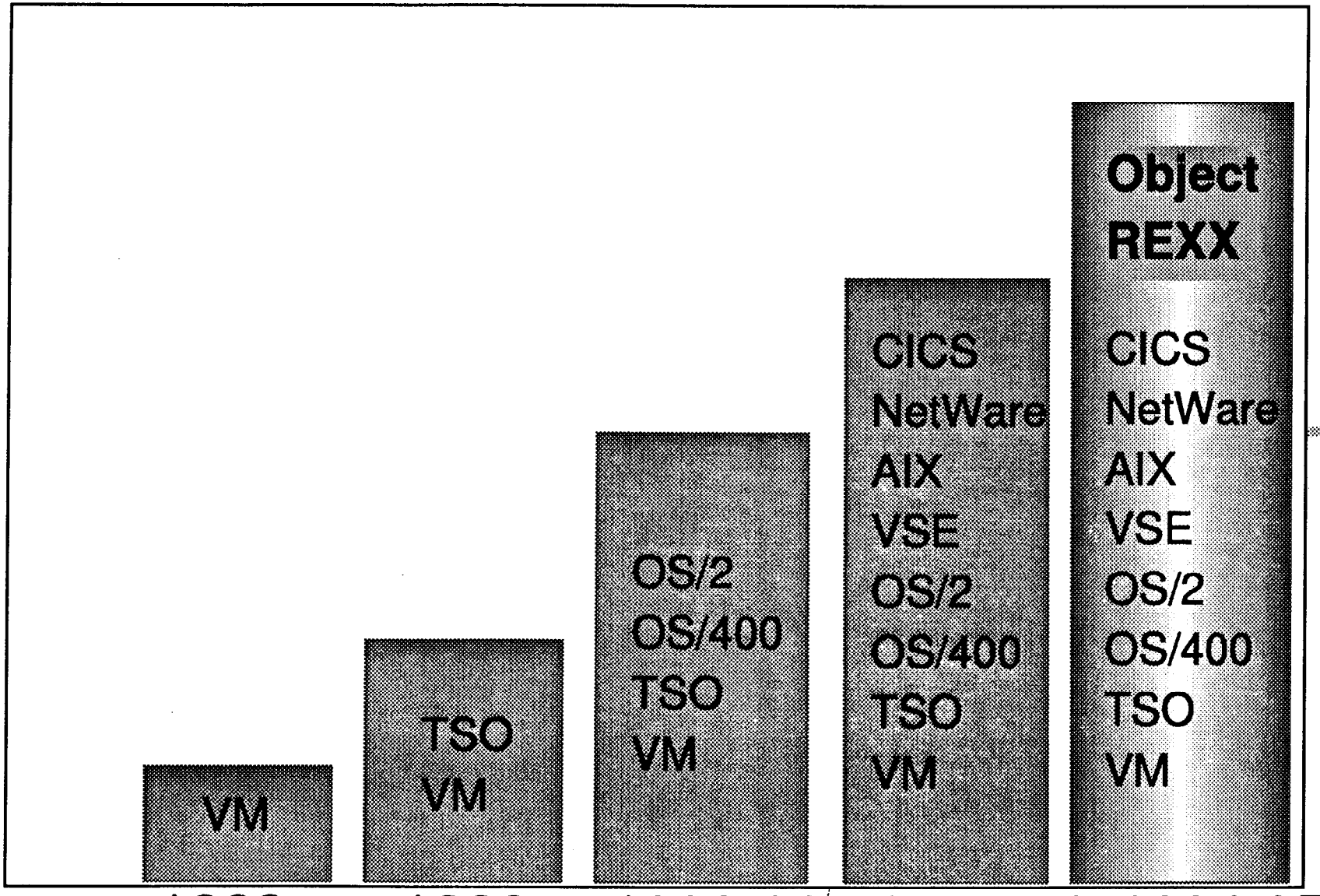
REXX Excitement!

- Customer support of REXX:
  - SHARE, GUIDE, COMMON, SEAS
  - REXX Symposium, OS/2 Technical Interchange, OS/2 Technical Conference
  - REXX Language Association
- REXX books
  - 40+ REXX books and manuals
  - 9 written in last year alone
  - Most recent include:
    - *REXX Reference Summary Handbook by D.Goran*
    - *Mastering OS/2 REXX by G.Gargiulo*
    - *Application Development Using OS/2 REXX by A.Rudd*

More REXX Excitement!

- Trade Press articles
  - 10+ articles in 1st quarter '94
  - Publications include:
    - *PC Week, Dr.Dobb's Journal, OS/2 Computing, OS/2 Professional,PC Magazine, Byte, OS/2 Monthly,...*
- Industry enthusiasm for REXX
  - *20+ implementations of REXX on many platforms*
- Visual programming with REXX
  - *VX REXX, VisProREXX, GpfREXX,...*
- ANSI Committee close to a REXX standard
- Enthusiastic response to IBM Object REXX beta program

IBM REXX Implementations



IBM REXX on AIX and NetWare

- IBM AIX REXX/6000 (5764-057)
  - Available 12/93
  - Port of IBM's OS/2 REXX kernel to the AIX platform (3.2.5 release and up)
  - "AS-IS" release
- IBM REXX for NetWare (5764-075)
  - Available 3/94
  - Port of IBM's OS/2 REXX kernel to Novell's NetWare platform (3.11, 3.12, or 4.0 releases)
  - "AS-IS" release

IBM CICS and VSE REXX

- IBM REXX for CICS/ESA (5655-086 Development System, 5655-087 Runtime Facility)
  - Available 4/94
  - Port of IBM's TSO/E REXX kernel to CICS/ESA (3.2.1 or 3.3 releases)
  - Development System:
    - *Editor, File System, Panel interface*

- IBM REXX/VSE (5686-058)
  - Available 9/93
  - Port of IBM's TSO/E REXX kernel to VSE/ESA
  - Supports both compiled and interpreted REXX programs

REXX Future Directions

- Wide range of platforms
- Apply new technology:
  - *Object Oriented programming (SOM, DSOM, etc)*
  - *Visual Programming tools*
  - *OpenDoc*
- Develop REXX function packages to expand presence:
  – Communications, MultMedia, Database,...
- Encourage use of REXX as the "application extender"
- Improve documentation
  - *Primer/on-line tutorial*
  - *Books, books and more books*
- Expand user base
  - *Students, non-"glass house" and OS/2 users*

The Open Scripting Language

Coming to
a system
near you!

Trademarks

Choosing a Command Language—
An Application-Centric Approach

Hal German
GTE

Choosing a Command Language -- An Application-Centric Approach

Hallett German

GTE Laboratories, Incorporated.

An Introduction to this paper

For over four years, the author has discussed a means for beginning and intermediate command language users to quickly choose the essential elements of their application without using a single piece of code. This paper is the first time the approach has been presented to the movers and shakers of the REXX world. It supplements the presentation by covering the following:

1. Why use such an approach?
2. Concepts behind the approach.
3. The approach itself
4. Conclusions
5. References

The presentation at the REXX Symposium will provide an overview of the approach, as well as an example of how to use, and include other factors to consider. Handouts can be obtained by contacting the author.

Why use such an approach?

In "the old days" it was easy. You used a mainframe host that had one command language and one editor (that usually had ties to the command language). Then PCs and UNIX systems snuck in from somewhere and the issues became more complex. There were more than one command language and editor to choose from. Programs could run on more than one operating system (and simultaneously if needed). Unfortunately, the theories and software practices for command languages were not enhanced to match the new realities. The approach listed below is a modest attempt to provide command language developers a strategy to deal with the new realities so they don't have to say "What do I do next?"

Concepts behind the approach

1. **What is a command language?**

Unfortunately, we only can briefly look at this area. My definition of a command language is the following:

A programming language consisting of a series of high-level English-like commands entered interactively (e.g., a keyboard, mouse, or other input device) or non-interactively (that is created with an editors, saved in a file, and executed in foreground or background). An interpreter or compiler for the command language then determines which user-specified operating system tasks to perform and processes them using corresponding task values.

Whew! A real mouthful. So what does it mean?

* Command languages are almost always interpreted languages. (REXX is one of the exceptions to this.)

* Command languages are usually executed in foreground. (Again, REXX is one of the exceptions.)

* Command languages are comprised of English-like verbs describing the task to perform. REXX is typical with instruction keywords like SAY and PULL.

* Command language provides a mean to directly or indirectly access the operating system. REXX shines in this area with the ADDRESS instruction and the environment model.

* Command languages offer user and third-party extensions. For REXX this includes functions, sub procedures, and interfaces to external environments.

2. Identifying the types of command language applications

In their CLIST manual, IBM talked about three types of command language applications. My eight years of working with various command languages have verified that this typology is a good match for the type of applications found in the real world.

These types are the following:

Front-end -- Also called "housekeeping" applications. In this case, the command language sets up the proper environment for an application to execute. This could be allocating files, creating files, or creating environment variables. They also can receive output from or send input to the application. I view the startup or login programs as a special example of a front-end application.

System and Utility -- This is like front-end command language application. However, the emphasis is on doing system tasks (such as backing up files) and utility operations (Such as being a function/sub procedure that performs a date operation.)

Self-contained -- The other two types are "blue-collar" applications. The "white-collar" application type is the self-contained application. It provides a dialog with the user (usually full-screen) while maintaining strict control over the process.

3. The Command Language Component

The last and most important piece of the puzzle is the command language component. All command languages that have examined to date have the following components:

* Input/Output (File operations, Stack operations, Output to the screen, Input from the keyboard)

* Flow Control (Conditional, Loop, Exception handling, Exit and return codes, Array operations)

* General Features (Debugging, Symbolic Substitution, Labels, Global Options, Numeric format, Interpreter Version)

* Interfaces (Internal functions, interface to operating system and external environments)

* Built-in (Functions, and System variables)

What the approach does is combine all three of the above elements. First determine your type of application, once you know that, you know the command language components that are usually used by that application type.Finally look up the commands corresponding to that command language component. **And not a single piece of code has been yet been written.**

The approach is application-centric because it encourages you to know your application requirements and data as much as possible before starting to code.

The approach

The following are the steps of the approach:

1. What type of application do I have?

   The three types were discussed above.

2. Which command language should I use?

   This is discussed in the presentation. This includes a look at the following:

   * Type of data
   * File type
   * benchmarks
   * Ease of use vs. power
   * features

3. Which command language components should I use?

   The components were listed above.

4. Which command language match these components?

   This is the crucial step. Table 1 lists a summary of the components.

5. Which command language components match these commands?

   Space does not permit listing this step. However, tables with this information can be found in the references section.

6. Where can I find more about these commands?

   There are many places you can learn about a command language command. These include: user guides, books, on-line references, summary references, electronic information servers, electronic mailing lists, user groups, and colleagues.

7. Do I need third-party extensions?

   Third-party extensions should be used in the following situations:

   * When portability is not a concern.

   * When the third-party extension performs an operation not found in the command language such as network and database operations.

   * When you can afford the run-time license costs for distributing the extension.

   * When the extension greatly enhances the look and feel of the application. Such as any of the "Visual REXXes."

Conclusions

I hope that this will be of use to you the next time that you are considering developing a command language application. I encourage others to look into this area.

Getting in touch with me

Hallett German
GTE Laboratories Inc
40 Sylvan Road
Waltham, Ma 02254
617-466-2290
hhg1@.gte.com

References

German, Hallett **Command Language Cookbook**, VNR 1992
[The approach is covered in detail. Looks at many different type of REXX implementations.]

German, Hallett **OS/2 2.1 REXX Handbook**, VNR 1994
[A Rexx tutorial and the approach with some enhancements.]

Table 1 Command Language Components by Application Type

Front-end
- Operating System Commands
- External Interfaces
- Input/Output: File operations & Command Line operations.
- Built-In Functions
- Flow Control: Conditional

System/Utility
- Operating System Commands
- External Commands
- Internal Commands
- Input/Output: File/Screen Operations
- Command line input
- Built-in Functions: String Operations
- Flow Control: Loops
- General: Batch Operations, Arrays

Self-contained
- External Commands
- Functions/Sub-procedures
- Input/Output: Command Line operations, user validation
- Flow control: Multiple conditions
- Built-in Functions: Text Case & string.
- General (Interactive operations, arrays)

News From the REXX Compiler

Klaus Hansjakob
IBM

Dr. Klaus Hansjakob

IBM Vienna Software Development Laboratory
Lassallestrasse 1
A-1020 Vienna, Austria
Europe

HANSJAKO@VABVM1.VNET.IBM.COM
ATIBMCXP at IBMMAIL

(+431) 21145-4243

Agenda

- News from the REXX Compiler
- Packaging an application
  - General considerations
  - DLINK
  - Function packages

Compiler

IBM Compiler and Library for SAA
REXX/370
Release 2

5695-013 5695-014

Available for CMS and MVS
Library is part of REXX/VSE

Alternate Library PTFs

	CMS		MVS	
	PTF	APAR	PTF	APAR
Compiler	UN51503 UN51504 JPN	PN48015	UN51833 UN51834 ENU UN51835 JPN	PN48006
Library	No PTF, additional product tape			

Introduction of Copyright, Alternate Library

Copyright

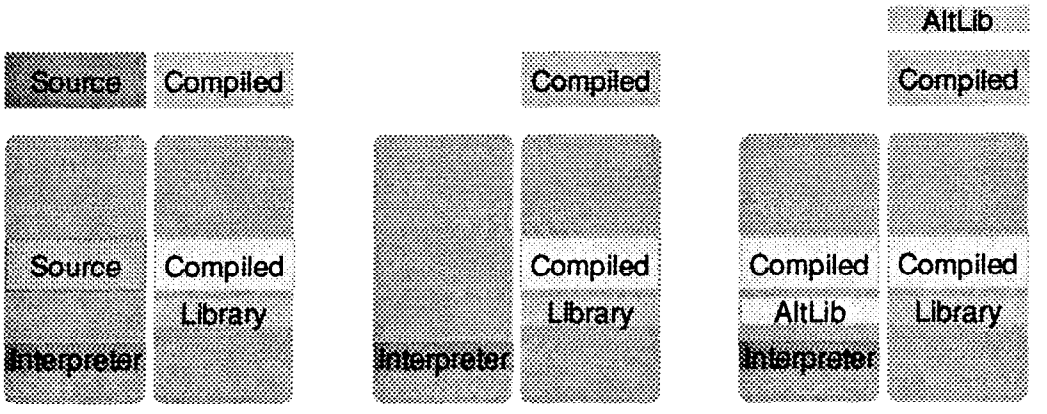
```
/* This program welcomes you ... */

/*%COPYRIGHT This program copyrighted for */
/*%COPYRIGHT MY Company, Vienna, Austria */

say 'hello world'
```

```
ESD             HELLO           á                                       00000001
TXT             á  èEXECPROCEAGRTPRC Compiled REXX   2.0   15 Mar 1994 100000002
TXT             3:46:29 CMS REXXC370 3.48 28 May 1993 PTF UN51503       ç00000003
TXT                ù° }  ¬l^&%  §j  ^  á  &      &  j   ²á  &µ  -     á0&¡  0   i0  00000004
TXT   y            ó  ¹10&  á)&2  ¹j   ²á  &Q   &ó     i       âh-        á10{  á\0 00000005
TXT   \            &      âá  &    &    q  }  ú                  This program cop00000006
TXT             yrighted for MY Company, Vienna, Austria                800000007
TXT   &         b          °            É                 a      ÷  {  a  É  -   00000008
TXT   h         ~*  à      @  h  m  µ  *         é      É     A  A  ~       ÷ 00000009
TXT   {          /  a      a  A          ~  a  £  0      Jê  ç  Å é      a  ~00000010
TXT   8            ÷  aá      à                £  ÷          £     é  ~  J  a  00000011
TXT             ~:  %  H  è          Æ J  1# £        z  {   N  l  {  á  0  l00000012
TXT   Ç           ^  {  )o     a^  0  0  ¡ J%*j   -     £       e  l  -  -o °00000013
TXT   µ         l  ö;     A é'     ^    H   R µ   É0 }E  K;°E &G °N &E jû É÷ 00000014
TXT   Q         aZ  I     /  &U     k  ;°R &X  C  7      .  8  2 \M  y  1 °900000015
TXT                ÉK;°S  I  T °F °C ¥       éò    *(0 A2    *1Z*£  k  ;ê. )00000016
TXT   ç         S °G         &S  L ÉH ~  Bë      \       ¥à  ê¹ êÇ k  ¥% K> 2  00000017
TXT              ê  è  ëî  êl  k:  ¥@  K= 2    b  d êf  êh  k« ¥ï   á      0  2m  1á00000018
TXT   ½           j      }  °   ^   bÆ  8         /Z  B¡  }   J  â^  °è     B¶ âÆ {00000019
TXT   0         u £á     ÷¡  -   /    °   j   !  /   âI   ô     /    M  L  j¬  ÷Ñ  èà 00000020
TXT             ~H   ¡   z  /ê  a   ~   B    n                              00000021
END                              1569501301 010094074                       00000022
```

Alternate Library

- Compile program
  - with ALTERNATE and SOURCELINE
    - use CONDENSE to hide source
    - DLINK does not work
- Distribute Alternate Library
  - without royalties, without paperwork
- Alternate Library
  - is installed on systems without Library
  - invokes interpreter when compiled program is run

Distributing REXX



Library optional	required	optional
Source exposed	not exposed	hidden (CONDENSE)
Maintenance problems	no problems	no problems
Packaging few options	all options	some options

What's missing

- TRACE support
- REXX I/O for CMS
- INCLUDE facility
- MARGINS

Agenda

- News from the REXX Compiler
- Packaging an application
  - General considerations
  - DLINK
  - Function packages

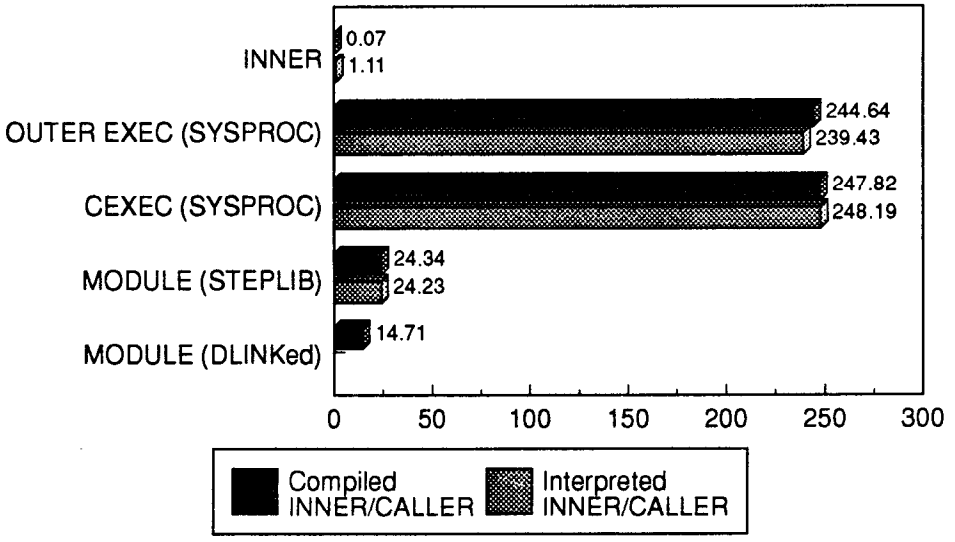
Performance

```
/* REXX Inner */
cvi=sysvar('SYSCPU')
Do i=1 To 1000
  Call inner
End
cve=sysvar('SYSCPU')
Say 'CPU time' cve-cvi
Exit

inner:Procedure
Return
```

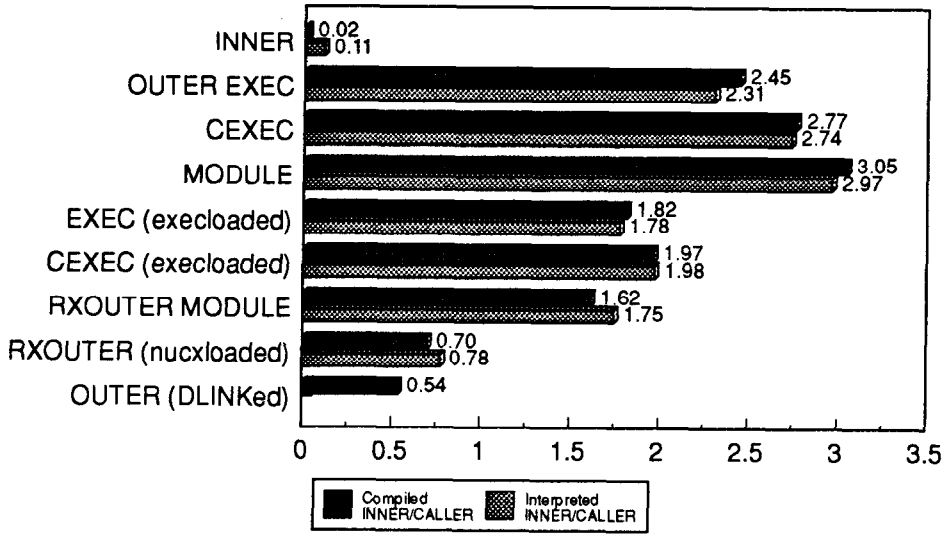
```
/* REXX Caller */
cvi=sysvar('SYSCPU')
Do i=1 to 1000
  Call outer
End
cve=sysvar('SYSCPU')
Say 'CPU time' cve-cvi
Exit
```

```
/* REXX Outer */
Return
```

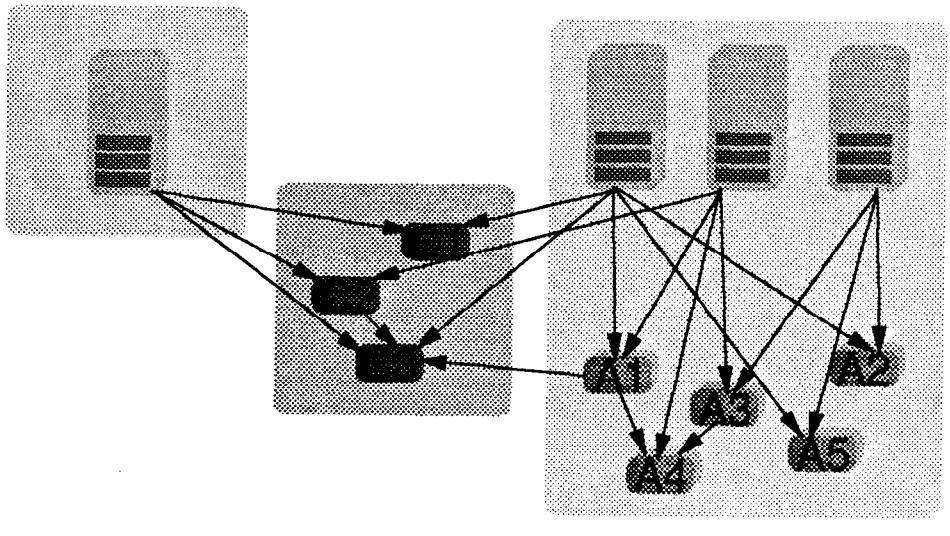
Performance (MVS)



Performance (CMS)



Applications



Internal - External

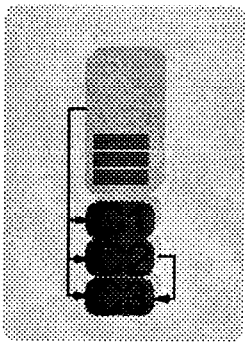
	Internal	External
Performance	good	bad
Maintenance	many	one
Distributed development	impossible	possible
Pieces	one	many
Search order	certain	uncertain
Variable sharing	easy	hard

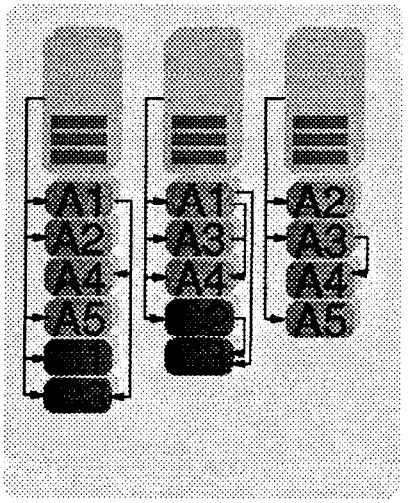
Include (not yet available)

Programming, DLINK, Function packages

DLINK, Function packages

DLINKed Applications





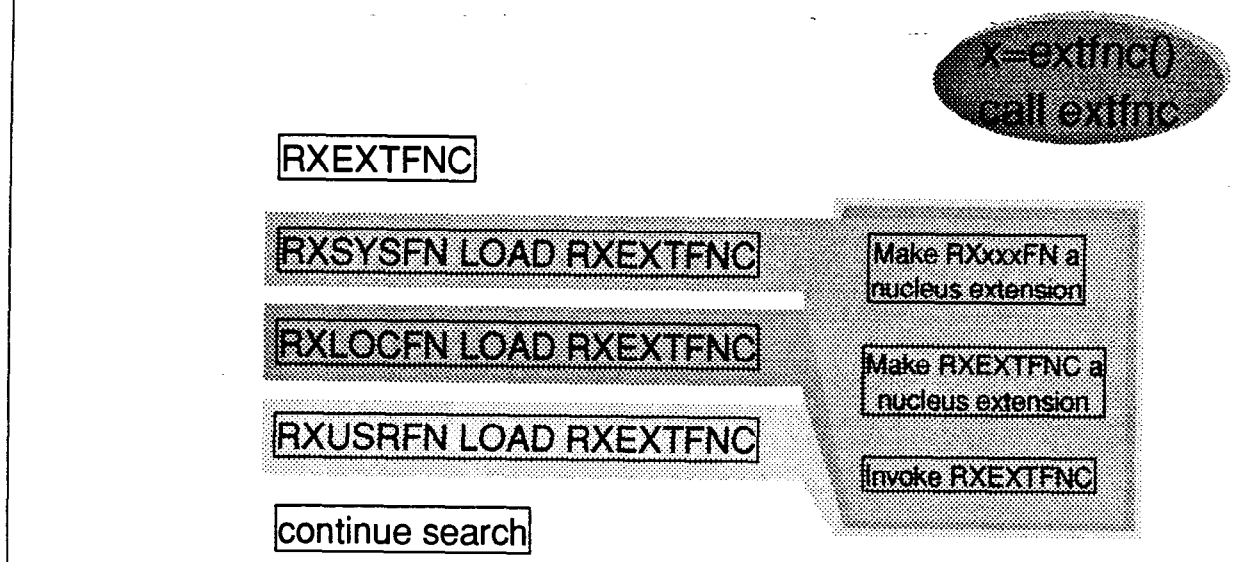
DLINK

- Search overhead zero
- Requires Compiler
- Does not work with Alternate Library

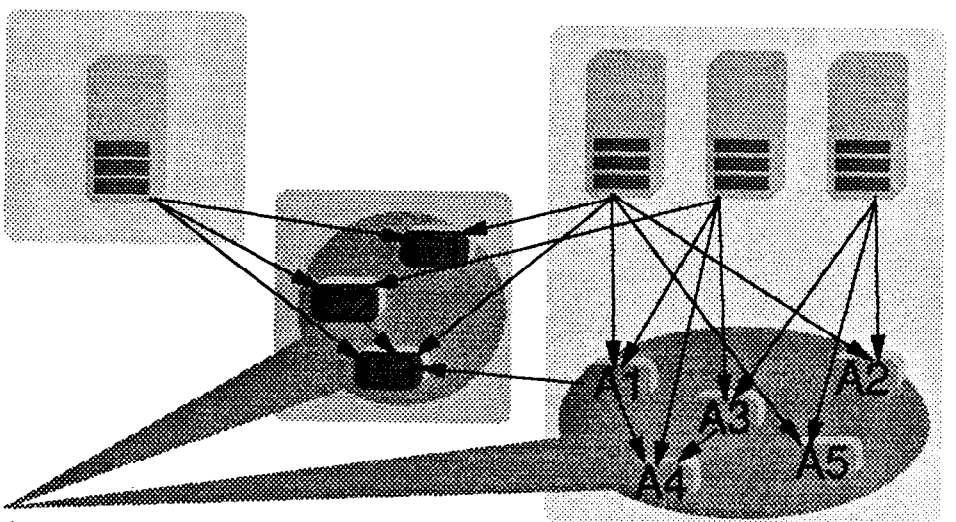
Function Packages

- Commonly used functions
- Early in the search order
- Functions must understand REXX function invocation

Search Order (CMS)



Applications with Function Packages



Function packages

Function Packages

- First in search order
- Compiler allows to write functions in REXX
- Works with Alternate Library
- May require explicit loading/unloading on CMS
- DLINK may be used when Alternate Library is not used

CMS Function Package Example

- Two files
  - RXUSERFN is function package loader
  - USERFN is function package glue code
- "Glue" code for a function package
  - Use without royalties
  - Allows free naming of function package
    - Requires renaming of files
    - Explicit loading of package and all functions with "RXmyname LOAD"
    - Explained in RXUSERFN header

Necessary Modifications

RXUSERFN ASSEMBLE

```
&PACKAGE    SETC   'USERFN'          Name of the package to load
&RXPACK     SETC   'RX&PACKAGE'      Name of this program
&CR(1)      SETC   'My Copyright'    Copyright notice
&CR(2)      SETC   'second line'     Copyright notice continued
```

USERFN ASSEMBLE

```
&PACKAGE    SETC   'USERFN'          Name of the package
&CR(1)      SETC   'My Copyright'    Copyright notice
&CR(2)      SETC   'second line'     Copyright notice continued
&FUN(1)     SETC   'USER1'           Name of function
&FUN(2)     SETC   'USER2'           Name of function
&FUN(3)     SETC   'USER3'           Name of function
```

Obtain the Source Code

- Email - write a note to
  - hansjako@vabvm1.vnet.ibm.com
  - ATIBMCXP at IBMMAIL
- Disk - get one
  - as long as supply lasts
  - if you can't use email
  - if you have a way to upload code
- Supplement
  - type in

Agenda

- News from the REXX Compiler
  - Copyright
  - Alternate Library
- Packaging an application
  - General considerations
  - DLINK
  - Function packages
  - Function package example

News from the REXX Compiler - Supplement

Klaus Hansjakob

IBM Vienna Software Development Lab
Wien 2, Lassallestrasse 1

c/o IBM Austria
Obere Donaustrasse 95
A-1020 Austria
EUROPE

HANSJAKO@VABVM1.VNET.IBM.COM

(+431) 21145-4243

May/95

Function Packages

```
         TITLE 'RXUSERFN * REXX Function Package Loading Stub'          00010000
***********************************************************************00010100
*                                                                     *00010200
*       Describe your function package here                           *00010300
*                                                                     *00010400
***********************************************************************00010500
*                                                                      00010600
&PACKAGE SETC 'USERFN'              Name of the package to load        00010700
*                                                                      00010800
&RXPACK  SETC 'RX&PACKAGE'          Name of this program               00010900
*                                                                      00011000
&CR(1)   SETC '                     Copyright notice                   00011100
*                                                                      00011200
                                                              EJECT    00011300
****Start of Specifications*************************************      00020000
*                                                                     *00030000
*       This code is provided on an as-is basis.                      *00040000
*                                                                     *00050000
***********************************************************************00060000
*                                                                      00070000
* Module name:      RXUSERFN                                           00080000
*                                                                      00090000
* Descriptive name: REXX function package loader                       00100000
*                                                                      00110000
* Function:                                                            00120000
*                                                                      00130000
*   NUCXLOAD module USERFN as RXUSERFN; when invoked with a LOAD       00140000
*   request invoke USERFN with same PLIST as on entry.                 00150000
*                                                                      00160000
*   To generate:                                                       00170000
*      MASM RXUSERFN                                                   00180000
*      LOAD RXUSERFN ( ORIGIN TRANS                                    00190000
*      GENMM RXUSERFN                                                  00200000
*                                                                      00210000
*   Note: RXUSERFN is interrogated as part of the REXX search          00210010
*   order for external functions. The functions are loaded             00210020
*   automatically when invoked. You can also create a function         00210030
*   package with a different name (e.g. RXMYPKG):                      00210040
*     Rename this file to RXMYPKG.                                     00210050
*     Change the macro variable PACKAGE in the header of this          00210060
*       file from USERFN to MYPKG.                                     00210070
*     Rename the file containing the functions to MYPKG.               00210080
*     Change the macro variable PACKAGE in the header of the           00210090
*       file containing the functions to MYPKG.                        00210100
*   When the function package is not interrogated as part of           00210110
*   the REXX search order you must load the functions in the           00210120
*   package explicitely ('RXMYPKG LOAD RXaaaaaa' where aaaaaa          00210130
*   is the name of the function) or globally ('RXMYPKG LOAD')          00210140
*   before you can invoke them.                                        00210150
*   To drop all functions of a package issue a NUCXDROP command        00210160
*   for the package ('NUCXDROP RXMYPKG').                              00210170
*                                                                      00220000
```


Function Packages ...

```
* Entry/exit conditions:                                               00230000
*                                                                      00240000
*     NOTE: The MODULE is generated as a transient module.             00250000
*                                                                      00260000
*     Entry:                                                           00270000
*       Standard SVC conventions.                                      00280000
*       R1 points to a tokenized PLIST (SVC 202 linkage).              00290000
*                                                                      00300000
*     Exit:                                                            00310000
*       R15 = 0  USERFN successfully loaded and returned with 0        00320000
*       R15¬= 0  - Return code from unsuccesful NUCXLOAD USERFN         00330000
*                - Return code passed back from USERFN after           00340000
*                  invocation with original PLIST                      00350000
*                - 4 to indicate bad PLIST                             00360000
*                                                                      00370000
* Exit:                                                                00380000
*   Return to caller.                                                  00390000
*                                                                      00400000
* Operation:                                                           00410000
*   When invoked without argument NUCXLOAD USERFN as RXUSERFN,         00420000
*   and pass back the return code obtained from NUCXLOAD.              00430000
*   When invoked with 'LOAD' as the first argument then NUCXLOAD       00440000
*   USERFN as RXUSERFN, invoke RXUSERFN with the same PLIST as         00450000
*   obtained on entry, pass back the return code given back by         00460000
*   USERFN.                                                            00470000
*   Otherwise display message and return with return code 4.           00480000
*                                                                      00490000
* Maclibs:                                                             00500000
*   CMSLIB or DMSGPI                                                   00510000
*                                                                      00520000
* Macros and control blocks:                                           00530000
*   REGEQU                                                             00540000
*                                                                      00550000
* Change Activity:                                                     00560000
*   91-11-21  KH  Cleanup and comments                                 00570000
*   93-06-28  KH  Make function package name a macro variable, add     00570010
*                 copyright as macro variable.                         00570020
*                                                                      00580000
****End of Specifications*************************************         00590000
*                                                                      00600000
&RXPACK  RMODE 24                   Must be loaded below 16MB!         00610000
&RXPACK  AMODE 24                   Expects SVC 202 linkage            00620000
*                                                                      00630000
&RXPACK  CSECT ,                                                       00640000
*                                                                      00650000
*   Make sure this is a LOAD                                           00660000
*                                                                      00670000
         USING *,R12                Establish addressability           00680000
         B     STARTCOD             Branch around header               00680100
         DC    CL8'&RXPACK'         Package ID                         00680200
&MAX     SETA  N'&CR                                                   00690300
&I       SETA  1                                                       00680400
```


Function Packages ...

```
.CRLOOP  ANOP                                                          00680500
         AIF   (&I GT &MAX).CRLOOPE                                    00680600
         DC    C'&CR(&I)'                                              00680700
&I       SETA  &I+1                                                    00680800
         AGO   .CRLOOP                                                 00680900
.CRLOOPE ANOP                                                          00681000
*                                                                      00681100
STARTCOD DS    0F                                                      00681200
         LR    R10,R14              Save return address                00690000
         SR    R2,R2                Assume install only                00700000
         CLI   ARG1(R1),X'FF'       Any arguments ?                    00710000
         BE    GOLOAD               Br if not - go install             00720000
         CLC   ARG1(8,R1),=CL8'LOAD'  Is this explicit load?           00730000
         BNE   BADPL                Br if not - go complain            00740000
         LR    R2,R1                Keep invocation PLIST              00750000
*                                                                      00760000
*   NUCXLOAD USERFN as RXUSERFN.                                       00770000
*                                                                      00780000
GOLOAD   EQU   *                                                       00790000
         LA    R1,NUCXLOAD          Address NUCXLOAD Plist             00800000
         SVC   202                                                     00810000
         DC    AL4(1)               Return even if error               00820000
         LTR   R15,R15              Did load work?                     00830000
         BNZR  R10                  No, pass back rc                   00840000
*                                                                      00850000
*   If explicit load requested pass through invocation PLIST           00860000
*                                                                      00870000
         LTR   R1,R2                explicit load?                     00880000
         BZR   R10                  No, return                         00890000
         SVC   202                  Invoke nucleus extension           00900000
         DC    AL4(1)                                                  00910000
         BR    R10                  Return, pass through rc            00920000
*                                                                      00930000
*   Error handling routines.                                           00940000
*   Note that in order to avoid the generation of relocatable          00950000
*   address constants, the TYPLIN PLIST is "hand-built" rather         00960000
*   than using WRTERM.                                                 00970000
*                                                                      00980000
BADPL    EQU   *                    Something's wrong with PLIST       00990000
         LA    R1,MSG1              Get message address                01000000
         LA    R2,L'MSG1            Get message length                 01010000
         STCM  R1,B'0111',TYPBUFF   Set it in PLIST                    01020000
         STH   R2,TYPLEN            Set it in PLIST                    01030000
         OI    TYPLIN+13,X'40'      Request error message edit         01040000
         LA    R1,TYPLIN            Point at PLIST                     01050000
         SVC   202                  Give it to CMS                     01060000
         DC    AL4(1)               Ignore errors                      01070000
         LA    R15,4                Set non-zero return code           01080000
         BR    R10                  Return                             01090000
         DROP  R12                                                     01100000
*                                                                      01110000
TYPLIN   DC    CL8'TYPLIN',X'01',AL3(0),C'B',X'00',AL2(0)              01120000
```

Function Packages ...

```
TYPBUFF  EQU   TYPLIN+9,3                                          01130000
TYPLEN   EQU   TYPLIN+14,2                                         01140000
MSG1     DC    C'DMSRUF870E Invalid parameter'                     01150000
*                                                                  01160000
NUCXLOAD DS    0D                                                  01170000
         DC    CL8'NUCXLOAD'                                       01180000
         DC    CL8'&RXPACK'          Name to load as               01190000
         DC    CL8'&PACKAGE'         Name of module to load        01200000
         DC    CL8'('                                              01210000
         DC    CL8'SYSTEM'           System attribute              01220000
         DC    CL8'SERVICE'          Service attribute             01230000
         DC    CL8'PUSH'             Push down existing extension  01240000
         DC    8X'FF'                                              01250000
         ORG   ,                                                   01260000
*                                                                  01270000
         LTORG                       Literal pool                  01280000
*                                                                  01290000
ARG1     EQU   8,8                   First argument                01300000
         REGEQU                                                    01310000
         END                                                       01320000
```


Function Packages ...

```
         TITLE 'USERFN    REXX Function Package Glue Code        '  00010000
         LCLC  &PACKAGE,&CR(1),&FUN(1)                             00020000
*************************************************************      00030000
*                                                           *      00040000
*       Describe your function package here                 *      00050000
*                                                           *      00060000
*************************************************************      00070000
*                                                                  00080000
&PACKAGE SETC  'USERFN'             Name of the package           00090000
*                                                                  00100000
&CR(1)   SETC  ' '                  Copyright notice              00110000
*                                                                  00120000
&FUN(1)  SETC  'USER1'              Name of function              00130000
&FUN(2)  SETC  'USER2'              Name of function              00140000
&FUN(3)  SETC  'USER3'              Name of function              00150000
*                                                                  00160000
                                                          EJECT    00170000
*****Start of Specifications*****************************        00180000
*                                                           *      00190000
*       This code is provided on an as-is basis.            *      00200000
*                                                           *      00210000
*************************************************************      00220000
*                                                                  00230000
* Module name:      USERFN                                         00240000
*                                                                  00250000
* Descriptive name: Glue code for REXX function package            00260000
*                                                                  00270000
* Function:                                                        00280000
*                                                                  00290000
*   The following code resides in free storage and is capable      00300000
*   of replying to LOAD and RESET.                                 00310000
*   A LOAD call results in identifying the function (whose name    00320000
*   is passed as parameter following LOAD) as entry point.         00330000
*   A LOAD call without function specified will identify all       00330010
*   functions in the function package as entry points.             00330020
*   A RESET call from NUCXDROP will turn the functions off.        00340000
*   A PURGE service call is ignored.                               00350000
*                                                                  00360000
*   To generate                                                    00370000
*      HASM USERFN                                                 00380000
*      LOAD USERFN                                                 00390000
*      GENMM USERFN ( NOMAP                                        00400000
*                                                                  00410000
*   Note: To get a MODULE which can execute on CMS release 5 and   00420000
*         later --> generate the module on a CMS release > 5.      00430000
*   Note: When user functions are compiled REXX with the DLINK     00440000
*         compiler option, then you must explicitely specify       00450000
*         the functions to be DLINKed explicitely on the LOAD      00460000
*         command, and you must use the RLDSAVE option for the     00470000
*         LOAD command.                                            00480000
*         Example:                                                 00490000
*           USER1, USER2, and USER3 are functions in the package.  00500000
```


Function Packages ...

```
*           USER1 contains a call to USER3 and a call to USERX.    00510000
*           - compile USER1 with the DLINK compiler option         00520000
*           - LOAD USERFN USERX ( RLDSAVE                          00530000
*           - GENMM USERFN ( NOMAP                                 00540000
*           If you do this on CMS rel > 5 you will get a module    00550000
*           runnable on CMS rel 5 and later, which will reside     00560000
*           above the 16 MB line if possible.                      00570000
*           USER1, USER2, and USER3 are the known functions.       00580000
*           Calls in USER1 to USER3 and USERX will use DLINK.      00590000
*                                                                  00600000
* Entry/exit conditions:                                           00610000
*                                                                  00620000
*      Standard SVC conventions.                                   00630000
*      R1 points to a tokenized PLIST.                             00640000
*      This code can run with AMODE 24 or AMODE 31                 00650000
*                                                                  00660000
*    Exit:                                                         00670000
*      R15 = 0  - Function successfully loaded or exists           00680000
*               - Functions unloaded                               00690000
*               - Service call, Endcmd call, ...                   00700000
*      R15¬= 0  - Return code from unsuccesful NUCEXT LOAD         00710000
*               - 1 LOAD or RESET, but no second argument         00720000
*                   or function not in package                    00730000
*               - 4 no first argument                              00740000
*                                                                  00750000
* Maclibs:                                                         00760000
*   DMSSP or DMSGPI                                                00770000
*                                                                  00780000
* Macros and control blocks:                                       00790000
*   REGEQU                                                         00800000
*   CMSIEVEL                                                       00810000
*   NUCON                                                          00820000
*                                                                  00830000
* Change Activity:                                                 00840000
*   91-11-21  KH  Added AMODE 31 capability                        00850000
*   93-06-22  KH  Added missing branch for service call with       00850010
*                 AMODE 24 (thanks to Roderic A. Davis for pointing 00850020
*                 out the problem and providing the solution).     00850030
*   93-06-28  KH  Load all functions in response to a LOAD request 00850040
*                 without function name specified.                 00850050
*                                                                  00860000
*****End of Specifications*******************************         00870000
                                                          SPACE 1  00880000
&PACKAGE RMODE ANY                                                 00890000
&PACKAGE AMODE 31                                                  00900000
                                                          SPACE 1  00910000
&PACKAGE CSECT ,                                                   00920000
         USING NUCON,8              Establish Addressability       00930000
         USING *,R12                Establish Addressability       00940000
         B     STARTCOD             Branch around header           00950000
         DC    CL8'&PACKAGE'        Package ID                     00960000
&MAX     SETA  N'&CR                                               00970000
```


Function Packages ...

```
&I       SETA  1                                                   00980000
.CRLOOP  ANOP                                                      00990000
         AIF   (&I GT &MAX).CRLOOPE                                01000000
         DC    C'&CR(&I)'                                          01010000
&I       SETA  &I+1                                                01020000
         AGO   .CRLOOP                                             01030000
.CRLOOPE ANOP                                                      01040000
*------------------------------------------------------------      01050000
* List of functions included in this pack, with their offsets     01060000
*                                                                  01070000
FUNLNAME EQU   4,8                 Offset & length of name       01080000
FUNOFFS  EQU   0,4                 Offset to the routine         01090000
LENTRY   EQU   12                  Length of a single entry      01100000
FUNLIST  DS    0F                  List of functions             01110000
&MAX     SETA  N'&FUN                                              01120000
&I       SETA  1                                                   01130000
.FULOOP  ANOP                                                      01140000
         AIF   (&I GT &MAX).FULOOPE                                01150000
         EXTRN &FUN(&I)            Name of compiled program        01160000
         DC    A(&FUN(&I)-&PACKAGE)  Offset of compiled prog       01170000
         DC    CL2'RX'             Name prefix                     01180000
         DC    CL6'&FUN(&I)'       Name                            01190000
&I       SETA  &I+1                                                01200000
         AGO   .FULOOP                                             01210000
.FULOOPE ANOP                                                      01220000
EFUNLIST DC    A(*-*)                End fence                     01230000
*                                                                  01240000
*------------------------------------------------------------      01250000
STARTCOD EQU   *                                                   01260000
         LR    R10,R14             Save return address           01270000
         LR    R11,R14             Save return address once more 01275000
         CLC   ARG1(8,R1),=CL8'LOAD'     Is this a load?          01280000
         BE    CHK4ARGS            Yes, check for any arg's      01290000
         CLC   ARG1(8,R1),=CL8'RESET'  Reset?                     01300000
         BE    DOOFF               Yes, turn off functions       01310000
         SR    R15,R15             In case of service call       01320000
         L     R2,=X'FF000000'     Set HOB of register           01330000
         LA    R2,0(,R2)           LA will clear HOB in AMODE 24 01340000
         LTR   R2,R2               Br if AMODE 24                01350000
         BZ    PL2021                                            01355000
         CLI   $6(R13),X'FE'       Look at Call Type (XA)        01360000
         B     CONTA1                                            01370000
PL2021   CLM   R1,8'1000',=X'FE'   Look at Call Type             01380000
CONTA1   EQU   *                                                   01390000
         BNLR  R14                 Return Service, Endcmd, ...   01400000
         LA    R15,4               Bad Plist, set error code     01410000
         BR    R14                 .. and return                 01420000
CHK4ARGS EQU   *                                                   01430000
         LA    R15,1               Set possible return code      01440000
         CLI   ARG2(R1),X'FF'      Any arguments passed?         01450000
         BE    ALLLOAD             No, load all functions        01460000
*                                                                  01470000
```

Function Packages ...

```
*   LOAD request. Check function name against FUNLIST.                  01480000
*   Only turn on the requested function.                                01490000
*                                                                       01500000
         PUSH  USING                    Save USING status               01510000
         USING DNUCX,R13                Use save area for PLIST         01520000
AUTOLOAD EQU   *                                                        01530000
         MVC   DNLIST(LNLIST),NLIST Move skeleton to work area         01540000
         LR    R3,R1                    Save old plist pointer          01550000
         LA    R4,LENTRY                Length of FUNLIST entry         01560000
         LA    R5,EFUNLIST              End of function table           01570000
         LA    R2,FUNLIST               Start of function table         01580000
         LA    R15,1                    Set error return code           01590000
CHECK1   EQU   *                                                        01600000
         CLC   ARG2(,R3),FUNLNAME(R2) Check against name               01610000
         BE    TURNON                   Found - turn function on        01620000
         BXLE  R2,R4,CHECK1             Loop for another check          01630000
         BR    R10                      Return with RC = 1              01640000
*                                                                       01640100
*   LOAD request without function name, load all functions in           01640200
*   package. Return with RC 0.                                         01640300
*                                                                       01640400
ALLLOAD  EQU   *                                                        01640500
         LR    R3,R1                    Save old plist pointer          01640600
         LA    R4,LENTRY                Length of FUNLIST entry         01640700
         LA    R5,EFUNLIST              End of function table           01640800
         SR    R5,R4                    Last entry in function table    01640900
         LA    R2,FUNLIST               Start of function table         01641000
NEXTF1   EQU   *                                                        01641100
         MVC   DNLIST(LNLIST),NLIST Move skeleton to work area         01641200
         BAL   R10,TURNON               Turn on the function            01641300
         BXLE  R2,R4,NEXTF1             Loop for another function       01641400
         SR    R15,R15                  Set ok return code              01641500
         BR    R11                      Return                          01641600
*                                                                       01650000
TURNON   EQU   *                                                        01660000
         MVC   DNLNAME,FUNLNAME(R2) Copy startup name                  01670000
         LA    R1,DNLIST                -> PLIST                        01680000
*                                                                       01690000
*   See if function is already a nucleus extension, make it             01700000
*   a nucleus extension if not (CMS rel > 5)                            01710000
*                                                                       01720000
         LA    R15,1                                                    01729000
         LNR   R15,R15                  -1                              01730000
         ST    R15,DNLADDR              Query form of NUCEXT plist      01740000
         CLI   CMSPROG,VMSP5            Are we on CMS release 5         01750000
         BNH   SV2021                   Br if yes, use SVC 202          01760000
         L     R15,=X'00000000'         tok Plist, COPY/FENCE flags     01770000
         SVC   204                                                      01780000
         LTR   R15,R15                  Exists?                         01790000
         BZR   R10                      Yes, immediate return           01800000
         L     R6,FUNOFFS(,R2)          Load address offset             01810000
         LA    R6,0(R6,R12)             True start address              01820000
```


Function Packages ...

```
         ST    R6,DNLADDR               Add to startup PSW              01830000
         L     R15,=X'00000000'         tok Plist, COPY/FENCE flags     01840000
         SVC   204                                                      01850000
         BR    R10                      Return                          01860000
*                                                                       01870000
*   See if function is already a nucleus extension, make it             01880000
*   a nucleus extension if not (CMS rel 5)                              01890000
*                                                                       01900000
SV2021   SVC   202                                                      01910000
         DC    AL4(1)                   Fall through if error           01920000
         LTR   R15,R15                  Exists?                         01930000
         BZR   R10                      Yes, immediate return           01940000
         L     R6,FUNOFFS(,R2)          Load address offset             01950000
         LA    R6,0(R6,R12)             True start address              01960000
         ST    R6,DNLADDR               Add to startup PSW              01970000
         SVC   202                                                      01980000
         DC    AL4(1)                   Ignore errors                   01990000
         BR    R10                      Return                          02000000
         POP   USING                    Restore USING status            02010000
*                                                                       02020000
*   RESET request: switch off functions                                 02030000
*                                                                       02040000
DOOFF    EQU   *                                                        02050000
         PUSH  USING                    Save USING status               02060000
         USING DNUCX,R13                Use save area for PLIST         02070000
         MVC   DNLIST(LNLIST),NLIST Move skeleton to work area         02080000
         LA    R5,FUNLIST               -> to list                      02090000
         LA    R1,DNLIST                -> PLIST                        02100000
         CLI   CMSPROG,VMSP5            Are we on CMS release 5         02110000
         BNH   SV2022                   Br if yes, use SVC 202          02120000
SV2042   EQU   *                                                        02130000
         L     R15,FUNOFFS(R5)          Any more to cancel?             02140000
         LTR   R15,R15                                                  02150000
         BZR   R10                      0 = all done ... Get out        02160000
         MVC   DNLNAME(8),FUNLNAME(R5) Copy startup name               02170000
         L     R15,=X'00000000'         tok Plist, COPY/FENCE flags     02180000
         SVC   204                                                      02190000
*        (we ignore errors e.g.: function already cancelled)            02200000
         LA    R5,LENTRY(,R5)           -> next item in FUNLIST         02210000
         B     SV2042                                                   02220000
*                                                                       02230000
SV2022   EQU   *                                                        02240000
         L     R15,FUNOFFS(R5)          Any more to cancel?             02250000
         LTR   R15,R15                                                  02260000
         BZR   R10                      0 = all done ... Get out        02270000
         MVC   DNLNAME(8),FUNLNAME(R5) Copy startup name               02280000
         SVC   202                                                      02290000
         DC    AL4(1)                   Ignore errors                   02300000
*        (we ignore errors e.g.: function already cancelled)            02310000
         LA    R5,LENTRY(,R5)           -> next item in FUNLIST         02320000
         B     SV2022                                                   02330000
         POP   USING                    Restore USING status            02340000
```


Function Packages ...

```
                                                        EJECT           02350000
*                                                                       02360000
*   Equates                                                             02370000
*                                                                       02380000
ARG1     EQU   8,8                      First argument                  02390000
ARG2     EQU   16,8                     Second argument                 02400000
         REGEQU                                                         02410000
         CMSLEVEL                                                       02420000
*                                                                       02430000
*   PLIST for invoking 'NUCEXT'  (setup as CANCEL PLIST)                02440000
*                                                                       02450000
NLIST    DS    0D                       NUCEXT Plist                    02460000
         DC    CL8'NUCEXT'              Name                            02470000
NLNAME   DC    CL8'&PACKAGE'            Function name                   02480000
         DC    X'FF'                    System mask enabled             02490000
NLKEY    DC    X'04'                    System key                      02500000
NLFLAG   DC    AL1(SYSTEM)              NUCEXT flag                     02510000
         DC    X'00'                    Spare flags                     02520000
NLADDR   DC    A(0)                     Entry point address             02530000
         DC    AL4(*-*)                 private                         02540000
NLSTART  DC    A(0)                     Start address                   02550000
NLLEN    DC    F'0'                     Length                          02560000
LNLIST   EQU   *-NLIST                  Length of list                  02570000
*                                                                       02580000
*   NUCEXT PLIST Flags:                                                 02590000
*                                                                       02600000
SYSTEM   EQU   X'80'                                                    02610000
*                                                                       02620000
*   DSECT for NUCEXT plist                                              02630000
*                                                                       02640000
DNUCX    DSECT                          Based on register 13            02650000
DNLIST   DS    CL8 'NUCEXT'             Name                            02660000
DNLNAME  DS    CL8 '&PACKAGE'           Function name                   02670000
DNLMASK  DS    X 'FF'                   Mask                            02680000
DNLKEY   DS    X '04'                   System Key                      02690000
DNLFLAG  DS    A(1) (SYSTEM)            NUCEXT flag                     02700000
         DS    X '00'                   Spare flags                     02710000
DNLADDR  DS    A                        Entry point address (0=cancel)  02720000
         DS    AL4 (*-*)                private                         02730000
DLSTART  DS    A                        Start address                   02740000
DLNLLEN  DS    AL4 (FREELEN)            Length                          02750000
         NUCON                                                          02760000
         END                                                            02770000
```

Using REXX as a Database Tool

Mark Hessling
Griffith University

Using REXX as a Database Tool

Mark Hessling

Griffith University
Brisbane, Australia

Introduction

Having been involved in Database Administration for the last 5 years, and having a long relationship with REXX (over 10 years) it was inevitable that the two should come together eventually.

GUROO History

During 1993, I found I needed a scripting tool to manipulate some data in Oracle tables as part of my Grad. Dip. course. I decided that it would be quicker to write an interface from REXX to an Oracle database, and then write the programs I required in REXX than it was to write the same programs in the tools supplied by Oracle. GUROO (Griffith University Rexx Oracle Overseer) was the result.

As I already had the basic framework, courtesy of the SAA REXX API in Regina, and the interface to Regina in THE, filling in the remainder of the tool was relatively simple.

The prime design consideration in GUROO was simplicity. I had a rough idea of the interface to existing REXX-SQL tools like the Database Manager in OS/2 Extended Edition. These interfaces seem too complicated. Compare the same program written using the REXX interface to Database Manager and the GUROO example. See Examples 1 and 2.

Example 1

```
/*-------------------------------------------------------------*/
/* Display the names of all tables owned by the default user.*/
/* DBM version.                                                */
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/* Load the DBM dynamic link libraries...                      */
/*-------------------------------------------------------------*/
If rxfuncquery('SQLDBS') \= 0 Then
   rcy = rxfuncadd('SQLDBS','SQLAR','SQLDBS');
If rxfuncquery('SQLEXEC') \= 0 Then
   rcy = rxfuncadd('SQLEXEC','SQLAR','SQLEXEC');
/*-------------------------------------------------------------*/
/* Connect to the SAMPLE database...                           */
/*-------------------------------------------------------------*/
Call sqlexec 'CONNECT TO sample IN SHARE MODE';
If ( SQLCA.SQLCODE \= 0) Then
   Do
     Say 'CONNECT TO Error: SQLCODE =' SQLCA.SQLCODE;
     Exit
   End
/*-------------------------------------------------------------*/
/* Prepare and declare the cursor for the SQL statement...     */
/*-------------------------------------------------------------*/
st = "SELECT name FROM sysibm.systables WHERE name <> ?";
Call sqlexec 'PREPARE s1 FROM :st';
Call sqlexec 'DECLARE c1 CURSOR FOR s1';
If ( SQLCA.SQLCODE \= 0) Then
   Say 'Error preparing statement: SQLCODE =' SQLCA.SQLCODE;
Else
/*-------------------------------------------------------------*/
/* Open the cursor associated with the SQL statement...        */
/*-------------------------------------------------------------*/
   Do
     parm_var = "STAFF";
     Call sqlexec 'OPEN c1 USING :parm_var';
/*-------------------------------------------------------------*/
/* Fetch and display each row selected...                      */
/*-------------------------------------------------------------*/
     Do While ( SQLCA.SQLCODE = 0 )
        Call sqlexec 'FETCH c1 INTO :table_name';
        If (SQLCA.SQLCODE = 0) Then
           Say 'Table = ' table_name;
        End
/*-------------------------------------------------------------*/
/* Close the cursor and end the transaction...                 */
/*-------------------------------------------------------------*/
       Call sqlexec 'CLOSE c1';
       Call sqlexec 'COMMIT';
   End
/*-------------------------------------------------------------*/
/* Disconnect from the database...                             */
/*-------------------------------------------------------------*/
Call sqlexec 'CONNECT RESET';
Return
```

Example 2

```
/*-------------------------------------------------------------*/
/* Display the names of all tables owned by the default user.*/
/* GUROO version.                                              */
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/* Connect to the SAMPLE database...                           */
/*-------------------------------------------------------------*/
if sql_connect('sample') < 0 Then
   Do
     Say sql_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Execute the select statement and return data...             */
/*-------------------------------------------------------------*/
st = "SELECT name FROM sysibm.systables WHERE name <> ?"
parm_var = 'STAFF'
if sql_command(q1,st,parm_var) < 0 Then
   Do
     Say sql_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Display each row selected...                                */
/*-------------------------------------------------------------*/
Do i = 1 To q1.name.0
   Say 'Table = ' q1.name.i
End
/*-------------------------------------------------------------*/
/* End the transaction...                                      */
/*-------------------------------------------------------------*/
if sql_command(q1,"COMMIT") < 0 Then
   Do
     Say sql_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Disconnect from the database...                             */
/*-------------------------------------------------------------*/
if sql_disconnect() < 0 Then
   Do
     Say sql_error_text()
     Exit
   End
Return
```

What Is GUROO

In its current original form, GUROO is a standalone program, written using Oracle's Pro*C and linked with Regina. GUROO is really 7 external functions:

- sql_connect()
  connect to an Oracle database
- sql_disconnect()
  disconnect from an Oracle database
- sql_command()
  execute an SQL command (select, update etc)
- sql_open_cursor()
  open a cursor
- sql_close_cursor
  close a cursor
- sql_fetch_row()
  fetch a row from a cursor
- sql_error_text()
  return the text of the last GUROO or Oracle error

For more details on the syntax of these functions, see the attachment.

Current Status

GUROO is currently in use solely as an internal tool within the Information Systems section of Griffith University. Many of the DBA tools are written using GUROO and the programming staff have also begun to use GUROO in situations where the Oracle supplied tools are inappropriate. In one instance, GUROO has replaced one function which was originally written using various combinations of C shell, awk, SQL*ReportWriter, SQL*Plus and SQL*Loader. The GUROO program is quicker, smaller and much easier to understand.

Despite being interpreted, the performance of GUROO programs is on par with other Oracle tools.

Currently, GUROO is not available for distribution outside of the Information Systems section of Griffith University.

Future Directions

As a result of Griffith University's reluctance to allow distribution of GUROO, I and a colleague of mine have begun an independent development of a similar tool; REXX/SQL. The structure and operation of REXX/SQL will be fundamentally the same as GUROO, but will also include most of the low-level functions like PARSE and EXECUTE that exists in current tools. This will give users the option of a simple interface or one which they are more familiar (for users of RXSQL, DB2 etc).

REXX/SQL will have the ability to make multiple connections to the same or different databases from the same vendor or different vendors. The ultimate goal for REXX/SQL will be to allow a programmer to access data from any combination of SQL databases as though all data were stored in the one database.

This can really only be achieved by writing each database access functions as dynamically linked libraries that can be loaded at run-time. Example 3 illustrates this goal.

Example 3

```
/*-------------------------------------------------------------*/
/* Display the name and payment details for all employees.     */
/* Employee information is stored in a DB2 database,           */
/* financial information stored in an Oracle database.         */
/* REXX/SQL multi-database example.                            */
/*-------------------------------------------------------------*/
/*-------------------------------------------------------------*/
/* Load the Oracle and DB2 external function libraries...      */
/*-------------------------------------------------------------*/
Call rxfuncadd 'LoadOracleFuncs','REXXSQL','LoadOracleFuncs'
Call LoadOracleFuncs
Call rxfuncadd 'LoadDB2Funcs','REXXSQL','LoadDB2Funcs'
Call LoadDB2Funcs
/*-------------------------------------------------------------*/
/* Connect to local Oracle database...                         */
/*-------------------------------------------------------------*/
If oracle_connect('/') < 0 Then
   Do
     Say oracle_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Connect to local DB2 database PERSONNEL..                   */
/*-------------------------------------------------------------*/
If DB2_connect('PERSONNEL') < 0 Then
   Do
     Say DB2_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Declare queries to be performed...                          */
/*-------------------------------------------------------------*/
query1 = 'select name,empno,from emp order by name'
query2 = 'select amt,paydate from gl_trans where accno = :ACCNO'
/*-------------------------------------------------------------*/
/* Execute the first query on the DB2 database and return all*/
/* rows ...                                                    */
/*-------------------------------------------------------------*/
if DB2_command(q1,query1) < 0 Then
   Do
     Say DB2_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* For each employee record, obtain the payment details from */
/* the Oracle database and display them...                     */
/*-------------------------------------------------------------*/
Do i = 1 To q1.name.0
   if oracle_command(q2,query2,'ACCNO',q1.empno.i) < 0 Then
      Do
        Say oracle_error_text()
        Exit
      End
   Say ''
   Say Right(q1.empno.i,8) Left(q1.name.i,35)
   Do j = 1 To q2.amt.0
      Say Copies(' ',15) Left(q2.paydate.j,12) Right(q2.amt.j,12)
   End
End
/*-------------------------------------------------------------*/
/* Disconnect from the Oracle database...                      */
/*-------------------------------------------------------------*/
If oracle_disconnect() < 0 Then
   Do
     Say oracle_error_text()
     Exit
   End
/*-------------------------------------------------------------*/
/* Disconnect from the DB2 database...                         */
/*-------------------------------------------------------------*/
If DB2_disconnect() < 0 Then
   Do
     Say DB2_error_text()
     Exit
   End
Return
```

ATTACHMENT A

Synopsis

sql_connect(username/password[,remote_database_string])

Description

This function connects you to an oracle database. You supply the function with the username/password and optionally the remote database connect string.

Arguments

username/password	- username/password
remote_database_string	- connect string for remote database

Return Values

0	- successful connection
Negative number	- Oracle error number

Example

To connect as your OPS$ Oracle login you would use the following call:

```
rcode = sql_connect('/')
```

To connect to the Oracle account SCOTT with TIGER as password:

```
rcode = sql_connect('scott/tiger')
```

To connect to the Oracle account SCOTT with TIGER as password on the remote host overthere; Oracle SID of X, using SQL*Net TCP/IP:

```
rcode = sql_connect('scott/tiger','T:overthere:X')
```

Synopsis

sql_disconnect()

Description

This function disconnects you from an oracle database and commits any outstanding transaction. By default, whenever the GUROO program exits, you are disconnected from the database.

Arguments

None

Return Values

0	- successful connection
Negative number	- Oracle error number

Example

```
rcode = sql_disconnect()
```

Synopsis

sql_command(statement_name,sql_command[,bind_variable_name,bind_variable_value...])

Description

This function enables you to execute any Oracle SQL*Plus command including DML and DDL statements. Typically you would execute commands like *select* or *update* using this function. Note that the command does not end in a semi-colon. If you do append a semi-colon to the end of the command, GUROO will remove it.

When the SQL command issued is a *select* statement, GUROO returns all column values in arrays. The stem variable name is composed of the statement name followed by a period followed by the column name specified in the select statement. As with all REXX arrays the number of elements in the array is stored in the variable with an index of 0. When the value of a column is NULL, an extra REXX variable is created. This variable has the same structure as the REXX variable containing the column value, but with 'NULL' before the index value. For example; the REXX variable created for a select statement containing the column 'COL_NAME' and a statement name of 'Q1' will be Q1.COL_NAME.1 (for the first row). If the value of that column is NULL, the REXX variable created is Q1.COL_NAME.NULL.1. To determine if a column is NULL, use the following test:

```
If q1.col_name.null.i = 'NULL' Then ... (column is NULL)
```

Because the contents of all columns for all rows are returned from a 'select' statement, the select command may return many rows and exhaust memory. Therefore the use of sql_command() should be restricted to queries that will return a small number of rows. For larger queries use a combination of sql_open_cursor() and multiple sql_fetch_row() calls.

When bind variables are used, a pair of arguments is used for each unique bind variable name. The first argument is the name of the bind variable as specified in the SQL statement, the second is the value that bind variable is to take.

Arguments

statement_name	- a string of up to 30 characters to identify the SQL command. This is used as the first part of the stem variable name containing column values.
sql_command	- any valid SQL*Plus command.
bind_variable	- optional bind variables values as specified in the SQL command.

Return Values

Positive number	- successful operation; the number of rows affected by the SQL command.
Negative number	- Oracle error number

Example

To select the names of all tables owned by the current user, issue the following call:

```
rcode = sql_command('Q1','select table_name from user_tables')
```

Assuming the user owns the tables EMP, DEPT, and CUSTOMER rcode will be set to 3 and the following REXX variables will be set:

```
Q1.TABLE_NAME.0 = 3
Q1.TABLE_NAME.1 = EMP
Q1.TABLE_NAME.2 = DEPT
Q1.TABLE_NAME.3 = CUSTOMER
```

To select the names of all tables and the tablespace in which they reside owned by the user, SCOTT, and use a bind variable, issue the following call:

```
query.1 = 'select table_name,tablespace_name'
query.2 = 'from all_tables where owner = :OWNER'
query = query.1 query.2
rcode = sql_command('Q1',query,'OWNER','SCOTT')
```

Assuming that user SCOTT owns the tables:

EMP	in TEMP_SPACE tablespace
DEPT	in USER_SPACE tablespace
CUSTOMER	in TEMP_SPACE tablespace
PRICE	in USER_SPACE tablespace

rcode will be set to 4 and the following REXX variables will be set:

```
Q1.TABLE_NAME.0 = 4
Q1.TABLE_NAME.1 = EMP
Q1.TABLE_NAME.2 = DEPT
Q1.TABLE_NAME.3 = CUSTOMER
Q1.TABLE_NAME.4 = PRICE

Q1.TABLESPACE_NAME.0 = 4
Q1.TABLESPACE_NAME.1 = TEMP_SPACE
Q1.TABLESPACE_NAME.2 = USER_SPACE
Q1.TABLESPACE_NAME.3 = TEMP_SPACE
Q1.TABLESPACE_NAME.4 = USER_SPACE
```

To delete rows from the EMP table where DEPTNO = 10, issue the following call:

```
rcode = sql_command('Q1','delete from emp where deptno = 10')
```

Assuming there were 5 rows in EMP for DEPTNO 10,rcode will be set to 5.

To delete rows from the EMP table where DEPTNO = 10, issue the following call:

```
rcode = sql_command('Q1','delete from emp where deptno = 10')
```

Assuming there is no table called EMP, then rcode is set to -947; the Oracle error number.

Synopsis

sql_open_cursor(statement_name,sql_command[,bind_variable_name,bind_variable_value...])

Description

This function passes a *select* statement to be parsed, prepared and executed. The rows that the SQL command retrieves are then made ready for repeated calls by sql_fetch_row(). An explicit cursor is associated with the statement name passed.

This function takes the same arguments as sql_command().

Arguments

statement_name	- a string of up to 30 characters to identify the SQL command. This is used as the first part of the stem variable name containing column values and as the argument to sql_fetch_row() so it knows from which cursor to fetch rows.
sql_command	- any valid SQL*Plus *select* command.
bind_variable	- optional bind variables values as specified in the SQL command.

Return Values

0	- successful connection
Negative number	- Oracle error number

Example

To prepare for returning the names of tables owned by the current user:

```
rcode = sql_open_cursor('Q1','select table_name from user_tables')
```

Assuming the user has select permission on the user_tables object, rcode is set to 0.

See sql_fetch_row() for further examples.

Synopsis

sql_fetch_row(statement_name)

Description

This function retrieves the next row in the previously opened cursor and sets REXX variables for each column specified in the sql_command passed to the sql_open_cursor() function.

The format of the REXX variables set is statement_name followed by a period followed by the column name. If the value of a column is NULL, an extra REXX variable is created. See the format and usage in the description for sql_command.

Arguments

statement_name	- a string of up to 30 characters to identify the SQL command. This is used as the first part of the stem variable name containing columns values and as the argument to sql_open_cursor().

Return Values

0	- successful connection
Negative number	- end of cursor

Example

To return the names of tables owned by the current user using an explicit cursor:

```
rcode = sql_open_cursor('Q1','select table_name from user_tables')
If rcode < 0 Then
Do
   Say sql_error_text()
   Exit 1
End
Do Forever
   rcode = sql_fetch_row('Q1')
   If rcode < 0 Then Leave
   Say q1.table_name
End
rcode = sql_close_cursor('Q1')
```

Assuming the user owns the tables, EMP, DEPT, and CUSTOMER the output from this code will be:

```
EMP
DEPT
CUSTOMER
```

Synopsis

sql_close_cursor(statement_name)

Description

This function closes the cursor associated with statement_name and frees up resources held by that cursor.

Arguments

statement_name	- a string of up to 30 characters used to identify which cursor is to be closed.

Return Values

0	- successful connection
Negative number	- Oracle error number

Example

To close an already opened cursor:

```
rcode = sql_close_cursor('Q1')
```

See sql_fetch_row() for a further example.

Synopsis

sql_error_text()

Description

This function returns the text of the last error encountered from the most recent GUROO external function. The error may relate to an Oracle error or to an error within GUROO itself.

If the most recent GUROO external function was successful, then the value returned is 'Last operation successful'.

Arguments

None

Return Values

Text of the result of the most recent GUROO external function.

Example

To display the text of the result of the most recent operation:

```
Say sql_error_text()
```

See sql_fetch_row() for a further example.

Using REXX in a UNIX Environment to Manage Network Operations

Lee Krystek
Boole and Babbage

1. Using REXX in a Unix Environment to Manage Network Operations:

Lee Krystek - Software Manager
Boole and Babbage Network Services

Abstract: When designing our network management and control product, we needed to provide a way for users to construct scripts to control any foreign system they might need to interface with via that foreign system's console. We selected REXX as this tool. Before we could use it, we had to augment the language to give it the capability to be started automatically, connect to those foreign systems, and manipulate our relational database.

1.1. The COMMAND/Post Product:

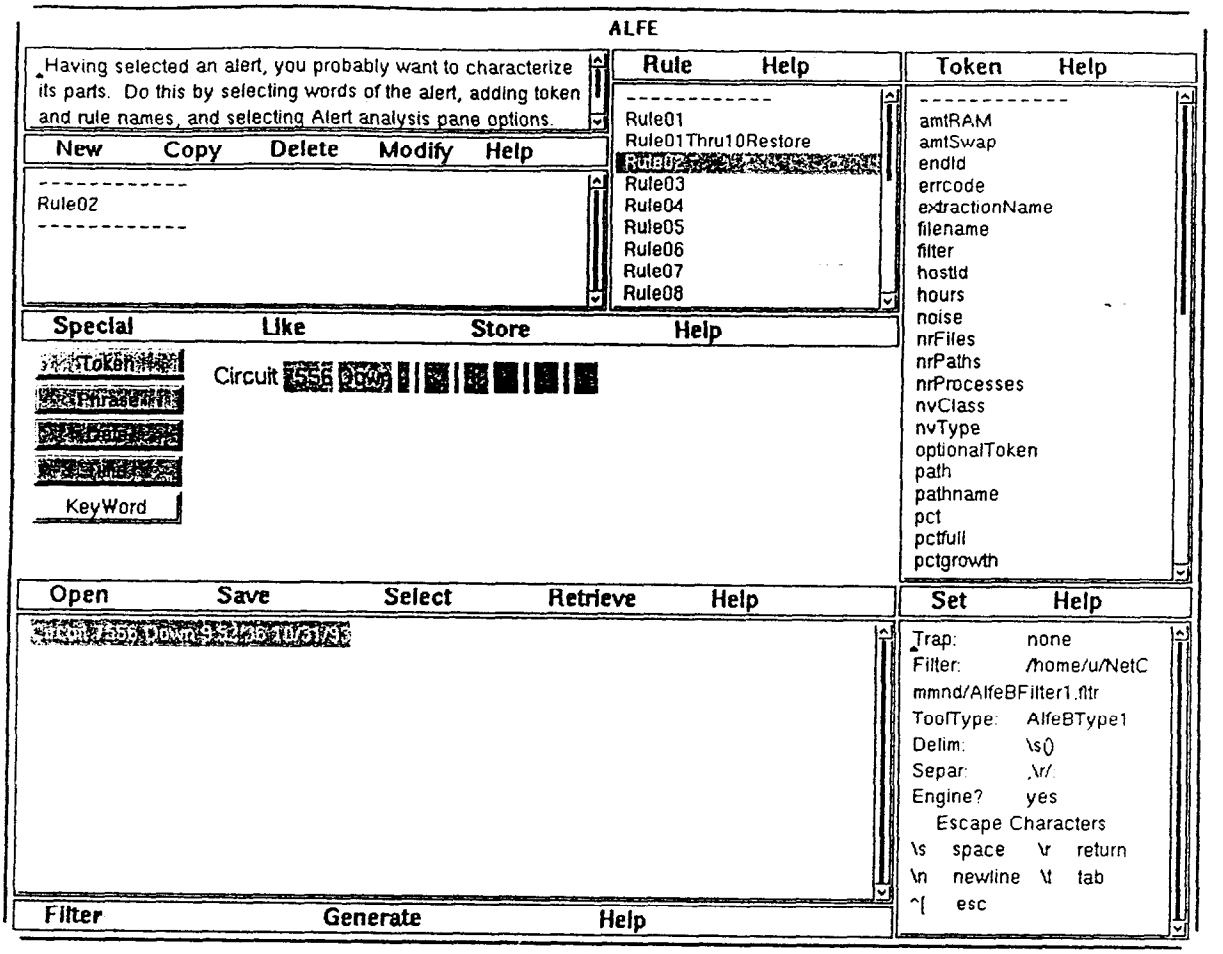
Several years ago Boole and Babbage recognized the need for a product that would be a focal point for network and systems management operations. This product would monitor and control network equipment, computer systems, and even application programs. Of special interest were non-SNA and non-SMNP systems which did not support any network management protocol.

COMMAND/Post runs on a UNIX workstations. Initially the SUN SPARC series of processors was used, however, porting to other UNIX systems is underway. A typical COMMAND/Post system consists of one or more (perhaps even as many as 50 at a large site) workstations with color monitors running a GUI such as Open Windows or Motif. The system is composed of modules written in Smalltalk (an object oriented language) for the user interface and "C" for the more intensive processing tasks. Sybase, a relational database provides the data storage.

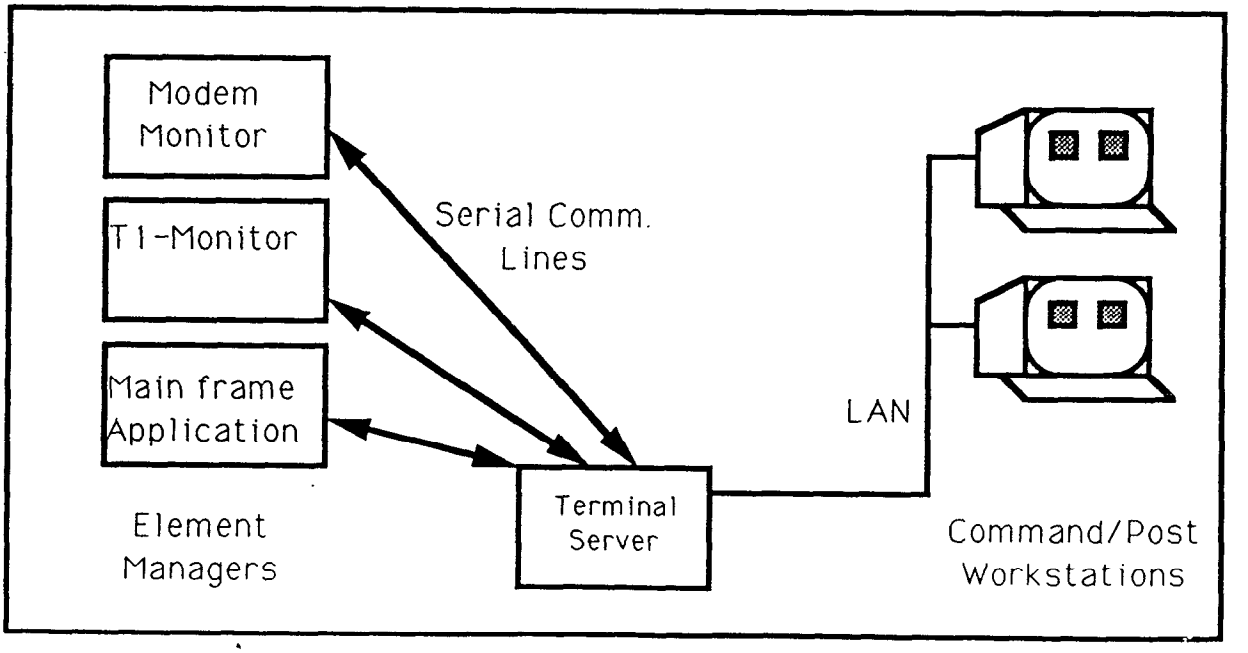
COMMAND/Post will typically connect to a network system, such as a modem monitor or T1-monitor or a computer, through the system's printer port and console. (These type of systems are typically referred to as "Element Managers" or EMs because they control one class of element in the whole network.)

An EM's printer port will often produce interesting information such as the failure of a modem or communication line. COMMAND/Post has a tool, called ALFE (ALERT LOGIC FILTER EDITOR), that implementers use through user friendly dialog screens, to construct an alert "filter." (Figure 1) The filter searches the message stream from the EM's printer port and recognizes important messages. The filter parses those messages and then creates an "alert" in the COMMAND/Post system using data obtained from the message. The filter assigns the alert a priority and an classification based on the OSI standard for network management. COMMAND/Post records the actions of supervisors and operators and tracks how the alert is handled and resolved.

COMMAND/Post operators use terminal emulations windows to access the EMs from their workstations. This allows operators to work on problems that might involve a dozen EM's without leaving their seat. (Figure 2)



An ALFE Screen - Figure 1.



Emulation Connections - Figure 2.

1.2. The Auto Operations Requirement

It became apparent after the initial release of COMMAND/Post that our prospective customers wanted to have the system support automated operations. That is, to have COMMAND/Post not only detect alerts and display them, but to also automatically take actions based on an alerts or alerts received from a single EM, or on a combination of alerts from several EMs.

COMMAND/Post already had the ability to connect with the system consoles for the various EMs. Therefore it seemed logical that if an automated operations facility could be built we could send commands to the appropriate EM, through the emulations, to get an EM to take the desired action.

The automated operations facility needed two parts. First, some kind of detection mechanism that would allow the triggering alert, or combination of alerts, to be recognized. Second, another mechanism that could have a conversion with an EM's console, as if it were a human operator, in order to enter the commands necessary to get the EM to carry out the desired action.

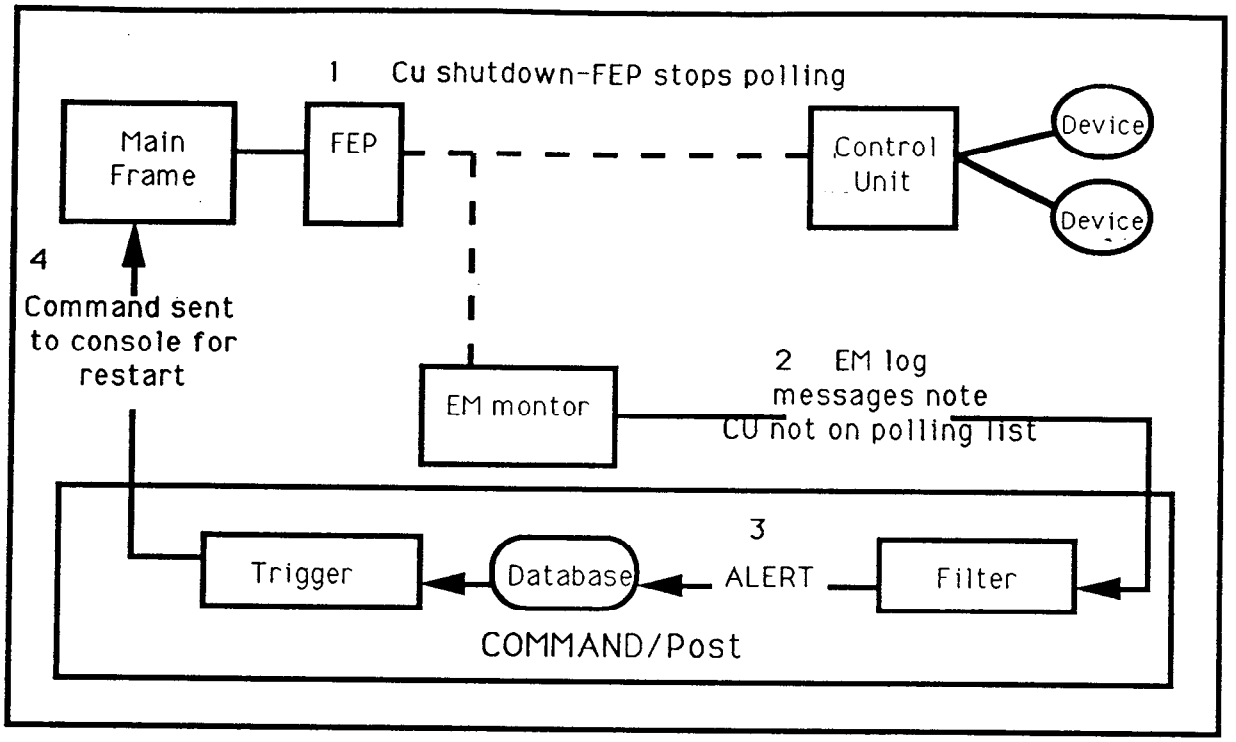
A simple example of an automated operation, though no longer a problem on most networks, is automatic restart of polling on a communication line. A Front End Processor (FEP) is polling several control units at remote sites across a single wide area network line. One of the controllers goes off line for a period of time and the FEP automatically drops that controller from the polling list. When the controller came back on line an operator would command the FEP to add that controller back into the polling list. Under COMMAND/Post automated operations, an EM monitoring that communications line would report the failure of the control unit. COMMAND/Post filters would detect this as an alert, and would then trigger an automatic operation to send a command to the FEP to add the controller back to the list. If the controller failed to respond over a specified period of time, a high-priority alert could be generated to inform the operator that a situation had occurred that could not be remedied through auto operations. (Figure 3)

The design of the alert detection and trigger mechanism took advantage of COMMAND/Post's relational database mechanisms for storing and accessing data. A graphic window display (known as a selector) already existed to select alerts. The implementer uses the selector and the mouse to click on certain rules that describe the alert(s) to be shown on an alert display window. This idea was extended to allow groups of alerts to be detected. When a specified combination of alerts is detected instead of having the alert(s) appear in a window a "trigger" would fire and the auto-operation would start. (Figure 4)

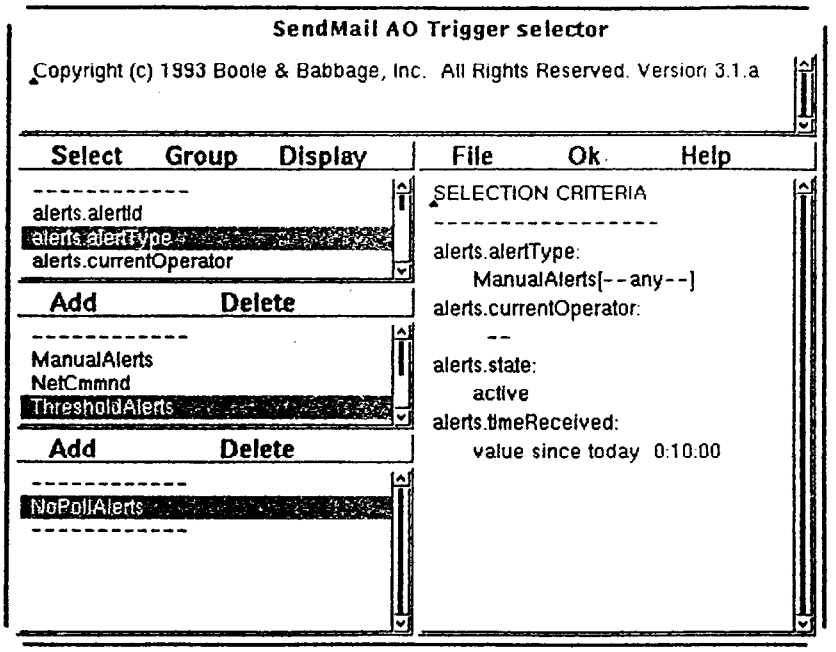
Once the detection facility was decided, the mechanism to allow the system to carry on a conversation with an EM console was next. An augmented version of REXX was chosen for that mechanism.

1.3. Why REXX?

The decision to use REXX was based on several factors. The actual REXX



Restarting a Controller by Auto-Operation - Figure 3.



A Selector - Figure 4.

product chosen was uni-REXX from the Workstation Group.

1.3.1. *REXX was already an established language for auto-operations on Boole's mainframe products.*

In addition to COMMAND/Post, Boole already had some main frame products that incorporated auto-operations. They used REXX extensively. It was decided there would be an advantage to keep the auto-operations language consistent between the products.

Using REXX also allowed us to draw upon the experience of our main frame programmers, and some of the extensions to the REXX language to support database operations were based on insights provided by the mainframe REXX group.

1.3.2. *Some of the auto-operations scripts would be written by customers and a language already familiar to IBM type main frame operators was desired.*

Although COMMAND/Post is a Unix-based product, many of the audience for it have their roots in the IBM culture where REXX is widely used. By choosing a familiar language it was hoped there would be less fear and resistance by customers to writing their own REXX scripts.

1.3.3. *REXX's ability to parse data strings would make analysis of messages coming from the EM's easier.*

It was expected that much of the function of the scripts would be to respond to messages coming from the EM systems. The REXX "parse" facility allows most of these messages to be handled without a lot of programming. The "parse" statement is usually easy for even a novice programmer to understand.

1.3.4. *REXX's ability to pass commands to underlying environments makes it easy to address COMMAND/Post's database.*

The extensions to the database were critical if we were to be able to write easy to read scripts. The ADDRESS instruction allowed us pass SQL command directly to the database. Also important was the ability of REXX to create new variables of any type "on the fly" as data was returned from the database. This eliminated the need for a rigid, complicated structure (as used in "C" when getting data back from the DB).

1.3.5. *Use of a "light," interpretive language makes debugging easier for non-professional users.*

Though interpretive languages execute more slowly than compiled languages they are often easier for the novice to debug since there is no

compilation wait involved. Also unless the compiled language has a sophisticated debugger, the source line is not displayed in association with a run-time error. In addition, REXX has a built in trace feature which is easily used.

The use of a "light" language that didn't need extensive variable declarations, etc. was also an advantage. While such languages become increasingly difficult to maintain as a single program grows larger and more structure is needed, because of the anticipated size of the scripts (500 lines or less), that was not a concern.

1.4. External Access

The first change we made to REXX was to give it the capability to connect with the EM's. This was more complicated than simply opening a new file descriptor to a new tty port. Connection to EM's for filtering and emulation are managed as resources by COMMAND/Post. A connection to a EM's system console might be used for a period of time by an operator via an emulation, and later reassigned by the system for use by auto-operation via a REXX program.

Connections were made from a program to a physical port using the UNIX socket/stream facility. The actual physical ports might be a tty, or, more likely a port on a terminal server connected remotely, via LAN or WAN, from the workstation where the REXX was actually executing. The resource management system was designed to make the details of the actual connection transparent to the connecting program. This means the REXX program need only know a single name to invoke the connection.

In order to allow the REXX to connect through COMMAND/Post's resource management system several functions were added to the language by inserting additional code into the REXX interpreter so it could use UNIX sockets and streams:

```
<fd> = ao_targetConnect(<name>)

ao_targetClose(<fd>)

ao_targetComm(<fd>,<function>,<data>,<length>,<pos>)
```

The first function, **ao_targetConnect**, requests the opening of a connection to a named port. The name implies more than simply a physical port. It also implies a pathway to get there and, in some cases, a terminal emulation appropriate to the external target system on the other side of the port. These are defined externally to REXX by COMMAND/Post's system management faciiity.

A file descriptor, or more appropriately a "handle" is returned by **ao_targetConnect** to identify the path for future communications calls.

The **ao_targetClose** function simply reverses the connect function closing down the path. The handle from **ao_targetConnect** is the single argument to **ao_targetClose**.

The third function, **ao_targetComm**, actually carries out the transfer of data between the REXX program and the target system.

1.4.1. Application Program Interface

When it came to actually talking to the target system we were faced with an additional problem. Usually the device a REXX EXEC needs to talk to is a system console. That means the program would be responding to the commands we sent it with data (including our own full duplex echo) as well as occasionally sending out, from our point of view, random lines of data as the result of activity on the system. How could we develop an interface for REXX that would allow us to send data at will and handle messages from the target when they came in at any time? Turning to an interrupt model, where we would sit in a wait state until an incoming message would trigger a designated REXX function seemed to be too complicated for easy use by most of our customers, especially when more than one target system might be involved in a single REXX program.

Instead we decided to use an Application Programming Interface (API) to interact with the target. The API we developed was similar to that defined by IBM as the "IBM PC 3270 Emulation Program, Entry Level, High-Level Language Application Program Interface" or EEHLLAPI. Where the IBM was targeted to a 3270 terminal interface, our API widens the definition to cover terminals that do not use field positioning.

The API operates much like a person sitting at the terminal console. A pseudo-screen is created (which does not display on the COMMAND/Post workstation monitor), and the REXX program uses functions defined in the API to interact with this screen. Some functions allow the entire screen to be captured as an array and transferred back into a REXX variable for processing. Other functions allow a portion of a screen to be captured, or in the case of a terminal supporting fields, a field to be captured. Other functions allow data to be sent to the screen as if was coming from the keyboard. There are a number functions dedicated to positioning the cursor and searching the screen, or fields, for text. A few give status information, including the height and width of the screen.

The API also allows for an interrupt driven capability for situations where a simpler set of calls cannot handle the exchange. The REXX program waits until new data arrives on the pseudo-screen and then is released so it can make additional calls to observe how the screen has changed.

All calls to the API interface are made through the **ao_targetComm** function described above. The "fd" argument contains the handle for the particular target system involved, and the "function" argument contains the number of the API function that will be used. The "data", "length", and "pos" arguments definitions vary based on the function call. In general, "data" is data being read or written to the pseudo screen. "Length" is the

length of that data string. And "pos" is the position involved when data is written or read. The API views the screen as an array of characters (row one, followed by row two, etc.) and the position is a value pointing to that array.

Using the API model to connect with the target system has a number of advantages. First, as noted above, it removes the need for an interrupt type interface when only simple communications are involved. When only a single thread of communication is involved it is relatively easy to create a loop in REXX to read the screen, write to it, read the screen again, identify what has changed and then act on the new data.

Another advantage is the interface allows some measure of emulation independence. That is, a REXX script can be designed that will operate with either a EM running a VT-320 interface or a IBM3151. Then the only change needed between the two would be in the definition of the pathway during the configuration step external to REXX. The user would define the path as using a VT-100 API interface instead of an IBM3151.

Despite the interface, there are some restrictions on how transparently a REXX program can be written. Some terminals support the use of "fields." A REXX program that made use of the API field related functions to interact with a Tandem 6539 would not work with a VT-100, because it does not support fields.

1.5. Parameters

The REXX interpreter was also augmented to accept command line arguments that could be passed in the the REXX programs as parameters. The command line to the left of a "--" remained the standard uni-REXX command line. The part to the right represented parameters passed to the REXX program. Argument flags (items starting with a "-") became variable names in the program filled with the values that followed them. The following command line:

```
ncrx -- -customerName "Fred"
```

would cause the REXX program to start execution with a variable called "customerName" initialized to the value of "Fred". This allowed the triggers to pass useful information to a REXX program. Standard information passed included the number of alerts that caused the trigger to fire and the identification numbers of those alerts.

1.6. Database Interface

We aiso wanted the REXX auto-operations programs to be able to access the COMMAND/Post database so they could create, query, update, and delete the alerts the system maintained.

COMMAND/Post uses a relational database that is divided over two

dataserver programs using the Sybase "open server" model. The primary database is accessed through the standard Sybase dataserver. Temporary high activity tables are assigned to the "Event Handler" server: a memory resident server of our own design.

Access to either server is via REXX's ADDRESS instruction. Addressing NCDB connects the REXX program to the primary sybase dataserver, using "ALERTS" connects it to the Event Handler. To interact with either the programmer need only code an SQL command, or use a stored procedure (a Sybase term for SQL routines maintained in the dataserver) in the address command. The success of the command can be evaluated by looking at the special REXX variable "sqlCode".

While, for the most part, addressing the dataservers via this command is straight-forward, a few SQL commands represent a problem. For example, "SELECT *" command may return row after row of data from the table, each row with many individual data items. Each item can be of a variety of data types. Here's where REXX's ability to create variables on the fly and have variables types change make it an excellent choice of our application. As a data item is returned, let say the time field for particular alert, a REXX variable named "TIME" is created, if it does already exist. It is filled with the text representation of the time. The same thing for integers or for character strings (which the database can store in several varieties) The programmer need not immediately be concerned with making sure the variable type matches what's coming back from the database.

Multiple rows are handled by returning one row at a time and having a special "fetch" command. The program can use to indicate that it is finished with the current row and is ready to receive the next. Values for the new row are written over and into the same variables used by the last row. If all rows are exhausted the "sqlCode" variable returns an error value (non-zero). If there is no need for additional pending row a special "cancel" command can be used to drop them.

A typical code fragment to print the item "alertId" from the "ActiveAlert" table might be:

```
address NCDB "select alertId from activeAlerts"
if(sqlCode = 0)then
  do forever
    address NCDB "fetch"
    if(sqlCode <> 0) then
      leave
    else
      say alertId
  end
end
```

One limitation created by this architecture is that all values returned by a "select" statement must have some associated name for creation of the variable. This means that an SQL statement that used some function (like SUM) to create a value that would not have a name associated with it must be

written in such a way that it is forced into a variable name. For example:

```
select total = sum(occurrences) from activeAlerts
```

instead of

```
select sum(occurrences) from activeAlerts
```

To make common operations, like creating an alert, (which would normally require multiple table inserts) easier, a number of stored procedure are included in the database. This means that typically only a single "address" clause is needed for even a fairly complex database operation.

1.7. Other Uses of REXX in the Product.

One of the bonuses of implementing REXX as our auto-operations language was that we could use it for general programming. We have a large library of scripts (mostly written in Bourne or C shell) used for installation and maintenance of the product. When these scripts interacted with the database they had to first create a second file that would act as input to the Sybase's Interactive SQL program (ISQL). Then they had to start ISQL directing the second file to the standard input, and finally monitor the standard output for errors. This convoluted approach made the script hard to read. It also made isolating a particular SQL statement that failed difficult since the script was not feeding the commands to ISQL one by one.

Our REXX, with the ability to address the server through the ADDRESS instruction has simplified this problem. Since the REXX can address the database directly it is easier to write and test the script/program. Errors are also easier to detect and handle.

1.8. Results

Over 100 sites now use COMMAND/Post with the automated operations facility. A majority of the customers involved have decided to write their own custom auto-operations scripts which lessens the load on our support staff. We are pleased with our decision to use REXX for auto-operations.

1.9. Bibliography

COMMAND/Post How to Guide, Release 3.0, Boole and Babbage Network Systems, San Jose California, 1993.

Programmers Guide: High Level Language Application Program Interface, IBM Corporation, Austin Texas, 1987.

A REXX CookBook for COMMAND/Post, Boole and Babbage Network Systems, Mt. Laurel New Jersey, 1993.

uni-REXX Reference Manual, The Workstation Group, Rosemont Illinois, 1991.

1.10. Glossary

ALFE - Alert Logic Filter Editor - The facility in COMMAND/Post used to construct a "filter".

API - Application Program Interface - The interface that allows REXX and C based programs to interact with COMMAND/Post emulations.

Dataserver - A process that manages and provides access to a database.

EM - Element Managers - Network Control and Monitoring systems that manage a domain of network elements like modems, communication lines, etc.

Event Handler - COMMAND/Post primary memory resident dataserver.

EXEC - A REXX program for COMMAND/Post that is part of the auto-operations subsystem.

IBM 3151 - IBM async terminal.

IBM 3270 - IBM sync terminal.

Filter - A program in COMMAND/Post which parses a stream of data, usually from some external source, looking for messages. When a message is found the filter created an alert for the COMMAND/Post database.

Open Server - A database design that allows dataservers from multiple vendors to operate together.

RDBS - Relational Database System - A database designed to adhere to relational principles.

Shell - A Unix command interpreter.

Shell Script - A program that is interpreted by a Unix C or Bourne shell.

SQL - A 3rd generation database manipulation language.

Sybase - A RDBS product.

Tandem 6539 - Tandem async terminal

Unix - Operating System on which COMMAND/Post runs.

VT-100 - VT-320 - DEC async terminals

1.11. APPENDIX: Sample COMMAND/Post REXX program.

```
#!/usr/nc/bin/ncrx -s
/*************************************************************
 * REXX program to perform simple paging.  The OSI Severity
 * of the first triggering alert will be sent to the pager.
 *
 *      This program does the following:
 *
 *       >Connects to a tool called "pagerModem" which is
 *              assumed to be an "AT" mode modem
 *       >Initializes the modem for word responses.
 *       >Sends touchtone dial sequence which leaves the modem
 *              in command mode
 *       >Sends "PIN" followed by "#"
 *       >Looks at the first underlying alert passed in and gets
 *              the OSISeverity value from the data base.
 *       >Sends OSISeverity followed by "#" out to the pager.
 *       >Waits 5 seconds for repeat
 *       >Sends final "#" to force posting of message
 *       >Does a "hangup"
 *       >Disconnects from modem.
 *
 *       The following variables should be passed in from the trigger:
 *          pagerModem - Access path of the modem.
 *          PIN - Users pin number.
 *          underlyingAlerts (optional)
 *          alertCount (option)
 *
 *       If no alertCount or underLying alerts are available the value
 *       "9999" will be sent to the pager.
 *
 *       Note: Triggering Alerts must be forwarded to the database so
 *       that the Severity can be obtained.
 *
 *************************************************************/

/*  Get the OSI severity from the database */

if alertCount <> "ALERTCOUNT" then do
  cnt = 1
  alertId = underlyingAlerts.cnt
  address NCDB "select OSISeverity from alerts where ",
  alertId" = alertId"
  if(sqlCode <> 0)then do
    say "Could not get Severity from database"
    exit
  end

  address NCDB "fetch"
  message = OSISeverity
end
else
```

```
      message = "9999"

/*  Connect to the modem */

nl    = "0a"X

path = ao_targetConnect(pagerModem)
if rc \= AON_noError then do
    say "failed to make connection to" pager
    exit
end

/* Get the dimensions of the emulation screen and
   calculate the Presentation Space size */

string = "A"
r = ao_targetComm(path, 22, string, 1, 0)
columns = delstr(delstr(string, 16), 0, 13)
rows = delstr(delstr(string, 14), 0, 11)
PSsize = columns * rows

/* Start the conversation with the modem */

if modemSend(path, "ATV1"nl,"OK") = 1 then
    exit

/* Dial the service */

if modemSend(path, "ATDT9,18007597243@;"nl, "CONNECT") = 1 then
    exit

/* Send the PIN */

if modemSend(path, "ATDT"PIN"#;"nl, "OK") = 1 then
    exit

/* Send the message (OSI Severity) */

if modemSend(path, "ATDT"message"#;"nl, "OK") = 1 then
    exit

address UNIX "sleep 5"

/* Clean up */

if modemSend(path, "ATDT#;nl", "OK") = 1 then
    exit

if modemSend(path, "ATH"nl, "OK") = 1 then
    exit

/* Disconnect from the path */

r = ao_targetClose(path)
```

```
exit

/*********************************************************************
 * modemSend
 *
 *   sends the contents of string to modem and checks that the
 *   response from the modem contains the contents of pattern
 *
 *********************************************************************/

modemSend:
   arg path, string, pattern

   len = length(string)

   /*  Find the cursors current location on the screen */

   r = ao_targetComm(path, 7, 0, cursor, 0)
   if rc \= AON_noError then do
    say "Can't find cursor: rc =" rc
    return 1
   end

   /*  Send the command to the modem */

   r = ao_targetComm(path, 15, string, len, 0)
   if rc \= AON_noError then do
    say "Send failed: rc = "rc
    return 1
   end

   /* Loop ten times waiting each time 2 seconds for a response. */

   do 10
    address UNIX "sleep 2"

    /* Look for response following cursor position */

    r = ao_targetComm(path, 8, string, (PSsize - cursor), cursor)
    if rc \= AON_noError then do
        say "Send failed on check: rc =" rc
        return 1
    end

    /* Search for a the expected pattern */

    if pos(pattern, string) \= 0 then do
        return 0
    end
   end

   say "did not find" pattern
   return 1
```

REXX at Simware

Luc Lafrance
Simware

REXX at Simware

by

Luc Lafrance
Software Developer

Multiple Scripting Uses

- 2 products
  A2B, REXXWARE
- 3 platforms
  Windows, Macintosh, NetWare
- presentation, automation, reports

A2B BIFs

- terminal emulation
  - type, press, waitfor, checkfor, whenever
- views (presentation)
  - openview, closeview
  - addelement, additem
  - getelement, getitem
  - setelement, selectelement
  - readelement, writeelement

REXXWARE BIFs

- 140 calls into the CLIB interface
  GetObjectID, GetVolumeInformation,
  LoginObject, NWQAttachServer
- RCCs and client job scheduler
- utility functions
  currentpath, listfile, readscreen

History of Development

- 18 months to develop a 4.0 kernel
- grammar defined with LEX/YACC
- written in 'C'
  Microsoft C, MPW C, WATCOM C/386

Multiple Personalities

- ported to the Macintosh in days
- ported to NetWare in weeks
- abstraction of OS interfaces to file system and memory

Adapting REXX to an O/S

- Windows and Macintosh
  hard issues to ADDRESS
- NetWare
  a console and a prompt
- multi-threading
  the infernal data queue

The Way to an Easy Port

- SIMWARE.h

```
typedef signed short      SHORT
typedef SHORT FAR         *gpSHORT
```

- SIMSTR.h

```
#define strupr(a,b)       AnsiUpper(a,b)
#define memcpy(a,b,c)     _fmemcpy(a,b,c)
```

The Way to an Easy Port

- SIMFILE.c

  fhOsOpen, fhOsClose

  lOsRead, lOsWrite, lOsSeek...

- SIMMEM.c

  OsGPtrAlloc, OsGPtrFree

- #ifdef Hungarian_notation

YACC... What Is This?

```
call :      RX_CALL symbol opt_blank parm_list
            {   if (fInterpret == TRUE)
                    GenExternal(P_CL_OUT_SYM, $2, $4);
                else if (wLabel = wIsInternal($2))
                    GenInternal(P_CL_INTERNAL, --wLabel, $4);
                else if (wRoutine = wIsBuiltIn($2))
                    GenBuiltIn(P_CL_BUILTIN, wRoutine, $4);
                else
                    GenExternal(P_CL_EXTERNAL, $2, $4);
                OsUPtrFree($2);
            }
```

- output is portable 'C' code

Simware Conventions

- pointers in REXX

  passing a symbol as a literal

```
call GetObjectID "Luc", "OT_USER", "objectID"
```

- arrays in REXX

  passing the name of a stem

```
call listcreen "array."      /* array.0 holds count */
```

REXX at Simware

- commitment to REXX
  - substantial development investment
  - ANSI committee nomination
  - big push at Novell
- future directions
  - integration of products / third parties

REXX Resources on the Internet

Linda Littleton
Pennsylvania State University

REXX Resources on the Internet

Linda Littleton

lrl@psu.edu

Center for Academic Computing
Pennsylvania State University
214 Computer Building
University Park, PA 16802

REXX Symposium - Boston
May, 1994

Rexx Resources on the Internet

- Discussions via Listserv
- Programs available via Listserv
- Info available via FTP
- Info available via Gopher

Listserv lists

List Name	List Location	Discusses
REXXLIST	uga.uga.edu	General Rexx discussion
	vm.gmd.de	
	nic.surfnet.nl	
	vm.ucs.ualberta.ca	
	ucf1vm.cc.ucf.edu	
PC-REXX	bitnic.educom.edu	Personal REXX
REXXCOMP	bitnic.educom.edu	REXX Compiler
TSO-REXX	bitnic.educom.edu	TSO REXX
VM-REXX	vm.marist.edu	VM/CMS Rexx
ANSIREXX	psuvm.psu.edu	ANSI Documents

How to subscribe to a Listserv list

- Send mail to listserv@list-location
- The Subject can be anything or can be omitted.
- The body of the mail should be:

  SUBSCRIBE list-name your-full-name

  Example:

  SUBSCRIBE rexxlist Linda Littleton

Notes: Be sure to send the mail to LISTSERV, not to the list.

How to Send a Message

- **Write to list-name@list-location**
- **Whatever you write is forwarded to everyone on the list.**

Other Useful Listserv commands

SIGNOFF list-name
"unsubscribe" from the list

REVIEW list-name
get a list of people subscribed

LIST GLOBAL
get a list of all Listserv lists

HELP

Rexx programs available via Listserv

Library	Location
rexxlib	**psuvm.psu.edu**
psutools	**psuvm.psu.edu**
vm-util	**vm.gmd.de** **ubvm.cc.buffalo.edu** **vm.marist.edu**

How to get programs from a Listserv

Send commands, one per line, in the body of the mail to:
LISTSERV@location

To get a list of packages:
GET <library> FILELIST

To get the files in a particular package:
GET <package-name> PACKAGE

To get a list of files in a package:
GET <package-name> $PACKAGE

To get a specific file:
GET <filename> <filetype>

To Submit a Program to Rexxlib

Send it to rexxlib@psuvm.psu.edu

- **Source code only**
- **No fees**
- **Must be well documented & readable**

Rexx Info via FTP

Locations to try:

rexx.uwaterloo.ca -- /pub
- general Rexx info
- Rexx FAQ
- Free interpreters for Unix and DOS

flipper.pvv.unit.no -- /pub/rexx
- Regina code
- Archive of comp.lang.rexx

ftp-os2.cdrom.com

ftp.luth.se -- /pub/os2
- OS/2 archives

wuarchive.wustl.edu -- /pub/aminet
- Amiga archive

How to get files via FTP

- **FTP rexx.uwaterloo.ca**
- **You will be asked to identify yourself**
  Type: anonymous
- **You will be asked for your password**
  Type: your-userid@your-domain
- **Useful commands:**

dir	list directory
cd	change directory
get	copy a file to your machine
help	get help
quit	exit FTP

Rexx Info on Gopher

Gopher site bigblue.pvv.unit.no

- Rexx FAQ
- Comp.lang.rexx archives
- CMS Rexx manual
- Documentation for Rexx/imc and Regina
- Papers on Rexx
- ANSI committee documents

Using REXX and Notrix for Lotus Notes Data Manipulation

Alan P. Matthews
Percussion Software

Using REXX and Notrix for Lotus Notes Data Manipulation

Alan P. Matthews
Technical Director, Percussion Software
5th Annual REXX Symposium

May 2nd, 1994

Agenda

- Lotus Notes
  - A review
  - Issues
- Notrix
  - A Core Technology
- Notrix Composer
  - A Notrix Application
- Percussion Software, Inc.

Agenda

- **Lotus Notes**
  - **Client components**
    - Workspace, Databases, Replication
  - **Typical Applications**
  - **Server Components**
    - Architecture, Network, Mail, Administration
  - **Good & Bad !**
  - **Issues**

User Interface Components

- Workspace Desktop
  - Folders
- Database Icons
  - Mail
  - Name & Address

Databases

- Views
  - Main & Response Documents
- Forms
  - Fields & Formulas
- Searching
  - Full-text, linear
- Macros
- Importing & Exporting

Replication

- Replica ID
- Selective Replication
- Architecture
  - Hub & Spoke
  - Serial
- Conflicts

Typical Notes Applications

- Discussion databases
- Reference databases
- Workflow applications
  - Centralized
  - Peripheral
- Broadcast applications
- Mail

Lotus Notes Database Architecture

- Document Architecture
- Document ("Note") Types
  - Data notes
  - Filter notes
  - Form notes
  - View notes
  - Icon, Policy, Help, ACL etc

Mail

- Workstation or Server
- MAIL.BOX
- Name & Address Book
  - People & Groups
  - Program Documents
  - Connection Documents
  - Server Documents

Server Components

- Database
- Login
- Replication
- Mail router
- Add-in processes

Network

- SPX
- Vines
- NetBios
- TCP/IP
- Local

Lotus Notes Security

- Access Control List
- RSA Encryption
- Authentication

Lotus Notes Administration

- Certificates
- Domains
- Mail Gateways
- Monitoring
  - Logs, Statistics
- International

Inside Track

- Good things
  - Platforms (Windows, OS/2, Mac, UNIX)
  - Replication
  - Authentication
- Bad things
  - Relational Capabilities
    - No unique fields, No relational link
  - Macro Language
  - API - Complexity

Product Issues (Front End)

- Data Importing & Exporting
- No Notes Programming Model
- Version Control

Product Issues (API)

- Requires 'C' Programming
- Contains over 200 calls
- Inconsistent interface
- Specialist skills - $$$$$$$

Enterprise Issues

- Unstructured data
  - Non-relational model
  - Response Documents
  - Non-standard field types
- Workflow Integration
- N & A Synchronization
- Production Control
- Infrastructure Planning
  - Certificate types

Integration Issues

- Notes is good at:
  - Anecdotal information storing
  - Reference information
  - Customer service
- Notes is *not* good at:
  - Transactional data
  - Structured data
  - Relational data

Notrix - Agenda

- Why does it exist ?
- What is it ?
  - Overview
  - Functions

Notrix - Why does it exist ?

- Customer demands for data manipulation
- Scheduling of jobs
- Centralization
- Complexity of the API
- Triggers & Event Notification

Notrix - Overview

- The Notrix Database
  - REXX Program Document
  - Program Log
- Other Notrix Features
  - Event Manager
  - Trigger Mechanism
- REXX Programming
  - Standard REXX Language
- Present Environment
  - OS/2

Notrix - Interface Variables

- note_stem.
  - server, database, noteIDs
- field_stem.
  - name, value, type, flags

Notrix - Functions 1

- FindView()
- FindNote()
- ReadNote()
- WriteNote()
- DeleteNote()

Notrix - Functions 2

- NotesFieldList()
- NotesEvaluateFormula()
- File Attachment Support()
- NotesSetStatus()

Notrix - Internals

- Notrix reads program document
- Registers functions
- Calls RexxStart() to execute
- Uses System Exits for I/O

Notrix - Extensions

- Many existing libraries of REXX functions
  - Communications
  - Monitoring
  - Report Writers
  - User Interface Systems

Agenda

- Notrix Composer
  - Overview
  - Functions
  - Forms
  - Internals

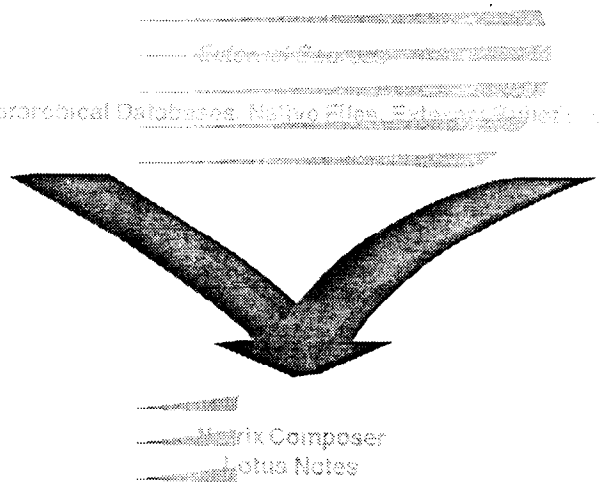
Notrix Composer

- Notrix Application
- Notes Import/Export Manager
- Accesses over 50 datasources
- Utilizes Standard Notes Form- requiring no programming

Notrix Composer

- Composer Program
- Runs on server
- Uses Event Manager

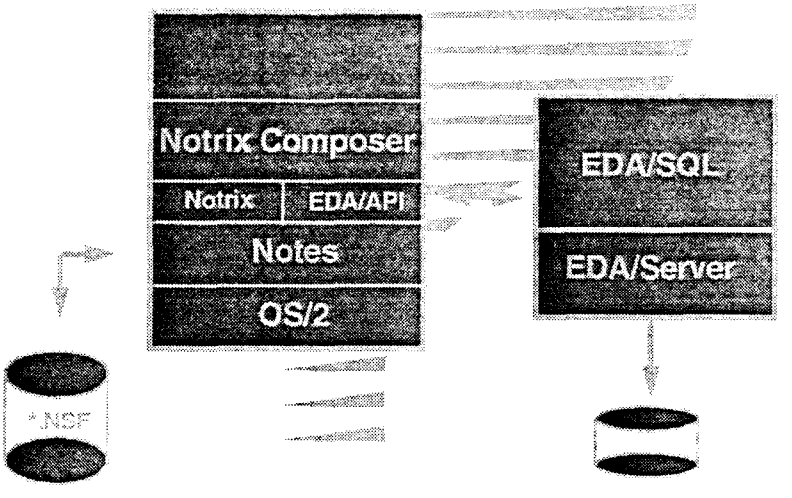
Importing External Data Sources to Notes



Notrix Composer Functions

- EdaOpen()
- EdaExecSQL()
- EDAExecRPC()
- EdaFetchData()
- EdaFieldList()
- TransferEDA2Notes()

Composer Data Flow



Notrix Composer User Interface

- Complete access through Notes Interface
- Forms-based definition of Notes and SQL
- No need for users to be aware of Notrix
- Notrix handles requests in background

Notrix Composer Job Form

- Specify Source and Target DBs
- Catalog Retrieval
- Column/Field Definitions
- Record Selection
- Calculated Fields
- Scheduling

Notrix Composer - Features and Benefits

- Scheduled Data Importing
  - Production Control
  - Change Control
- Single Interface for Data Import
- No Programming Required

Current Development

- Notrix
  - Available on OS/2
  - Coming : Novell NLM, Microsoft NT, UNIX variants
- Notrix Composer
  - Coming: Bi-directional SQL

Percussion Software

- **Focus on Software Development to help customers access, integrate, and manipulate information**
- **Lotus Notes Development and Business Partner**
- **Headquarters: Boston, Massachusetts**
  - Audrey Augun [617] 267-6700

Notrix

PRODUCT NAME: Notrix Version 1.0

Notrix is a programming tool that lets you do complex manipulation of Lotus Notes data WITHOUT Lotus Notes API or 'C' programming. Lotus Notes database administrators or designers, can quickly develop applications that easily manipulate Lotus Notes database documents and their fields. Within Lotus Notes, you can compose a REXX program document, store it in a Notes database, and schedule it to run automatically on the Lotus Notes server via the Lotus Notes Name and Address Book. The program accesses Lotus documents by searching views and manipulates fields by reading/writing documents in a Lotus Notes database. Notrix runs on the Lotus Notes server and uses the Lotus Notes front-end.

Notrix implements REXX, IBM's powerful command processing language that possesses a rich set of built-in functions. REXX is supplied with OS/2 and its advantages include readability, available source code, and easy source-level debugging. Notrix extends the REXX language to work within Lotus Notes, also adding functions that make it simple to manipulate Lotus Notes databases.

KEY FEATURES/BENEFITS:

o Notrix is completely 'Notes-centric' and takes advantage of Lotus Notes replication and security features; giving you a distributed code base and a secure development environment.

o Notrix does not require knowledge of the Lotus Notes API or complex "C" programming so that the project development cycle is reduced by 80%, saving hundreds of hours and thousands of dollars in outside consulting and technical support time.

o Notrix eases the implementation of large Lotus Notes data manipulation projects since it utilizes standard REXX Dynamic Link Libraries (DLLs) to minimize programming time and provides sample programs to enhance ease-of-use and supply instant productivity.

o With the Notrix Event Manager, you can build Notrix applications that automatically trigger when documents in Lotus Notes databases are opened, updated, or deleted. Also, a job log can provide an audit trail for system management tracking purposes.

o Notrix is server-based and no additional software is required on the Lotus Notes client so that any Lotus Notes client (Windows, OS/2, Macintosh, etc.) can be used -- on a LAN or dial-up.

o Notrix includes a Notrix Discussion Database for users who wish to exchange information with Percussion Software around topics such as feature wishlist items, bug reporting, and application examples built with Notrix. To receive updates to this database, a user needs only to replicate with Percussion's Lotus Notes server.

o An Online Help Database is included with each copy of Notrix to streamline the development process.

SOFTWARE REQUIREMENTS AND PACKAGING:

Notrix requires OS/2 Version 2.1 or later, OS/2 REXX, Lotus Notes Version 3.0 or later.

Notrix is supplied on 1 3.5-inch PC-formatted diskette containing the Notrix database (Notrix.NSF), DLLs and the Help Database. A user manual describing Notrix use and the installation process is also included.

PRICING AND AVAILABILITY:

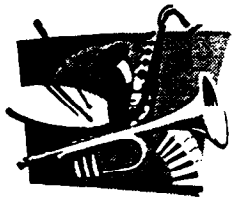
Notrix is priced at $3,500 and will be available in May, 1994.

Percussion Software, headquartered in Boston, Massachusetts, develops software products to help customers access, integrate, and manipulate information they need in their day-to-day business operations. Percussion is both a Lotus Notes Business Partner and a member of the Lotus Professional Developer's Program. Percussion offers programs for Lotus Notes VARs and consultants who wish to use our products in their solutions for a price/performance advantage.

FOR MORE INFORMATION, PLEASE CALL:

Percussion Software
222 Berkeley Street, Suite 1620
Boston, Massachusetts 02116

Phone: (617) 267-6700
Fax: (617) 266-2810



Notrix Composer for EDA/SQL

PRODUCT NAME: Notrix Composer for EDA/SQL Version 1.0

OVERVIEW:

Notrix Composer for EDA/SQL is a non-programming tool that lets you define bulk data movement between enterprise databases and Lotus Notes. Using a standard Lotus Notes form, you pull data from external sources into Notes by specifying the source database and target Lotus Notes database. You can map fields, select records, calculate new fields and determine job frequency, all without programming. Also provided is a Lotus Notes database that acts as a repository and log for all Notrix Composer activities.

Notrix Composer consists of a set of program libraries and a Lotus Notes database that lets you call functions that interface with the Information Builder's EDA/SQL Server. The Lotus Notes database provided with Notrix/Composer contains a server program that runs on the Lotus Notes server and fulfills requests for EDA/SQL data. Requests are issued by clicking on one of the buttons of the supplied Lotus Notes Form. Buttons are provided to Catalog Jobs, Schedule Jobs for later execution, and Run Jobs for immediate processing.

KEY FEATURES/BENEFITS:

o Notrix Composer lets you bring information from external data sources into Notes with no programming. You can now access information from your company's databases and bring them right into your Lotus Notes documents through an easy-to-use Notes Forms Interface that is supplied with Notrix Composer.

o Notrix/Composer contains a Lotus Notes Forms Interface that lets you specify the tables to access within the enterprise database, the Notes server and the target database name. You can also design how the original fields map to Lotus Notes fields and apply selection criteria to extract only the data you want.

o Notrix Composer works with Information Builder's EDA/SQL Server to provide access to over 50 different relational, hierarchical and native file systems. Data throughout your corporate information systems, such as customer profiles, financial results or marketing information is now available for your Lotus Notes users and can be distributed across the enterprise using the facilities of Notes database replication.

o Familiar Notes facilities are used throughout Notrix Composer and all Notrix/Composer functions are integrated with the Lotus Notes environment. You can enter your information request into a Lotus Notes Form and defer processing to a schedule of your choosing.

o Notrix Composer is server-based and no additional software is required on

the Lotus Notes client so that any Lotus Notes client (Windows, OS/2, Macintosh, etc.) can be used -- on a LAN or dial-up.

o Notrix Composer runs on OS/2 today; future releases will add NT, NLM and UNIX platforms for cross platform coverage.

o Notrix Composer includes a Notrix Composer Discussion Database for users who wish to exchange information with Percussion Software around topics such as feature wishlist items, bug reporting, and application examples built with Notrix Composer. To receive updates to this database, a user needs only to replicate with Percussion's Lotus Notes server.

o An Online Help Database is included with each copy of Notrix Composer to streamline the development process.

SOFTWARE REQUIREMENTS AND PACKAGING:

Software Requirements: Notrix Composer requires OS/2 Version 2.1 or later, OS/2 REXX, Lotus Notes Version 3.0 or later.

Notrix Composer is supplied on 1 3.5-inch PC-formatted diskette containing the EDA job definition database (NXEDA.NSF), DLLs, and the Notrix-EDA/Link interface library.

PRICING AND AVAILABILITY

Notrix Composer is priced at $5,000 and will be available in May, 1994.

Percussion Software, headquartered in Boston, Massachusetts, develops software products to help customers access, integrate, and manipulate information they need in their day-to-day business operations. Percussion is both a Lotus Notes Business Partner and a member of the Lotus Professional Developer's Program. Percussion offers programs for Lotus Notes VARs and consultants who wish to use our products in their solutions for a price/performance advantage.

FOR MORE INFORMATION, PLEASE CALL:

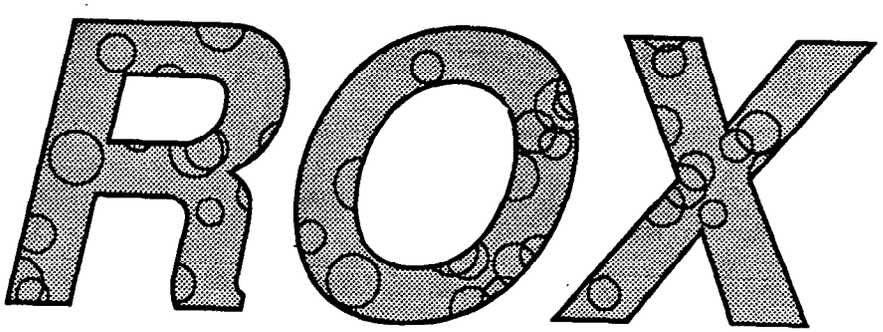
Percussion Software
222 Berkeley Street, Suite 1620
Boston, Massachusetts 02116

Phone: (617) 267-6700
Fax: (617) 266-2810

Adventures in Object-Oriented Programming in REXX

Patrick J. Mueller
IBM

Adventures in Object Oriented Programming with



(REXX Object eXtensions)

Patrick J. Mueller

pmuellr@vnet.ibm.com

May 1994, for the 1994 REXX Symposium

Trademarks

- IBM is a trademark of International Business Machines Corporation.

- OS/2 is a trademark of International Business Machines Corporation.

What is ROX?

- What ROX is:
  - A REXX function package for OS/2
  - Provides object oriented capabilities for REXX
  - An experiment
- What ROX isn't:
  - An interface to existing OO systems (C++, Smalltalk, SOM)
  - A new language
  - An IBM product

ROX object model

- Classes define:
  - Methods, implemented in REXX
  - Variables, accessible to methods
- Class inheritance
  - Classes obtain methods and variables of inherited classes
  - Multiple inheritance
- Modelled on Smalltalk, but:
  - Classes not 1st class objects
  - No garbage collection

Example ROX class

```
:*--------- animal class ----------
:class animal
:vars name sound

:method init
 name  = arg(1); sound = arg(2)

:method name
 return name

:method sound
 return sound

:*--------- dog class ----------
:class dog
:inherits animal

:method init
 name = arg(1)
 rc = animal.init(self,name,"Bark")
```

Example ROX usage

```
/* sample.cmd */

/* load the ROX file animal.rox */
rc = RoxLoad("animal.rox")

/* create a dog named Jackson */
dog = RoxCreate("dog","Jackson")

/* -> 'Jackson says Bark' */
say .name(dog) "says" .sound(dog)

/* destroy dog */
rc = RoxDestroy(dog)
```

Extra goodies

- C programming interface allowing methods to be implemented in C
- Auto-loaded DLLs to allow complete class definitions to be implemented in C
- Multithreaded support
- Execution profiling

Object creation/destruction

- Objects created with RoxCreate()
  - arg(1) is the class name
  - arg(2) ... are initialization parameters
  - The 'init' method of the class invoked automatically, if present
  - Initialization parameters passed to init method
- Objects destroyed with RoxDestroy()
  - The 'deinit' method of the class invoked automatically, if present

Object references

- RoxCreate() returns a string that is a reference to an object
- Object reference passed as first parameter to all methods, and RoxDestroy()
- Object references are plain old REXX strings - can be kept in a blank delimited string as in:

```
objs = ""
do i = 1 to 10
 objs = objs RoxCreate("dog")
end
```

- Special variables 'self' and 'super' available to methods which represent the receiver of the method

Sending messages

- Message sends are just REXX function invocations
- Object reference is always the first parameter
- Function name is method name, prefixed by "."
- Object and method name used to resolve the class that implements the method

  The two move methods invoked below are probably implemented in different classes:

```
xx = .add(aNumber,100)
xx = .add(aList,aListItem)
```

Instance variables

- Objects have as their instance variables all variables defined by their class, and its inherited classes.

- All instance variables apply only to a particular object - they are not shared between objects.

- All instance variables are 'exposed' when a method is invoked.

- Per-instance variables may be created with RoxAddVar(). This provides support for stemmed variables.

Packaging ROX classes

- RoxLoad utility allows classes to be packaged into their own files
- Multiple classes may be in one file
- Format is:

```
:include <a ROX file>

:class <class name>

:inherits <class name> ...

:vars <variable name> ...

:method <method name>
 <method code>

:method <method name>
 <method code>
```

Class-related functions

- RoxAddClass()
  create a class

- RoxClassAddInherit()
  add an inherited class to a class definition

- RoxClassAddMethod()
  add a method to a class definition

- RoxClassAddMethodDll()
  add a method (in a DLL) to a class definition

- RoxClassAddVar()
  add an instance variable to a class definition

Object-related functions

- RoxCreate()
  creates a new object
- RoxDestroy()
  destroys an object
- RoxSend()
  send a message to an object
- RoxSendThread()
  send a message to an object
  on another thread
- RoxClass()
  returns class of object
- RoxAddVar()
  add a per-instance variable
  to an object - used for stems

Utilities provided

- RoxLoad.cmd

  Calls the 'builtin' ROX functions to load a 'ROX' format file

- RoxInfo.cmd

  Prints class information for a given ROX file

- RoxProf.cmd

  Collects and analyzes output generated from RoxStats() function to generate timing information

Classes provided

- list.rox
- wordlist.rox
- set.rox
- collect.rox
  various collection classes;
  collect.rox is an abstract class

- sessions.rox
  illustrates multiple inheritance

- spinner.rox
  sample threaded class that displays
  an in-process spinner for activity

- cmdline.rox
  implements a function to read a line
  from input with history, editing, etc

- socket.rox
  usability enhancements for the
  rxSock function package

Problem areas

- Performance

  0.05-second overhead for message sends on 25/50 Mz 486 machine.

  That's pretty good, but still only 20 messages / second.

- File i/o

  Each invocation of a method opens a new file handle for a named file. Unpredictable because of buffering.

  Example: file 'a.file' opened twice

```
:method foo
 rc = lineout("a.file","x 1")

x = .foo(something)
x = .foo(something)
```

Implementation notes

- Uses REXX external function interface for message sends
- Internally, uses
  - RexxStart()
  - variable pool
  - init/term System exits
- Can be used by any REXX-macro-aware program
- Possible conflicts with programs that usurp REXX external function exit and depend on period prefixed functions

What's ROX good for?

- Experimenting with OO and REXX
- Whet your appetite for Object REXX
- A way to reuse large-ish chunks of REXX code, with shared variables

Availability

- Currently at version 1.8
- Available via:
  - anonymous ftp to ftp.cdrom.com in /pub/os2/program/rexx as rox.zip
  - Peter Norloff's OS/2 BBS

Availability

- Currently at version 1.8
- Available via:
  - anonymous ftp to ftp.cdrom.com in /pub/os2/program/rexx as rox.zip
  - Peter Norloff's OS/2 BBS

ROX - REXX Object eXtensions

Patrick Mueller
IBM Software Solutions Division
Cary, North Carolina
pmuellr@vnet.ibm.com

April 27, 1994

Contents

1 Introduction

1.1 What is ROX?

ROX is a function package for REXX that allows for object oriented (OO) programming in REXX. You should have some basic familiarity with OO programming before diving into **ROX**.

ROX allows classes to be defined. The classes have a number of features.

- they may inherit from other classes
- they specify variables that will be maintained for each object created of the given class
- they specify methods written as REXX code

Classes are defined in files with an extension of .rox. See *Format of .rox Files* on page 10 for the format of the .rox files.

1.2 What ROX isn't

ROX is not a new language - it is simply a function package that can be used from the OS/2 [1] REXX language providing some OO capabilities.

ROX provides NO facilities for interacting with other object oriented systems such as SOM or Smalltalk.

ROX has no distributed (cross-process, or cross-platform) capabilities.

1.3 Object creation

Objects are created and destroyed with the RoxCreate() and RoxDestroy() functions, described in *Object Lifecycle Functions* on page 9. The RoxCreate() function takes the name of the class to create the object from, and any number of additional parameters to initialize the object. The RoxCreate() function returns an object reference. This object reference is a regular REXX string, with a particular value which the **ROX** functions can use to dereference the object. This object reference is used as the first parameter for method invocation.

When an object is created, the *init* method for the class is invoked. Likewise, when an object is destroyed, the *deinit* method for the class is invoked. If the

[1] OS/2 is a trademark of International Business Machines Corporation.

init or *deinit* methods are not defined in the class, they will be searched for in inherited classes.

1.4 Method invocation

Once an object is created, you can send messages to it. This is also commonly referred to as invoking a method. The message is the name of the method, along with parameters that the method should be passed. To invoke a method, use REXX function call invocation. The name of the function is the name of the method, prefixed by ".". The first parameter to the function is an object reference, and any other method specific parameters can be passed as well.

It's time for a short example. In this example, we create an object of class *dog*, passing an additional parameter on the RoxCreate() function which is the name of the dog. The *init* method of the dog class will be invoked, passing the name as the first parameter. Next, the *bark* method of the dog class is invoked, in both function invocation formats available in REXX. Both invocations do the same thing.

```
jackson = RoxCreate("dog","Jackson")
call .bark jackson
g = .bark(jackson)
```

As noted before, during object creation, the *init* message is sent to the object. In order to allow an object's inherited classes to initialize themselves, the init and deinit methods may be invoked as functions whose names are a class name and the method name, concatenated together, with a "." in between them. For example, assuming the dog class inherits from the animal class, the dog init method can call the animal init method by invoking the function *animal.init*.

1.5 Variables

Classes specify both the methods that can be used on an object and the state variables associated with the object. The variables are plain old REXX variables, whose values are available to methods of the classes. The variables are non-stem variables, such as *name*, *size*, etc,. Stem variables are handled via per-instance variables (see below). Any number of variables may be associated with a class (and thus an object).

Per-instance variables are variables that can be added to an object in an ad hoc manner. For instance, one object of class X might have object variables *x.0*, *x.1*, *x.2*, where another object of class might have object variables *x.0*, *x.1*.

Per-instance variables are added to an object with the function RoxAddVar(). Per-instance variables are the only way to store stem variables with an object - stem variables can NOT be defined with a class.

When a method is invoked, the variables of the object will be available to the REXX code of the method. If the value of a variable changes in the method, the changed value will be saved with the object.

It's time for another example. In this example, we'll describe a simple class in the format acceptable for .rox files. The class is dog, and it has two variables - name and breed. They will be used to hold the name of the dog, and the dog's breed. We also define three methods - name, breed and describe. The name and breed functions either set or return the current value of the variable, depending on whether any parameters are passed to them. The describe method prints a line describing the dog.

```
:class dog
:vars name breed
:method name
   if (arg() = 1) then
      name = arg(1)
   return name
:method breed
   if (arg() = 1) then
      breed = arg(1)
   return breed
:method describe
   say "The dog's name is" name".  It is a" breed"."
   return ""
```

Below is some REXX code that uses the class dog. The result of the method describe invocation is that the line "The dog's name is Jackson. It is a Chocolate Labrador Retriever." will be printed on the screen.

```
Jackson = RoxCreate("dog")
x = .name(Jackson,"Jackson")
x = .breed(Jackson,"Chocolate Labrador Retriever")
x = .describe(Jackson)
```

1.6 Class Inheritance

Classes can inherit other classes in their definitions. This technique expands the variables and methods available to the class to the set of variables and

methods defined in any inherited classes. A class can inherit from more than one class. **ROX** has no scoping facility, so if classes are inherited that have the same method, the method will be available in the derived class (the one that inherits the other classes), but the actual method invoked is undefined. One of the methods will be invoked, but it's not possible to determine which one.

1.7 self and super

Two special variables are available to all methods. They are *self* and *super*. self refers to the receiver of the method (the object which the methods was invoked on). super also refers to the receiver of the method, however, if super is used as the receiver of a method, the method to be invoked will be searched for starting at the inherited classes of the class of the method currently running. self and super are similiar to the self and super variables in Smalltalk.

2 Installation and Removal

The **ROX** REXX function package is contained in the file rox.dll. This file needs to be placed in a directory along your LIBPATH. To get access to the functions in the **ROX** function package, execute the following REXX code:

```
rc = RxFuncAdd("RoxLoadFuncs","rox","RoxLoadFuncs")
rc = RoxLoadFuncs()
```

To unload the DLL, you should first call the RoxDropFuncs() function, then exit all CMD.EXE shells. After exiting all the command shells, the DLL will be dropped by OS/2 and can be deleted or replaced.

3 Function Reference

The functions provided by the **ROX** function package fall into the following categories:

- function package functions
- class definition functions
- object lifecycle functions

3.1 Function Package Functions

The following functions load, drop and query the version number of the **ROX** function package.

RoxLoadFuncs() - load the ROX function package

```
rc = RoxLoadFuncs()
```

Loads all the functions in the **ROX** package.

If ANY parameters are passed to this function, it will bypass the program, author, and copyright information normally displayed. All parameters are ignored (except to determine whether or not to bypass displaying the information).

RoxDropFuncs() - drop the ROX function package

```
rc = RoxDropFuncs()
```

Drops all the functions in the **ROX** package.

RoxVersion() - returns version number of the ROX function package

```
vers = RoxVersion()
```

Returns the current version number of the **ROX** package.

RoxStats() - generates execution profile info

```
rc = RoxStats(<parm>)
```

This function can be used to generate profile information on stderr. A parameter should be passed to start profile information, no parameter should be passed to stop profile information. For example:

```
rc = RoxStats("") /* start profiling */
rc = RoxStats()   /* end profiling */
```

The profile information can be analyzed with the RoxProf.cmd utility.

Returns "".

3.2 Class Definition Functions

The following functions are used to add class definitions to the system. Generally you will only need to use RoxLoad() and RoxQueryClassLoaded(). The other functions are used by RoxLoad() to to load .rox files.

RoxLoad() - load class definitions in a .rox file

```
rc = RoxLoad(roxFileName)
```

This function loads the named file as a class definition. See the section of .rox file definitions for the layout of the file.

This function is implemented as a REXX .cmd file.

RoxQueryClassLoaded() - query whether class is loaded

```
bool = RoxQueryClassLoaded(className)
```

Returns 1 if the class named className is available in the system. Returns 0 otherwise.

RoxAddClass() - add a class

```
rc = RoxAddClass(className)
```

This function adds the named class to the system.

RoxClassAddInherit() - add an inherited class to a class definition

```
rc = RoxClassAddInherit(className,inheritedClassName)
```

This function specifies that the class named className should inherit from the class named inheritedClassName.

RoxClassAddMethod() - add a method to a class definition

```
rc = RoxClassAddMethod(className,methodName,methodCode)
```

This function adds the named method, with the REXX code for the method to the named class.

RoxClassAddMethodDll() - add a method (in a DLL) to a class definition

```
rc = RoxClassAddMethod(className,methodName,dllName,entryPoint)
```

This function loads the dll, gets the address of the function given with the name entryPoint, and adds this to the named class.

RoxClassAddVar() - add an instance variable to a class definition

```
rc = RoxClassAddVar(className,varName)
```

This function adds the named instance variable to the named class.

3.3 Object Lifecycle Functions

RoxCreate() - create an object

```
object = RoxCreate(className<,p1<,p2< . . . >>>)
```

This function creates an object of the class named className. Any number of parameters, specific to the class, can be passed.

RoxDestroy() - destroy an object

```
rc = RoxDestroy(object)
```

This function destroys an object.

RoxSend() - send a message to an object

```
result = RoxSend(messageName,object,<,p1<,p2<. . . >>>)
```

This function sends the named message to the object specified. Any number of parameters, specific to the message and class, can be passed.

RoxSendThread() - send a message to an object

```
result = RoxSendThread(messageName,object,<,p1<,p2<. . . >>>)
```

Same as RoxSend(), but starts a new thread to process the message. No useful return value is returned.

RoxClass() - return class of given object

```
class = RoxClass(object)
```

This function returns the name of the class of the object.

RoxAddVar() - add a variable to an object

```
result = RoxAddVar(object,varName)
```

This function will the named variable to the set of instance variables associated with the object. Be careful not to add extra blanks to varName when passing it in. The characters in the variable name, up to the first ".", will be uppercased, to conform with REXX variable conventions. The remainder of the variable name is left as is.

4 Format of .rox Files

Classes are defined in files with an extension of .rox. A .rox file may contain one or more class definitions.

Classes defined in .rox files may be loaded by using the RoxLoad function (see *Utilities Provided* on 13).

The format of .rox files is a tagged file. The character ':' in column one indicates a tag. The rest of the line after the ':' indicates the type of tag.

The characters ':*', when located in column one, indicate a comment.

The following tags may be used in a .rox file:

```
:include <file>
```

This tag indicates that the file specified in the tag should be loaded as a .rox file. Useful for including inherited class definitions from separate files.

:class

This tag indicates the start of a new class definitions. Any :inherits, :vars, and :method tags following this tag, up to the end of the current .rox file, are associated with this class.

:inherits <class> <class> ...

This tag indicates the classes that should be inherited from. More than one class may be specified. This tag may be used more than once within a class definition.

:vars <var> <var> ...

This tag indicates the variables associated with the class. More than one variable may be specified. This tag may be used more than once within a class definition. Note stem variables may NOT be used. Use RoxAddVar() to add stem variables to an object.

:method <methodName>

This tag indicates that the code for the method named <methodName> follows. The code for the method ends at the next tag (including :* comment), or end of file.

5 C Programming Interface

ROX methods can be implemented in compiled languages, such as C, via a DLL. The function RoxClassAddMethodDll() adds a method to a class that points to a function in a DLL. The function in the DLL must have the following signature:

```
/*-------------------------------------------------
 * typedef for function that handles method invocation
 *-------------------------------------------------*/
typedef ULONG APIENTRY RoxMethodHandlerType(
   void      *object,
   PUCHAR     name,
   ULONG      argc,
   PRXSTRING  argv,
   PRXSTRING  retString
   );
```

The parameters passed to the method are:

object a pointer to the **ROX** object receiver

name the name of the method

argc the number of arguments passed to the method

argv array of RXSTRINGs that make up the parameters

retString pointer to the return value

Most of these parameters will be familiar to those of you who have written external functions for REXX in C. The only new one is the object parameter. It can be used in the following functions:

```
ULONG RoxVariableGet(
   void      *object,
   PRXSTRING  name,
   PRXSTRING  value
   );

ULONG RoxVariableSet(
   void      *object,
   PRXSTRING  name,
   PRXSTRING  value
   );
```

The functions above are used to query and set variables for an object. The functions return 0 when successful, !0 when not successful. The data pointed to by the value parameter returned from RoxVariableGet() must not be modified.

A sample of a compiled class is provided in roxsem.c.

A DLL can provide a self-loading function named RoxDllEntryPoint, with the following function signature.

```
ULONG APIENTRY RoxDllEntryPoint(
   ULONG init
   )
```

Currently the init parameter is ignored.

This function gets called when the REXX function RoxLoadDLL() is invoked. This function takes the name of the DLL (usually sans ".DLL", although you may specify an absolute path, including the ".DLL" suffix) and calls the RoxDllEntryPoint function.

This function in the DLL can call any of the functions defined in the **ROX** function package through their C bindings. The call is made as if the call was being made to a REXX external function. For example, to call RoxAddClass(), you invoke it in C as:

```
RXSTRING parm, result;

parm.strptr = "myClassName";
parm.strlength = strlen(parm.strptr);

RoxAddClass(NULL,1,&parm,NULL,&result);
```

Note that the function name and queue name (first and fourth parameters) may be passed as NULL.

Be careful how the return value is freed. See the sample roxsem.c code for examples.

Two *platform* independent functions are provided to allocate and free memory. The functions are:

```
void APIENTRY *osMalloc(
   int size
   );

void APIENTRY osFree(
   void *ptr
   );
```

The include file "roxapi.h" prototypes these functions, and the library "rox.lib" contains them.

6 Utilities Provided

The following utilities are provided with **ROX**:

RoxLoad.cmd

This program can only be used as a REXX function. It can not be called from the OS/2 command line. One parameter must be passed to the function - the name of a .rox file to load. The file will be searched for in the current directory, and then the directories specified in the ROXPATH environment variable.

RoxInfo.cmd

Prints a short reference of the class definitions in .rox files. Multiple .rox files may be passed as parameters, and wildcards may be specified. For every class in the .rox file, the following information will be provided:

- Classes inherited by the class. These classes will be listed in an indentation style which indicates the *tree* of class inheritance.
- Variables defined and inherited by the class. Inherited variables are marked with a prefix of "*".
- Methods defined and inherited by the class. Inherited methods are marked with a prefix of "*".

RoxProf.cmd

Analyzes the profile information generated by RoxStats(). Use "RoxProf ?" for help.

7 Classes and Testers Provided

list.rox

Implements a simple list class. The program testcoll.cmd tests this class, by passing it a parameter of "list". The list class inherits the collection class in collect.rox.

wordlist.rox

Implements a simple list class, similiar to the list class. The difference is that the list class can contain arbitrary strings, whereas the wordlist class can only contain strings with no blanks in them. The program testcoll.cmd tests this class, by passing it a parameter of "wordlist". The wordlist class inherits the collection class in collect.rox.

set.rox

Implements a simple set class. The program testcoll.cmd tests this class, by passing it a parameter of "set". The set class inherits the collection class in collect.rox.

collect.rox

Implements a simple collection class, that can be inherited by other, more specific collection classes, and will provide additional capabilities.

sessions.rox

This file implements some of the classes from Roger Sessions' book on OO with C and C++ (reference included in the .rox file). The program sessions.cmd tests the classes.

spinner.rox

This class implements a character spinner, which can be used as a progress indicator. Also uses roxsem.dll. This class is tested with testspin.cmd. The demo shows code testing a collection along with a spinner running independently in another thread.

testthrd.cmd

This program tests the thread capabilities of **ROX**.

cmdline.cmd

This program uses cmdline.rox as a command line reader with history. Use the up and down arrows to cycle through previous lines entered.

roxsocks.cmd & roxsockc.cmd

These programs demonstrate tcp/ip server and client programs X socket class (in socket.rox).

8 History

04/14/94 - version 1.8

- fixed problem with super calls
- removed RoxVarSynch()
- added RoxAddVar() and per-instance variables
- cut execution time in half with new memory management scheme
- added RoxStats() and RoxProf.cmd

01/06/94 - version 1.7

- minor documentation cleanup
- cleanup of internal structure of **ROX** - no external changes - most notably, no performance changes

10/22/93 - version 1.6

- fixed infinite loop when no variables set in an init method - ObjectSaveState/RoxStemSynch ping-ponged. Reported by Zvi Weiss as a problem when a syntax error occurred in an init method.
- changed compiled classes/methods stuff to have just one type of class, and either compiled or REXX macros. Compiled macros added with RoxClassAddMethodCompiled().

09/14/93 - version 1.5

- more thread reentrancy fixes
- added compiled class capability

08/31/93 - version 1.4

- added RoxSendThread() function
- first attempt at making everything thread reentrant (still some more to go).

08/27/93 - version 1.3

- print error when invalid object reference is passed to a method
- added exception handling, to try to catch method invocation on objects which are no longer alive

08/24/93 - version 1.2

- fixed problems with re-adding and re-registering classes and methods

08/22/93 - version 1.1

- fixed super behaviour
- added multiple inheritance capability
- added class-specific init and deinit methods
- added RoxStemSynch() - requires user notify the system when stem variables are added or dropped as instance variables
- added RoxInfo.cmd utility
- documentation turned into .inf file and enhanced

08/18/93 - version 1.0

- initial release

A Sample .rox file

Below is a the 'sessions.rox' file, which contains class defintions inspired by Roger Sessions' book on class development.

```
:*-----------------------------------------------------------------
:* REXX Object eXtensions :
:* classes described in Roger Sessions' book "Class Construction in
:* C and C++", Prentice-Hall, ISBN 0-13-630104-5.
:*-----------------------------------------------------------------

:*=================================================================

:class performer

:vars minSalary

:method setMinimumSalary
   minSalary = arg(1)

   if (0 = datatype(minSalary,"W")) then
      minSalary = 1000

   return self

:method bargain
   say "   I get" minSalary * 2 "dollars a performance."

   return self

:*=================================================================

:class animal

:vars name sound soundTimes

:method init
   name       = arg(1)
   soundTimes = arg(2)
   sound      = arg(3)

   if (name = "") then
      name = "unnamed"

   if (0 = datatype(soundTimes,"W")) then
      soundTimes = 1
```

```
   if (sound = "") then
      sound = "..."

   return

:method says
   say name "says:"

   do i = 1 to soundTimes
      say "   "sound
   end

   return self


:*================================================================

:class dog

:inherits animal performer

:method init
   rc = animal.init(self,arg(1),arg(2),arg(3))
   return

:method scratch
   say "   Ooooh... what an itch."
   return self

:*================================================================

:class littleDog

:inherits dog

:method init
   rc = dog.init(self,arg(1),arg(2),arg(3))
   return

:method trick
   say "   Watch my trick: I can roll over."
   return self

:*================================================================

:class bigDog
```

```
:inherits dog

:method init
   rc = dog.init(self,arg(1),arg(2),arg(3))
   return

:method trick
   say "   Watch my trick: I can fetch the letter carrier."
   return self

:*=============================================================

:class usedCarDealer

:inherits animal

:method init
   rc = animal.init(self,arg(1),arg(2),arg(3))
   return

:method makeSale
   say "   ... and only $500 more if you want the wheels."
   return self
```

B Sample ROX class usage

Below is a the 'sessions.cmd' file, which uses the classes defined in the 'sessions.rox' file.

```
/*----------------------------------------------------------------
 * sessions.cmd :
 *----------------------------------------------------------------
 * 08-21-93 originally by Patrick J. Mueller
 *----------------------------------------------------------------*/

say "testing the Sessions classes"

if RxFuncQuery("RoxLoadFuncs") then
   do
   rc = RxFuncAdd("RoxLoadFuncs","Rox","RoxLoadFuncs")
   rc = RoxLoadFuncs()
   end

rc = time("r")

rc = RoxLoad("sessions.rox")

Frenchie  = RoxCreate("animal",        "Frenchie",  1, "Grrrrrr")
Rover     = RoxCreate("dog",           "Rover",     1, "Woof")
Fifi      = RoxCreate("littleDog",     "Fifi",      2, "bow wow")
Rex       = RoxCreate("bigDog",        "Rex",       4, "BOW WOW")
HonestBob = RoxCreate("usedCarDealer", "HonestBob", 1, "Buy this deal of a car!")

g = .setMinimumSalary(Rex,30)
g = .setMinimumSalary(Fifi,20)

g = .says(Frenchie)
say

g = .says(Rover)
say

g = .says(Fifi)
g = .scratch(Fifi)
g = .trick(Fifi)
g = .bargain(Fifi)
say

g = .says(Rex)
g = .scratch(Rex)
g = .trick(Rex)
```

```
g = .bargain(Rex)
say

g = .says(HonestBob)
g = .makeSale(HonestBob)
```

C Output of previous samples

Below is a the output of running the 'sessions.cmd' file

```
testing the Sessions classes
Frenchie says:
   Grrrrrr

Rover says:
   Woof

Fifi says:
   bow wow
   bow wow
   Ooooh... what an itch.
   Watch my trick: I can roll over.
   I get 40 dollars a performance.

Rex says:
   BOW WOW
   BOW WOW
   BOW WOW
   BOW WOW
   Ooooh... what an itch.
   Watch my trick: I can fetch the letter carrier.
   I get 60 dollars a performance.

HonestBob says:
   Buy this deal of a car!
   ... and only $500 more if you want the wheels.
```

The Object REXX Class Hierarchy

Simon Nash
IBM

The Object REXX Class Hierarchy

Simon C. Nash
IBM UK Laboratories Ltd, Hursley Park,
Winchester, Hants SO21 2JN, England
Internet: nash@vnet.ibm.com

Abstract

Object REXX, an object-oriented extension of the popular REXX language, includes a class hierarchy. The design of this hierarchy posed some interesting challenges in providing mechanisms that would serve the needs of the base hierarchy together with probable user extensions to it. This paper presents the chosen design in the form of a tutorial introduction to the concepts and mechanisms involved, including abstract classes, mixins, and multiple inheritance. It also gives examples of how the mechanisms provided by REXX might be used by class users and implementers.

Objects and Classes

REXX objects are grouped into classes. For example, all character strings (whatever their content) belong to the String class, all directories belong to the Directory class, and so on. The class of an object indicates what "kind" of object it is — that is, what methods it provides to respond to messages sent to it. For example, string objects provide string-related methods such as POS and SUBSTR, and directory objects provide methods for collections such as ITEMS and SUPPLIER. You can look at the descriptions of the String and Directory classes to find out what methods are available on string and directory objects.

In REXX, everything is an object, so classes are objects too. Class objects are used in a number of ways, the most important of which is their role in creating other objects. They support this by providing NEW and INHERITED methods which create objects of the kind defined by the class. For example, the Directory class object returns a new directory object in response to the message

```
.directory~new
```

Classes and Instances

The objects created by a class are known as its instances. They are given methods that match the specification defined by the class for its instances. For example, a Rectangle class might define methods AREA and PERIMETER using the directives

```
::class Rectangle
::method area
expose width height
return width*height
::method perimeter
expose width height
return (width+height)*2
```

Then, when rectangle objects (instances of the Rectangle class) are created by sending NEW messages to the Rectangle class object, they will have methods AREA and PERIMETER with the REXX code shown above.

Object and Instance Methods

There's an important difference between the AREA and PERIMETER methods of a rectangle object and the AREA and PERIMETER method definitions in the Rectangle class. A rectangle object can respond to AREA and PERIMETER messages by running the methods shown above, and we say that it has these as *object methods.* The Rectangle class cannot respond to AREA and PERIMETER messages itself, but its instances can, and we say that it has AREA and PERIMETER as *instance methods.* The Rectangle class responds to other (class-related) messages, such as the NEW message that creates a rectangle object, so it has object methods (like NEW) as well as its instance methods.

Since only classes have instance methods, there's no need to distinguish between object methods and instance methods when talking about other kinds of objects, such as strings or rectangles. We usually just say plain "methods" when talking about the object methods of these objects.

Subclasses, Superclasses, and Inheritance

Every class in the system could be defined independently, with a complete set of instance methods. However, many classes have a lot in common. An example of this may be Student and Graduate classes — a graduate object has the same information as a student object (name, ID, course, etc.) and also some additional information (graduation details). We'd prefer not to repeat most of the instance methods of the Student class in the definition of the Graduate class, and we can avoid this (and express the close relationship between these two kinds of objects) by making the Graduate class a subclass of the Student class. This gives the Graduate class all the instance methods of the Student class, and the Graduate class can then add or override any necessary instance methods.

If Graduate is a subclass of Student, we call Student a superclass of Graduate. The subclass-superclass relationship is also called inheritance, so we say that Graduate inherits the NAME method from Student.

The inheritance relationship can be used to arrange the classes into a class hierarchy — a diagram in which superclasses are drawn above subclasses, with lines connecting them. The class at the top of this hierarchy is the Object class. Its instance methods (COPY, "=", STRING, etc.) are inherited (directly or indirectly) by all other classes and so become object methods of all objects. Most objects have additional object methods — for example, the Supplier class is a subclass of the Object class and has instance methods AVAILABLE, INDEX, ITEM, and NEXT, so all supplier objects have these object methods as well as COPY, "=", STRING, etc.

You can create a subclass by specifying the name of the superclass on the SUBCLASS option of the ::CLASS directive (which is equivalent to sending a SUBCLASS message to a class object). For example, to make Graduate a subclass of Student, you would write

```
::class Graduate subclass Student
```

If Student were itself a subclass of Person (inheriting some Person methods), this makes Grasuate a subclass of Person too (through the intermediate class Student). We sometimes use the terms direct and indirect superclasses (or subclasses) to distinguish these. If you don't specify the SUBCLASS option, your class becomes a subclass of the Object class.

When talking about a class's instance methods, which ones do we mean — just the ones it defines itself, or those and the ones it inherits from its superclasses? It's usually more convenient to take this as meaning the methods defined by the class itself, and we will follow this convention from now on. However, it's important to remember that when the class creates instances, the object methods of the instances include not only the instance methods of the class itself, but also those of all the superclasses from which it inherits.

Abstract Classes and Object Classes

Some classes have a close inheritance relationship, like Graduate and Student. Others are related in a slightly more distant way — more like siblings than parents and children. You can appreciate why the term "inheritance" is used to describe the class family! For example, array and list objects share a number of methods: FIRST, LAST, NEXT, PREVIOUS, SECTION, and SUPPLIER. Even so, neither is a subclass of the other — arrays have a DIMENSION method, but lists don't, and lists have a FIRSTITEM method, but arrays don't. So how can we express the common nature of arrays and lists?

The answer is an *abstract class*. Abstract classes are special classes that don't create instances (unlike "normal" classes, like Student and Graduate). Instead, they provide a set of instance method definitions that can be shared by a number of other classes. It's helpful if abstract classes define meaningful properties, with a collection of methods that relate to that property. For example, the property shared between arrays and lists is that of having some internal sequence which can be used to step through the items of the array or list — the idea of a "first" item, "next" item and so on. This leads naturally to the idea of a Sequenced class — but it's not a "normal" class, since it isn't meaningful to think about making instances of the Sequenced class. That's because the Sequenced class doesn't provide enough capability for a functional standalone "sequenced" object. Array and List inherit from Sequenced and add the missing pieces that Sequenced doesn't have.

We need a name for "normal" classes (that can create instances) to distinguish them from abstract classes. We call them *object classes* because these are the classes whose members (instances) are real live objects. Their names are usually nouns, such as Array, List, and Rectangle. In contrast, abstract classes define properties (or abstractions) that describe objects — they have no instances, and their names are usually adjectives like Sequenced.

Because of the way abstract classes factor out the common methods from their subclasses, you'd expect them to always have more than one subclass. This is true for all the abstract classes that REXX provides except the Condition and Supplier classes. These aren't object

classes because they don't provide NEW or ENHANCED methods for creating instances — REXX provides other ways to create condition objects and supplier objects. They're a very special case (since no user-created classes would be able to work like this), and it's convenient to use abstract classes for them.

To create an abstract class, use the ABSTRACT option on a ::CLASS directive. For example, to create an abstract class Visual which is a subclass of the Object class, you would write

```
::class Visual abstract
```

Multiple Inheritance

As well as sharing some methods with arrays, lists also share some methods with queues: MAKEARRAY, PEEK, PULL, PUSH and QUEUE. Again, it makes sense to create an abstract class for these. We call it Queuelike, since its instance methods apply to all objects that function as queues. So we need the List class to inherit from both the Sequenced and Queuelike abstract classes. This is called multiple inheritance, and although it may look quite simple (at least in this case), it is very powerful. It also raises some rather complicated issues — see More on Multiple Inheritance below.

You can use multiple inheritance by specifying the INHERIT option on a ::CLASS directive (which is equivalent to sending one or more INHERIT messages to a class object). For example, to create a class Window which is a subclass of Visual and also inherits from Movable and Sizeable, you would write

```
::class Window subclass Visual inherit Movable Sizeable
```

There's no limit to the number of classes you can inherit from in this way.

Class Methods

We've seen that class objects have both instance methods and object methods. How are their object methods (like NEW) defined? The CLASS option on a ::METHOD directive indicates that the method being defined is a *class method*, not an instance method. For an object class, this means that the class will have that method as one of its object methods. For example, the Array class defines OF as a class method, and this allows OF messages to be sent to the Array class object to create array objects whose contents are specified by the arguments to OF.

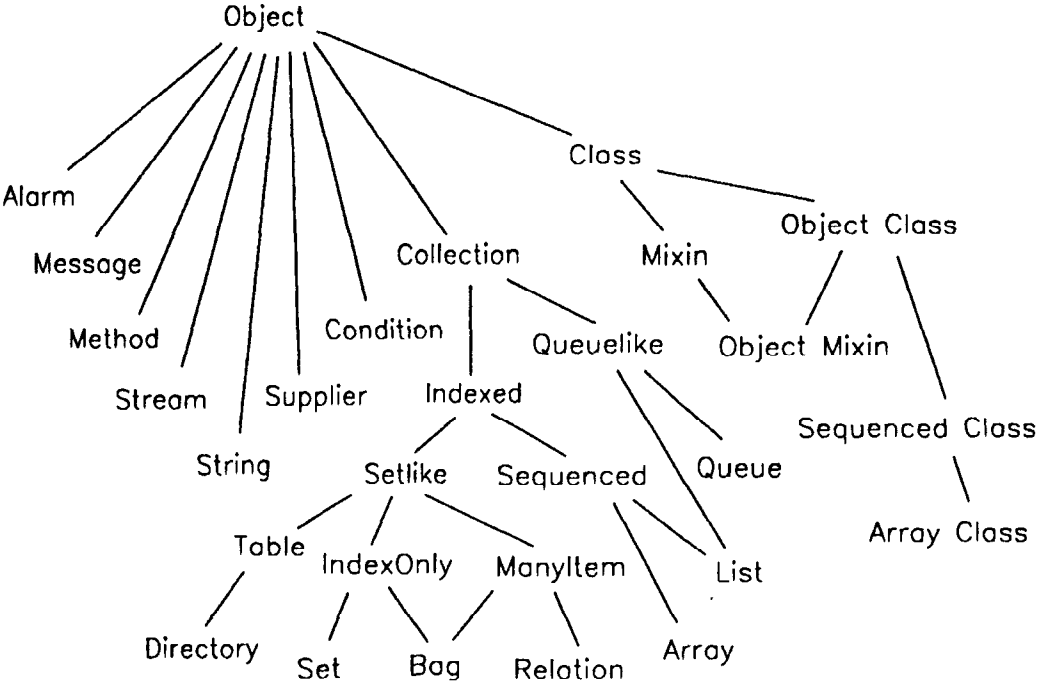
What about abstract classes — can they have class methods too? They can, but their class methods work slightly differently than those of object classes. They are defined in the same way, with the CLASS option on a ::METHOD directive, but they don't become object methods of the abstract class itself. Instead, they become object methods of any object classes that inherit (directly or indirectly) from the abstract class in which they are defined. For example, the Sequenced class also has an OF class method, but OF messages can't be sent to the Sequenced class to create "sequenced" objects (because the Sequenced class is abstract and so can't create objects). Instead, OF becomes an object method of any object classes that inherit from the Sequenced class, such as the List class. The List class doesn't have to do anything (except inherit from the Sequenced class) to make this happen.

So what object methods do abstract classes have? They all have the same ones: DEFINE, DELETE, ID, INHERIT, INITA, METHOD, METHODS, SUBCLASS, SUBCLASSES, SUPERCLASSES, and UNINHERIT. Of course, like all objects, their object methods include the instance methods of the Object class: COPY, "=", STRING, etc. Object classes have two additional object methods: NEW and ENHANCED, the methods that create objects.

Class methods are inherited in exactly the same way as instance methods. For example, the List class inherits the OF class method from the Sequenced class, just as it inherits the FIRST, LAST, MAKEARRAY, NEXT, PREVIOUS, SECTION, and SUPPLIER instance methods.

The Class Hierarchy

We've mentioned a number of REXX classes and the inheritance relationships between them. Let's take a look at the complete hierarchy for the classes provided and used by REXX.



That looks a bit daunting, but the REXX user doesn't have to be concerned with many of these classes. A number of them (Collection, Indexed, IndexOnly, ManyItem, Queuelike, Sequenced, and Setlike) are abstract classes used only for internal factoring out of common methods. In addition, the metaclass section of the hierarchy (Class, Mixin, Object Mixin, Object Class, Sequenced Class, and Array Class) is shown for completeness but isn't for general use (see the section on Metaclasses below). That leaves us with the classes that represent other objects: Alarm, Array, Bag, Condition, Directory, List, Message, Method, Object, Queue, Relation, Set, Stream, String, Supplier, and Table.

More on Multiple Inheritance

We used the List class to introduce multiple inheritance. The List class inherits from the Sequenced and Queuelike abstract classes, which means that lists have both the Sequenced and Queuelike properties (collections of methods). Another way of saying this is that a list can be used whenever either a queuelike or sequenced object is expected, and it will work correctly.

Multiple inheritance can be very useful if used properly, but it can cause a lot of problems when it's used incorrectly. This has made it quite controversial in object-oriented circles! A common mistake is to think of it as a "magic" way to combine two different objects into one — such as a hybrid of the Rectangle and List classes. That probably sounds rather ridiculous, but other examples can seem more plausible. For example, if I have classes Directory (which keeps names and some information for each name) and Phone (which dials a number passed to it), can't I create a Phone Directory class (which keeps names and phone numbers, and dials the number when given a name) simply by inheriting from Directory and Phone?

Unfortunately, it's not usually that simple. The reason is that Directory and Phone were each designed to do a specific job, with a set of methods appropriate to that job, but neither was designed (probably) to "mix in" with the other. For example, the DIAL method of Phone was designed to be given a number to dial, and multiple inheritance won't make it smart enough to change its behaviour to take a name instead and look it up in the directory. (By the way, the right way to do this is called aggregation, which means creating a new object that contains Directory and Phone and provides the right connections between them.)

So when is multiple inheritance useful? Actually, the answer's in the paragraph above. It's useful for classes that have been specially designed to "mix in" with other classes — like the Sequenced and Queuelike classes, which were specially designed to mix in with each other. However, these classes were not designed to mix more generally with other classes. You can try mixing them with other classes (REXX won't stop you), but it's unlikely that anything useful will result. For more general "mix in" classes, we'll have to look elsewhere in the hierarchy.

Mixins

A class that is designed to be mixed in with other classes in a general fashion is called a *mixin*. For example, the ManyItem mixin can be inherited by any setlike class to allow multiple items to have the same index, and is used by the Bag and Relation classes.

It's important to understand the difference between mixins and abstract classes. Both can be used with multiple inheritance, but their purposes are very different. Abstract classes are for the convenience of a class hierarchy implementer, to prevent the same methods being duplicated among more than one object class. They are of little use in themselves, but enable the construction of object classes below them in the hierarchy. They are not part of the public interface of the class hierarchy for inheritance.

A mixin, in the other hand, allows some class (and all the classes below it in the hierarchy) to be enhanced in some way. For example, if Persistent is a mixin to the Object class, all classes

in the hierarchy may exist in persistent and non-persistent versions. The persistent versions inherit the Persistent mixin, but the non-persistent versions don't. For example, to make a persistent directory class, you would write

```
::class PersistentDirectory subclass Directory inherit Persistent
```

which tells REXX that the PersistentDirectory class inherits from the Persistent mixin as well as the Directory class. Any number of mixins may be inherited, and a combination of inherited mixins and other inherited classes may be specified.

Since Persistent is a mixin to the Object class, it applies to all subclasses of the Object class — that is, all classes. Some mixins are more specialized — for example, the ManyItem mixin is a subclass of the Setlike class and so only applies to classes that inherit from the Setlike class. No other class (for example, a subclass of the Stream class) is allowed to inherit the ManyItem mixin. This is because the ManyItem mixin has been designed specifically to enhance the Setlike class (it's "tailor made" to fit this class only) and won't fit any other class. The Setlike class is called the base class of the ManyItem mixin, and we say that ManyItem is a mixin to the Setlike class.

So mixins, like object classes but unlike abstract classes, are intended for users of the class hierarchy and are part of its public interface for inheritance. They provide enhancing options for the object classes in the hierarchy, to be included or excluded at the user's discretion.

Some mixins provide a complete set of methods for some property, so that a class can acquire that property just by inheriting from the mixin. Other mixins define a property, and provide some of the methods required, but depend on subclasses to provide other necessary methods. For example, a Persistent mixin may provide methods that take care of saving object data to stable storage and restoring it when needed, but not the methods that actually extract the object's essential data (when saving) and recreate its state from saved data (when restoring). Those methods may be left as placeholders in the Persistent mixin, needing to be filled in by classes that inherit the persistent property from it, since only they know the intimate details of how they are constructed.

To create a mixin, use the MIXIN option on a ::CLASS directive. For example, to create a mixin OrderedSet which has a base class of Set, you would write

```
::class OrderedSet mixin subclass Set
```

The Class Search Order for Methods

In a single-inheritance hierarchy, classes inherit methods from their ancestors in the hierarchy. Since every class has exactly one superclass (except the root class Object, which has none), there is a simple line of inheritance from each class up to the root class Object, through any intermediate ancestor classes. This line of inheritance defines a search order for methods (the class search order). The order is important because more than one ancestor class may have an instance method with the same name — like PRINT. When a PRINT message arrives at an instance of the class, it's important to know which PRINT method will be run. The search order starts with the lowest class in the hierarchy (the class to which the instance that received the PRINT message belongs) and proceeds upwards to its superclass, then its superclass's

superclass, and so on up to the root class Object. The first PRINT method found is the one that gets run.

With multiple inheritance, the situation is quite a bit more complicated. Classes may have many superclasses (direct and indirect), and there may not be an obvious "right" order of searching them for a method. The rules REXX uses are:

1. A subclass is always searched before its superclasses.
2. Mixins are searched immediately before their base class.
3. Where multiple classes appear on the INHERIT option of the ::CLASS directive, the classes are searched in the order they appear (leftmost first).

If there is no search order that satisfies all these rules, or if a mixin is inherited without its base class already in the search order, the inheritance is in error.

What about multiple inheritance from object classes? It's this sort of thing that gave multiple inheritance a bad name. There are very few cases (if any) when it would be appropriate, but REXX doesn't prevent it – it doesn't seem right to limit the powers of object classes (compared with abstract classes) by making a special restriction here. Beware, though! Before doing this, you should see if your hierarchy can be restructured to make one of the superclasses an abstract class or a mixin, or consider whether aggregation (combining two objects into a composite object, as in the Phone Directory example) isn't a more appropriate way to accomplish what you want. It usually will be.

Metaclasses

For most users of Object REXX, the concepts and mechanisms presented so far will be all they need to create instances, subclasses, abstract classes, and mixins – making full use of the facilities that REXX provides for using and extending the class hierarchy. This section and the next one complete the picture for those who are curious to know more about how all this works, or need to understand or reprogram the underlying mechanisms of the class hierarchy.

Are class objects instances of some class? For completeness and consistency, it would be nice if they were. We call these special classes metaclasses. Their instances are classes, like the Supplier class and the Sequenced class. How many metaclasses are there? There could be one for each class (as in Smalltalk), but it's not necessary to go this far. However, we do need a metaclass for each class that has a different collection of class methods. To see why this is, let's look more closely at how class methods work.

The class methods of the Object class are NEW and ENHANCED. This means that they will be object methods of the Object class and every other object class that inherits from the Object class (that is, all object classes). A metaclass is needed to create these classes, and this metaclass needs NEW and ENHANCED instance methods so that its instances (the object classes) will have NEW and ENHANCED object methods. Let's call this class the Object Class metaclass.

Suppose we create a subclass of the Object class with another class method – for example, a Database class with a RESTORE class method to restore the previously saved state of an

object. There will have to be a new metaclass to create this class, since the Object Class metaclass doesn't have our RESTORE method. Let's call this new metaclass (with a RESTORE instance method) the Database Class metaclass. The Database class is an instance of the Database Class metaclass, and the RESTORE instance method of the Database Class metaclass becomes the RESTORE object method of the Database class.

How do these metaclasses fit into the hierarchy? As well as their specialized instance methods that correspond to the class methods of their instance classes, they have instance methods for all the standard object methods of classes: DEFINE, DELETE, ID, INHERIT, INITA, METHOD, METHODS, SUBCLASS, SUBCLASSES, SUPERCLASSES, and UNINHERIT. We need a class with these as its instance methods (they need to be instance methods somewhere) and we call this class the Class class. It's natural to make the Object Class metaclass and Database Class metaclass subclasses of the Class class, since they can then inherit all its instance methods listed above.

Which class is the metaclass for abstract classes? Their object methods are the ones that are shared by all classes: DEFINE, DELETE, etc. Since these are the instance methods of the Class class, the Class class is the metaclass for all abstract classes.

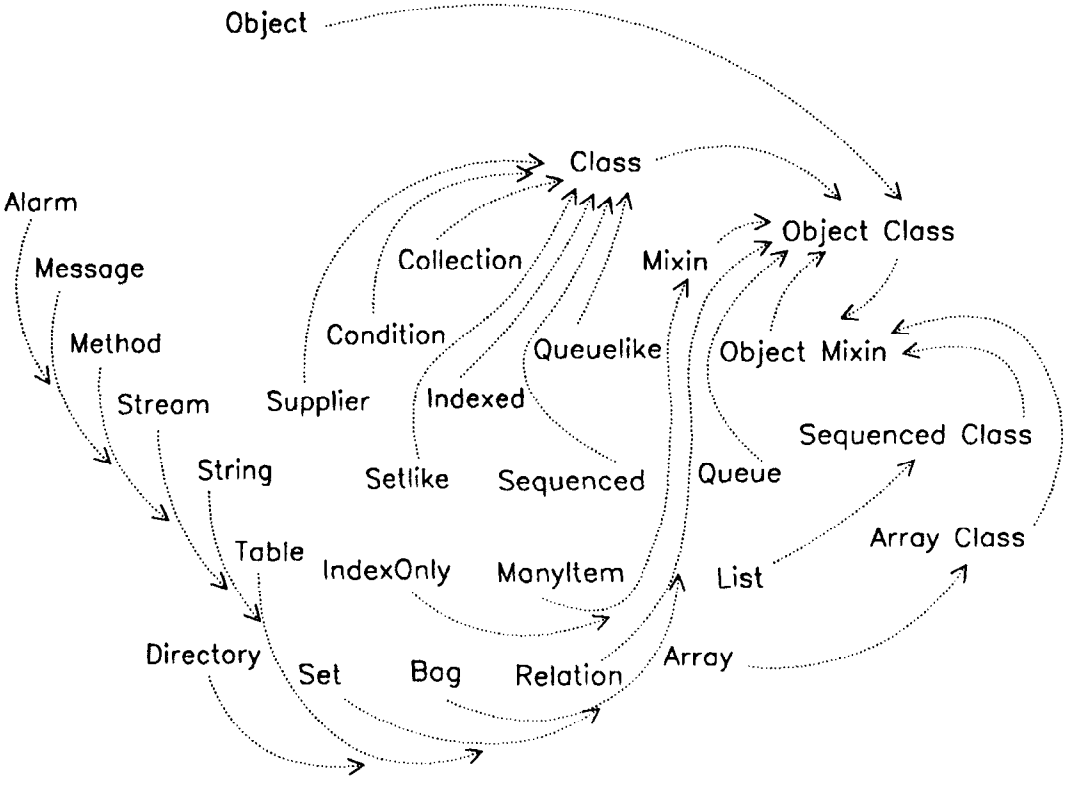
What about metaclasses for mixins? Mixins are very similar to classes, only differing in their inheritance rules, so we make the Mixin class (the class whose instance methods are object methods of all mixins) a subclass of the Class class. Is the Mixin class the metaclass for all mixins? It isn't, for the same reason that the Class class isn't the metaclass for all classes — just as different classes have different object methods, mixins do too.

Let's take an example to see why different mixins have different object methods. If we create a Relational mixin to our Database class, with instance methods but no class methods, what object methods does the Relational mixin have? They include all the standard mixin object methods (the instance methods of the Mixin class) as well as the inherited class methods: NEW, ENHANCED, and RESTORE. We want these as object methods because we want the Relational mixin (as a mixin to an object class, or an *object mixin*) to be able to create instances in its own right. If it couldn't, we'd have to create another object class (inheriting from Database and Relational) which could create these instances — adding an unnecessary class to the hierarchy.

It looks as though we might need a Relational Mixin metaclass to create the Relational mixin. In theory, we do; in practice, we don't. By making metaclasses mixins (with a base class of the Class class), they can also be inherited by subclasses of the Mixin class (since Mixin is a subclass of Class). So the Database Class metaclass becomes the Database Class mixin, and REXX can construct the Relational Mixin metaclass simply by inheriting from the Mixin class and the Database Class mixin. That's what mixins are all about!

Classes and Metaclasses

The earlier hierarchy diagram showed the superclass-subclass relationship of the REXX classes. It didn't show the class-instance relationships. Of course, these connections will be quite different, so it could be confusing to try to show both on the same diagram. We'll use a separate diagram here to show the class-instance relationships of these classes.



The classes are shown in the same positions as before, but the inheritance connections have been replaced by arrows which point from each class to its metaclass (from instance to class). There's an interesting circularity between the Object Class mixin and the Object Mixin class — each class is an instance of the other. This is a bit of a mindbender, and reminds me of the chicken and the egg question — how did these classes get created? Let's just say that someone had to do a little bit of cheating here.

Last Words

Don't worry if multiple inheritance, abstract classes, or mixins seem difficult or unnecessary. The simplest classes are the object classes. They create objects that do a particular job which is well-defined by their class definition. They are the place to start in familiarizing yourself with object-oriented programming, and in creating your own classes. Start by subclassing object classes, with single inheritance. Override a few methods and get a feel for how subclasses can be different from, yet similar to their superclasses. Then try multiple inheritance with a mixin, getting a feel for how that works. When you have developed a few object classes, you may start to notice relationships between them that don't match the

hierarchy — similar or identical methods cropping up in different places. That's the time to think about making use of abstract classes — to bring the relationships between your object classes into clearer focus.

This ongoing refinement of the class hierarchy is a hallmark of good object-oriented programming — seeing new relationships between your classes, and finding better ways to structure the hierarchy to express those relationships. Don't try to start out by designing a set of 20 abstract classes, 50 object classes, 15 mixins and all the relationships between them. You won't get it right at the first attempt! Far better to develop your hierarchy gradually, refining it as you acquire a feel through hands-on experience of how the classes relate to each other.

Summary

We have seen how object methods, instance methods, and class methods are used in Object REXX. The need for object classes, abstract classes, and mixins has been explained, together with guidelines for when they should be used and how they relate to single and multiple inheritance. The use of all the above facilities of the REXX language has been illustrated with examples from the class hierarchy provided by REXX. Finally, the role of metaclasses in completing the picture has been shown.

Acknowledgements

The main structure of the REXX class hierarchy was developed in a meeting of the REXX Architecture Review Board, with contributions from Jim Babka, Mike Cowlishaw, Brian Marks, Rick McGuire, and the other board members. The details took shape over several design iterations, with vital contributions and encouragement to continue from Jim Babka, Brian Marks, and Dave Renshaw.

Portable REXX Applications and Reusable Design

Edmond Pruul

"Portable Rexx Applications and Reusable Design"

Edmond A. Pruul

RD 1 Box 632
Afton NY 13730
USA
Electronic Mail: p00146@psilink.com
Voice Mail: 1-607-693-1030

ABSTRACT

The application owner and developer want to port their applications to new operating system environments. The Rexx language offers inherent advantages: readability, an active Standards organization, available source code, good input-parsing functions, easy source-level debugging; but no practical breakthrus have been identified in the Rexx community. The same problems that plague application portability in general apply to Rexx programs also: unwieldy code for several operating system environments; ownership conundrums; interface confoundment; and many other problems, old and new. Using Rexx as an example of a language whose applications should port easily, reveals the intractability of the portability problem. Reusable Code is problematic in startlingly similar ways. Reusable Design is a promising paradigm for a general attack on the Reusable Code problem. A Rexx application, being readable (accessible) by the average programmer is a possible stage to experiment, in a practical way, with Reusable Design. The Reusable Design paradigm is based on the classic principles of modularity in Computer Science. It can include object based or object oriented methods but the prime principle is semantic as well as syntactic readability -- the actions of the Reusable function are clear and concise to programmer. Readability allows early planning by potential reusers -- customers for a reusable function. As the Rexx Application developers rely more on reusable components, market forces could encourage the proliferation of popular reusable components to popular operating systems.

PREFACE

The thoughts and experience herein are those of an operating system coder and designer from 1968 to 1992 and do not pretend to be current in this year 1994.

PORTABLE Rexx APPLICATIONS AND REUSABLE DESIGN

Outline.

1. Porting Applications: motivation.
2. Measuring Success: when to stop?
3. Code Reuse: promise and problems
4. It is the Design, stupid.
5. Why Rexx?

PORTING APPLICATIONS: WHY BOTHER?

Definitions and 'ground rules' help address a problem. '**To port**' means to change a product such that it works in two or more environments. An example of a portable product is a 'Walkman' -- a personal tape player. A Walkman ports very easily around the world; the tape player's motor runs on DC batteries so the local power system's voltage and frequency are irrelevant to the Walkman. An '**application**' is a complete -- not a partial -- product. We focus on customer-related computer products. A complete operating system is an application. Rexx is an application. A PS/2 is not since it is bare metal -- it is not complete. A pre-loaded PS/2 is an application. Note the definition is broad. A ground rule in problem solving is to ask whose eyes to use. One could say we must know the scope of the problem or perhaps its environment. Our scope is strictly a business viewpoint -- a marketplace. We shall examine moving computer related products to another marketplace.

Compare your notions of 'Why Bother?' with these.

_ Increase Marketplace Share. One would think making that increased profits result from increasing market-share. What metric will predict our success? Will the cost of development and maintenance exceed revenues? That is the question.

_ Promote brand name recognition: "We have it all!" Or Foot-in-the-Door Syndrome: "Some day we will have it all on your computer."

_ Protect the product owner's other products in target. Spread development costs when one market would not profit the product owner. When American engineers looked at designing a small personal tape player they may have thought.

"We can not make a profit in the '60 hertz, 110 volt' marketplace. The cost of a port would be too high. Light weight batteries would not be powerful enough. We must package a different motor. We have no idea how to write diagnostic messages in Kanji or Cyrillic. (Add your own problems here)."

_ Clone a nifty application for my computer! Is cloning market related? Maybe. Examine the cloning of applications. What are our real motives? Are we violating patents or depriving someone of their copyrights?

MEASURING SUCCESS

Quality is the obvious metric and we know that surveys measure quality reasonably well but customer surveys are not predictive. What exactly does quality mean? Does quality mean delivering on time; or delivering a product that works as well as we can make it; or delivering what the customer wants? They all are good goals that any product should meet if possible. Assuming we could measure our product using these three quality goals how do we weigh the three against each other. We know that one of the definitions eventually must take priority. The developer can not decide if a port will be successful until we know which definition of quality the target marketplace demands.

Consider these metrics vis-a-vis porting applications.

_ Product makes a profit. Sadly profit is not a timely metric.
_ Maintenance cost: we can predict simple costs such as help lines, change teams, continuous market research, advertising, code control systems, legal fees. Often measuring failure is possible while measuring success eludes us.
_ The product looks and feels the same in both environments.
_ The product looks and feels like other applications in the new environment.

CODE REUSE: WHY IS IT PROBLEMATIC?

Does the Walkman have reusable components? What happens when the batteries run down? The engineers decide to put a DC plug in the Walkman so that anyone can buy a Reusable Component called an AC/DC Converter. The problem is there is no common voltage for battery powered appliances. Every engineer picked a different voltage -- 9.65, 13.1 and other peculiar voltages. Good try, engineers. Perhaps a variable voltage converter would be a better Reusable Component.

Think of examples of good reusable code: string.h in classic C libraries, Rexx functions such as STRIP or WATCOM's VXRexx, a 'visual editor' for Rexx on OS/2. Intuitively reusable components will be clearly useful if there is a big gain. The **mass** attribute could be due to many potential reusers or big functions replacing large amounts of new code. Consider the Rexx interpreter. It is an excellent example of a reusable 'scripting' component. Why? The Rexx reuser gets much more than originally specified, or **serendipity**. Rexx is massive since it replaces large amounts of code, Rexx is **mature**; Rexx is used by millions[1] of amateur programmers Rexx is **robust**; it does not break. Another good example is IBM's XEDIT used as an application base; the reuser's customers gain strong editing and searching function gratis.

The most gain for Reuse components is from serendipity and mass. Maturity and robustness are problematic.

Good designers know how to design things when they are expert. In practice the problems of general-purpose code-reuse by an average programmer are overwhelming. Consider these problems.

_ Maintenance: who maintains components; how to compensate the maintainer; how to control many versions[2]; lose of intellectual control as time passes and persons pass on to new jobs and the next life; delivery of new function and service including preventive service.

_ Disappearing customer base: First, our reuse candidate loses a prospective customer. So development stops. Next month a new customer surfaces, the project restarts only to disappear again.

_ How to measure reliability or quality of a reusable code component. How does customer convince management to trust reusable code? What would be the service cost projection?

_ Publicity: how, where, and issues of truth in advertising touch on personal sensitivities.

_ Packaging: When would we bind reusable components to the reuser's program? It could bind when compiled, at product build, at installation of the product, when the application loads or at run-time.

_ Myopic design and semantic provincialism: a coworker needed a subroutine to test, in a secure way, if a person is a "SFS Administrator." The words mislead. In fact, the programmer wrote a routine to test if a process-id is acceptable by a named resource manager for a certain specific authority. The word "SFS" is superfluous. The word "administrator" implies a permanent attribution of a human being. Worse, this label implies the reusable component has some power to enforce or guarantee its response for some un-stated period of time. False, the answer is advisory only. The power of authorization remains with the resource manager's authorization mechanism.

REUSABLE DESIGN: CAN WE HAVE SERENDIPITY AND MASS?

Reusable design could mean "good external design." Good syntax is a given: simple, targeted for performance,[3] no surprises and no side effects. Semantic clarity is the rub. Cultural tunnel vision is problematic by definition. The cure is an accessible and readable design. Early disclosure and serious attention to criticism are good; continuous disclosure is better. Rexx Library functions are outstanding example of reusable components; the required attributes are present: one responsible person, expert in the field, serious helpful customers.

What can we do?

_ Study and understand today's and tomorrow's methodologies: Temporal Logic; Gries' Axiomatic technique; SMALLTALK; Finite State Machines; Data Flow Analysis; and Event Analysis amongst many others.
_ Buy and read books.
_ Take all the design courses available; retake them a few years later.
_ Practice off the job.
_ Volunteer for inspections. The more design and code we study the wiser we are.
_ Join the local Reuse Advisory Board, evaluate reuse code candidates.
_ Read code and designs.
_ Join or get advice from the local Wisemen Council.

WHY Rexx?

_ Standards Group is in-place and active. Rexx semantics as well as syntax are consistent.

_ Slivers are easy to implement.

A sliver is the slimmest possible layer between a portable application and the complete computer system it would run on. Syntax errors will occur in the sliver. Semantic errors are harder find and harder to isolate. "The final" semantic error may be impossible to find; An application can not port to environments that are semantically incomprehensible to the original environment.

_ Universality: Rexx will run on all new operating systems[4].

_ Readability or accessibility: Rexx is justly famous for first run successes. Problems in Rexx code are rare. Errors are most likely when calling system commands. The author's first Rexx program worked perfectly on its first test; in Rexx circles this experience is not a surprise.

_ Debuggability: Rexx has implicit symbolic debugging; Rexx programs can be distributed as human readable code. There is an optional compiler but normally Rexx programs are interpreted from the source code. Anyone could read the program to solve a problem and test a fix.

---

1 This number seems too large but the author extrapolated it from an estimate by Bill Fischofer in 1991. He calculated the number of VM users to be 30 million.

2 Need strong code control systems.

3 A Rexx example is LEFT and SUBSTR, the former being a special case of the latter.

4 Author's opinion.

REXX for CICS/ESA

David Shriver
IBM

REXX for CICS/ESA

David Shriver

May 4, 1994

Contents

- Disclaimer

This discussion is about REXX for CICS/ESA, a set of products that IBM has announced an early customer program for, but has not announced for general availability.

- Copyright

- Trademarks

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

CICS/ESA, IBM, MVS/ESA, OfficeVision, OS/2

What is "REXX for CICS/ESA"

- Two products

- REXX language support for CICS/ESA

  - Native CICS application environment

  - Run-time facility

- And More

What were design goals for REXX/CICS

- Focus on Productivity
- Common REXX (across CICS platforms)
- Production REXX (suitable for use in a production environment)
- Distributed REXX (Client/Server enabled)
- CICS REXX (REXX language under CICS with CICS interfaces)
- Integration Platform (REXX is natural application integration platform)

- REXX prototype to IBM PP
  - From Assembler & REXX to PL/X & REXX for portability
- FROM TSO/E REXX base to direct use of REXX kernel

Why Now

- Growing popularity of REXX
- Growing emphasis on productivity
- Additional REXX implementations
- Product requirements
- Need for Application Integration platform
- Need for Common, Production REXX
- Need for high-level Client/Server support

Highlights

- Full REXX 3.48 language support under CICS
- Dynamic EXEC CICS command level support
- REXX interface to CEDA, CEMT
- DB2 Interface (SQL statements & DB2 commands)
- CICS native text editor for REXX execs and data
- High-level VSAM-based REXX file system (RFS)
- Execs may also be run from MVS Partitioned Datasets
- High-level Panel I/O facility
- Support for REXX Subcommands written in REXX
- Pseudo-conversational support
- System and user profile exec support
- Shared execs in storage (via EXECLOAD & EXECDROP)
- High-level Client/Server interfaces

Need for REXX/CICS

- As a tool to streamline support staff activities
  - CICS Systems Programers and Administrators
  - DB2 Analysts
  - CICS and DB2 testers, other support staff
- More productive CICS application development
  - Native CICS development (simpler)
  - Enjoy the strengths of REXX under CICS
- More flexible, powerful product customization & extension (macros)
- Quick prototyping and procedural language functions
- Preserve REXX investments in migrations
- Needed for products with REXX requirements
- As a script language to automate/streamline development sequences
- Help enable enterprise-wide Client/Server computing
- Better enable CICS end-user computing
- CICS Application Integration

REXX/CICS Basic Environment Support

- Invoking REXX execs

- Where execs run

REXX File System (RFS) Features

- Hierarchical Directory structure (like OS/2, AIX)
- VSAM RRDS based
- No need to register most new users
- No need to register individual EXECs
- Import/Export to MVS Partitioned Datasets
- Management functions for members (COPY, DELETE, RENAME)
- FLST file directory interface utility
- An EXECIO-like I/O utility (RFS)
- Maximum records per member is approx. 2**32 minus 2
- Maximum VSAM datasets in a RFS filepool is 511
- Number of filepools is only limited by DASD

REXX/CICS Summary

- REXX/CICS is more than just support for another language
- REXX/CICS introduces significant new capability
- REXX/CICS provides new approaches to CICS computing
- REXX is a good integration platform
- REXX is useful for serious programming
- REXX is a natural for Client/Server computing
- REXX is in step with current trends
- REXX + CICS = Greater Productivity

- ? ? ? ? ?

Foils 5.3	REXSYM94 - 11 foils - 0 notes
Apr 17/92	Apr 29, 94 - 17:12
Fonts Format	MIX Foils
Options	Pitch 6 10 - Bind 7 7 Tags . . .
DCF	Release 4.0.0

Working (and Playing!) with REXX and OS/2 Multimedia

Timothy Sipples
IBM

Working (and Playing!) with REXX and OS/2 Multimedia

Timothy F. Sipples
IBM Personal Software Products
Chicago, Illinois
(312) 245-4003
(312) 245-7624 fax
Internet: usib58c5@ibmmail.com

What is Multimedia?

- Combining still images (bitmaps), animation, software motion video, text, and/or audio to present information
- Principle technologies: CD-ROM, laserdisc, digital audio, MIDI, high resolution displays with more colors
- Principle file formats: WAV, MID, and AVI
- Multimedia NOT invented by Comptons

...We like to call it ULTIMEDIA

Why use Multimedia?

- "It's the market, stupid."
- Triggers: to "describe" events
- Education/training
- Kiosks (point-of-sale)
- Presentations and demos
- Better human interfaces generally
- Entertainment and games

Requirements for OS/2 Multimedia

- OS/2 2.1 with Multimedia Presentation Manager/2 (included)
- Standard OS/2 hardware requirements
- Some additional RAM (to 12 MB) recommended
- CD-ROM drive
- Audio adapter (Creative Labs, MediaVision, IBM, etc.)
- Display with at least 256 colors recommended
- Video capture adapter (optional)
- Laserdisc player with computer control (optional)

Major Features in MMPM/2 1.1

- Multimedia folder
- Sound setup object (for system sounds)
- Applets

- Volume control
- Drivers
- Sample files
- Lotus 1-2-3 and Excel audio macros
- External function library for REXX and help file

Principle File Formats

- Generally a superset of Windows file formats
- WAV: Digital audio (pulse code modulation, and variants)
- MID: Standard MIDI file format (for instrumental music)
- AVI: Audio-video interleaved (IBM Ultimotion and Intel Indeo)
- Conversion applet for some additional file formats included (AVC, VOC, DIB, DMP, ADPCM, M-Motion)
- High degree of modularity permits addition of more file formats (e.g. FLI/FLC)

Principle REXX Features

- *Multimedia with REXX* help file
- PLAY.CMD
- RECORD.CMD
- Entire MCI (media control interface) command set available, not just subset described in online help
- External function library (MCIAPI.DLL) provides access to MCI command set

Key Limitations

- REXX is unable to deal with loss of device (meaning ACQUIRE EXCLUSIVE must be used)
- REXX program should not hold device exclusive for long
- Unless using PMREXX (or one of the visual REXX builders), MCI commands which require Presentation Manager (such as Ultimotion playback) will fail
- REXX does not receive PM messages (to easily monitor the status of playback and devices)
- MCI's implicit opens are assumed shareable (and not necessarily desired with REXX)

Sample REXX Script

```
/* Load and initialize Multimedia REXX support */
call RXFUNCADD 'mciRxInit','MCIAPI','mciRxInit'
call mciRxInit

/* Open default digital audio device, exclusive use */
rc = mciRxSendString('open waveaudio alias wave wait','RetStr','0','0')

/* Check error, call function to return error string */
if rc <> 0 then
do
```

```
   MacRC = mciRxGetErrorString(rc,'ErrStVar')
   say 'rc =' rc ', ErrStVar =' ErrStVar
end

/* Load a digital audio file */
rc = mciRxSendString('load wave sample.wav wait','RetStr','0','0')

/* Obtain ID for device context that was just opened */
DevID = mciRxGetDeviceID(wave)

say 'DevID =' DevID

/* Set the time format to milliseconds */
call mciRxSendString 'set wave time format ms','RetStr','0','0'

/* Determine whether the microphone connection enable */
call mciRxSendString 'connector wave query type microphone
   wait','RetStr','0','0'

say 'connector query microphone:  RetStr =' RetStr

/* Query length of the opened file, value in ms */
call mciRxSendString 'status wave length wait','RetStr','0','0'

say 'status wave length:  RetStr =' RetStr

/* Play the multimedia file, wait for completion */
call mciRxSendString 'play wave wait','RetStr','0','0'

/* "Rewind" to the beginning of the file */
call mciRxSendString 'seek wave to start wait','RetStr','0','0'

/* Close the device context */
call mciRxSendString 'close wave','RetStr','0','0'

/* Ensure proper termination of Multimedia REXX */
call mciRxExit

exit(0)
```

Live Demos

- PLAY.CMD
- FOR..DO
- Ultimotion
- RECORD.CMD
- Modifications to the sample .CMD files
- Dial 1.1 by Helge Hafting

Converting MVS/JCL to REXX/TSO

Hobart Spitz
MTA New York City Transit

Converting MVS/JCL to REXX/TSO

Presented at REXX Symposium

May '94

Hobart Spitz

MTA New York City Transit

718-694-3112

5520808@MCIMail.com

Converting MVS/JCL to REXX/TSO

Abstract:

The speaker will discuss his experiences in using REXX/TSO in place of MVS/JCL. The advantages of REXX over JCL will be covered, as will a step-by-step methodology for converting existing JCL to REXX for batch and/or interactive use. JCL to REXX/TSO equivalents will be spelled out in detail. Guidelines and techniques for portability positioning to VM, OS/2, etc. will be reviewed.

Speaker:

Hobart Spitz (SBW)
MTA New York City Transit
130 Livingston St. 5041A
Brooklyn NY 11202

Phone: 718-694-3112
Alternate Voice Mail: 718-694-1719
Fax: 718-694-4309
E-mail (internet): 5520808@MCIMail.com

Application Backgrounds

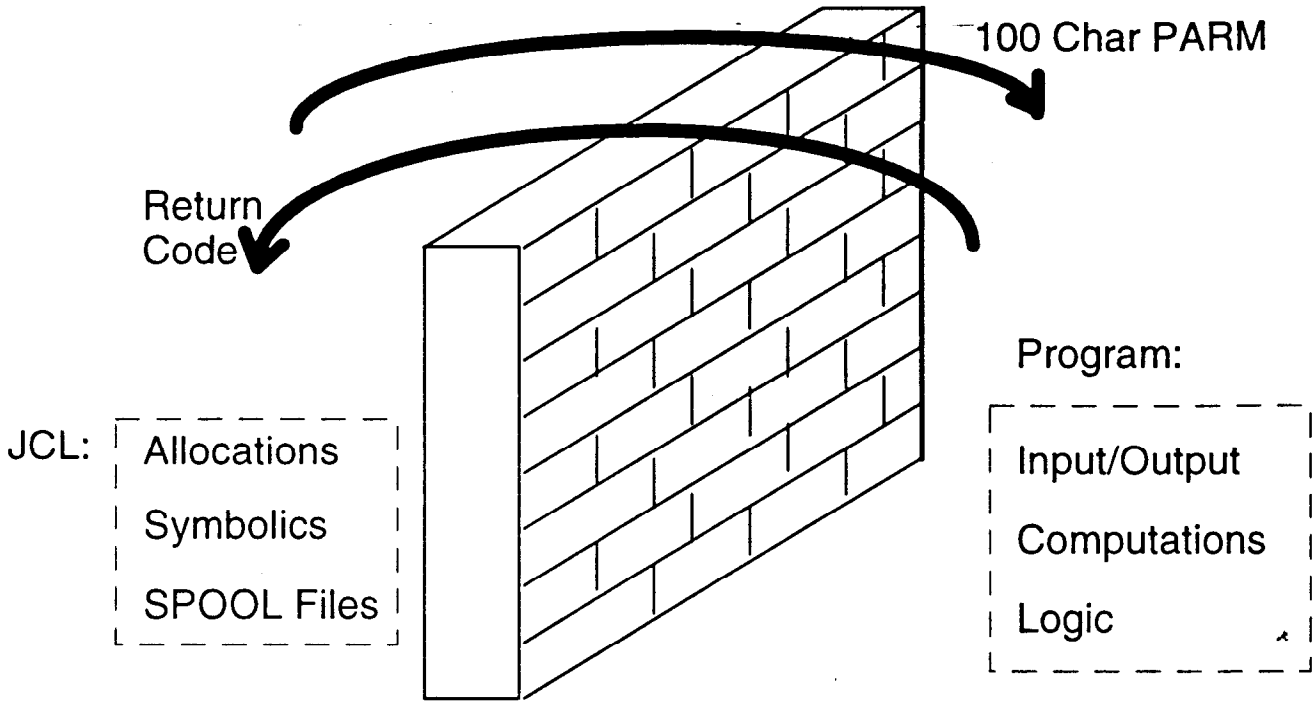
NYNEX Computer Services - Billing Service Bureau - Multiple Clients

- DB2 with 3rd party host command interface
- CICS Transaction Processing
- TSO Scheduler Access
- Limited TSO User Access
- Original Batch Design:
  JCL, COBOL II.
- Final Application:
  REXX, JCL, COBOL II.

New York City Transit Authority - Change Control Managment System

- Data stored in VSAM file and ISPF tables
- Entirely TSO Based user access
- Original Batch Design:
  ISPF Skels, JCL, COBOL.
- Final Application:
  REXX, ISPF Skels, JCL, COBOL.

"The Great Wall of MVS Batch"



Limitations of MVS/JCL

- Rigid isolation between JCL level (allocation, SPOOL datasets, symbolics, and return codes) and program level (I/O, computation, and logic).
- No interaction between application data and application control.
- Limit of 100 characters in PARM=.
- Minimal logical operators, even with new MVS features.
- Heavy manual intervention requirements in most cases.
- Single level PROC invocation, until recently.

Advantages and Benefits of REXX/TSO Over JCL

- Automation
- Simplification
- Readability, Write-ability, and Maintainability
- Modularity
- Environment dependant code can be isolated. Portability and Reusability is feasible between Foreground TSO, Batch, VM, MS-DOS, OS/2, Windows, and, maybe someday, CICS.
- Flexibility/Control Structures - Looping, General Conditionals, Expressions, PARSEing
- Controlled Recovery and Restart
- Up-front Validation and Handling of Clerical Errors
- Addresses Batch Window Criticality
- Reduced Programming Requirements
- Almost Unlimited Procedure Invocations Levels, Including Recursion
- Application data and control can interact
- Avoids MVS Steps per JOB limit.

In short, every // costs you time and your installation money.

Conversion Steps

- Extract each JCL step to REXX EXEC by PGM= and PROC name.
- Change DDs to equivalent "ALLOCATE . . .":
  - KEYWORD=VALUE becomes KEYWORD(VALUE).
  - Subparameter, KEYWORD=(SUBPARM=VALUE), becomes parameter, SUBPARM(VALUE).
  - Add quotes around permanent dataset names.
- Move each // EXEC PGM= to end of its step.
- After EXEC PGM= effect normal disposition:
  IF RC = 0 THEN "FREE DD(SYSUT2. . .) CATALOG"
- Change EXECs to equivalents.
- Drop IEFBR14 DELETE/ALLOCATE; use SYSDSN().
- Create JOB stream to invoke converted REXX module.

EXEC Equivalents

JCL		REXX/TSO Allocate Parameters
// EXEC PGM=ppp,[PARM=xxx]		ADDRESS ISPEXEC "SELECT PGM(ppp) [PARM(xxx)]"
// EXEC [PROC=]mmm,kwd=val		ADDRESS ISPEXEC "SELECT CMD(%mmm kwd=val . . .)"
	or	ADDRESS TSO "%mmm kwd=val . . . "
	or	CALL mmm kwd=val . . . (mmm has also been converted to REXX/TSO.)
// PROC kwd=val		ARG "kwd="kwd . if kwd = "" then kwd = "val"
	or	ARG kwd1 kwd2 kwd3
// EXEC . . . ,COND=(0,NE)		IF RC = 0 THEN . . . after commands. IF RESULT = 0 THEN . . . after REXX CALL. Save RC/RESULT for complex or deferred tests.

DD Dataset Parameter Equivalents

JCL	REXX/TSO Allocate Parameters
DSNAME=q1.q2.q3 DSN=q1.q2.q3	DSNAME('q1.q2.q3')
DISP=(OLD,KEEP,DELETE)	OLD DELETE (no wait, see SOMVSE93039) . . . IF RC = 0 THEN "FREE DDNAME(...) KEEP"
DCB=(model.dsn,BLKSIZE=bbb, LRECL=lll,RECFM=abc)	LIKE('model.dsn') BLKSIZE(bbb) LRECL(lll) RECFM(a b c)
VOL=SER=(vvvvvv,volcount)	VOLUME(vvvvvv) MAXVOL(volcount)
LABEL=(n,ll,EXPDT=yyddd)	POSITION(n) LABEL(ll) EXPTD(yyddd) (ddd = 0 not valid; IBM future direction.)
UNIT=(uuuu,n)	UNIT(uuuu) UCOUNT(n)
SPACE=(CYL,(pp,ss,dd),RLSE) SPACE=(800,(pp,ss),,ROUND)	CYL SPACE(pp ss) DIR(dd) RELEASE BLOCK(800) SPACE(pp ss) ROUND

Other DD Equivalents

JCL	REXX/TSO Allocate Parameters
Label on DD	DDNAME(...)
Concatenated DD	DSN('q1.q2.q3' 'q4.q5.q6')
Repeated DD across steps	REUSE required in absense of FREE.
DD * or DD DATA	"ALLOC DD(dddd) UNIT(VIO)", "TRACK SPACE(1 1)", "RECFM(F B) LRECL(80) BLKSIZE(4000)" QUEUE "information" QUEUE "more information" QUEUE "" "EXECIO * DISKW dddd (FINI"
SYSOUT=c SYSOUT=*	SYSOUT(c), c ^= *. DSN(*) (output goes to SYSTSPRT)
DEST=rmt	DEST(rmt)
HOLD=YES/NO	HOLD/NOHOLD
COPIES=n	COPIES(n)
FORMS=ltrh	FORMS(ltrh)
OUTPUT=opnam	OUTDES(opnam)
DUMMY	DUMMY
JOBLIB, STEPLIB, ISPLLIB // OUTPUT, SUBSYS=	retain - no direct eqvuivalents.

Example - Original JCL

```
//SHARE81C   JOB     (...),...
//*
//TAP2DSK    PROC    MEM=
//      EXEC  PGM=IEFBR14
//XXX     DD  DISP=(MOD,DELETE),
//            UNIT=SYSDA,
//            SPACE=(TRK,1),
//            DSN=NCSCB40.OUTPUT.TEXT
//*
//      EXEC  PGM=IEBGENER
//SYSUT1        DD    DISP=OLD,
//            VOL=SER=C12345,UNIT=TAPE,
//            LABEL=(3,BLP,EXPTD=98000),
//            DCB=(RECFM=FB,LRECL=82,BLKSIZE=8200,
//            OPTCD=Q),
//            DSN=TAPE.INPUT
//SYSUT2        DD    DISP=(,CATLG,DELETE),UNIT=SYSDA,
//            SPACE=(CYL,(2,5,10),RLSE),
//            DCB=(RECFM=VB,LRECL=100,BLKSIZE=10000),
//            DSN=NCSCB40.OUTPUT.TEXT(&MEM)
//SYSPRINT DD SYSOUT=*
//SYSIN   DD  DUMMY
//      PEND
//*
//      EXEC TAP2DSK,MEM=D930722
```

Example - REXX Equivalent

```
//SHARE81C  JOB (...)...
//         EXEC TDCMUTR1,CMD='%TAPTODSK'     (batch "logon" proc)
```

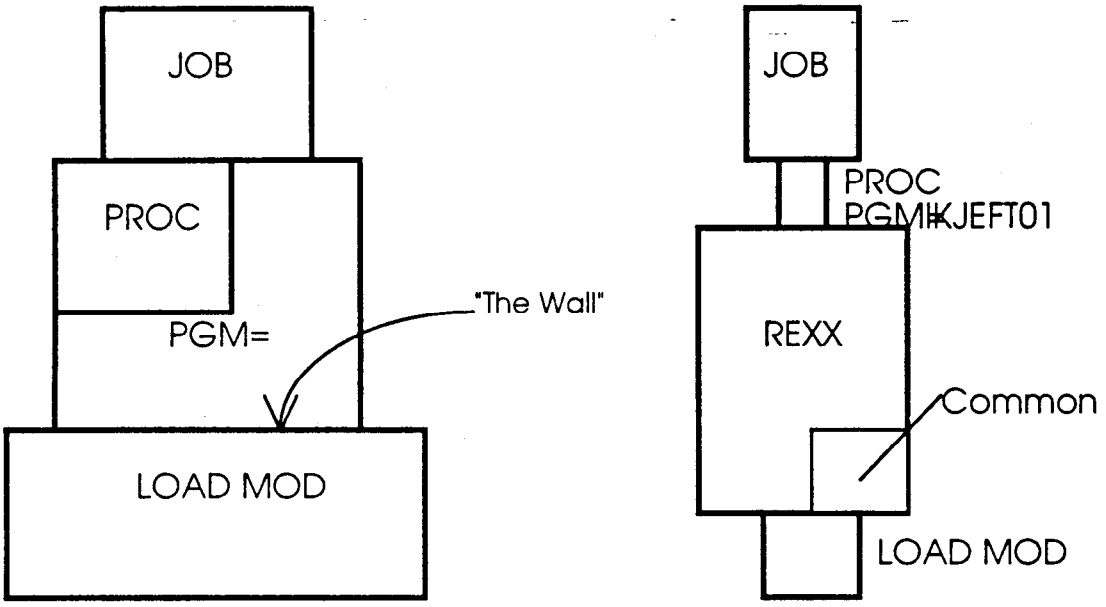
TAPTODSK

```
/* REXX */
IF SYSDSN("'NCSCB40.OUTPUT.TEXT'") = "OK" THEN DO
    "DELETE OUTPUT.TEXT"
    END

SIGNAL ON ERROR
"ALLOCATE REUSE DDNAME(SYSUT1) OLD UNIT(TAPE)",
  "VOL(C12345) POSITION(3) LABEL(BLP) EXPDT(98001)",
  "RECFM(F B) LRECL(82) BLKSIZE(8200) OPTCD(Q)",
  "DSN('TAPE.INPUT')"
"ALLOCATE REUSE DDNAME(SYSUT2) NEW DELETE",
  "UNIT(SYSDA) CYL SPACE(2,5) DIR(10) RELEASE",
  "RECFM(V B) LRECL(100) BLKSIZE(10000)",
  "DSN('NCSCB40.OUTPUT.TEXT(D"RIGHT(DATE("S"),6)"'))"
SIGNAL OFF ERROR
"GENER"
RETURN RC
```

GENER

```
/* REXX */
/* IEBGENER DRIVER */
SIGNAL ON ERROR
"ALLOCATE REUSE DDNAME(SYSPRINT) DSN(*)"
"ALLOCATE REUSE DDNAME(SYSIN)    DUMMY"
SIGNAL OFF ERROR
ADDRESS ISPEXEC "SELECT PGM(IEBGENER)"
CONDCODE = RC
IF RC = 0 THEN "FREE DDNAME(SYSUT2) CATALOG"
RETURN CONDCODE
```

Code Volume Perspective



Classic REXX

Converted System:

- Fewer JOB Streams
- Near Elimination of JCL
- Reduction of Compiled Language Application Code
- Increased Modularity
- Facilitates isolation of host dependant code and creates portability.

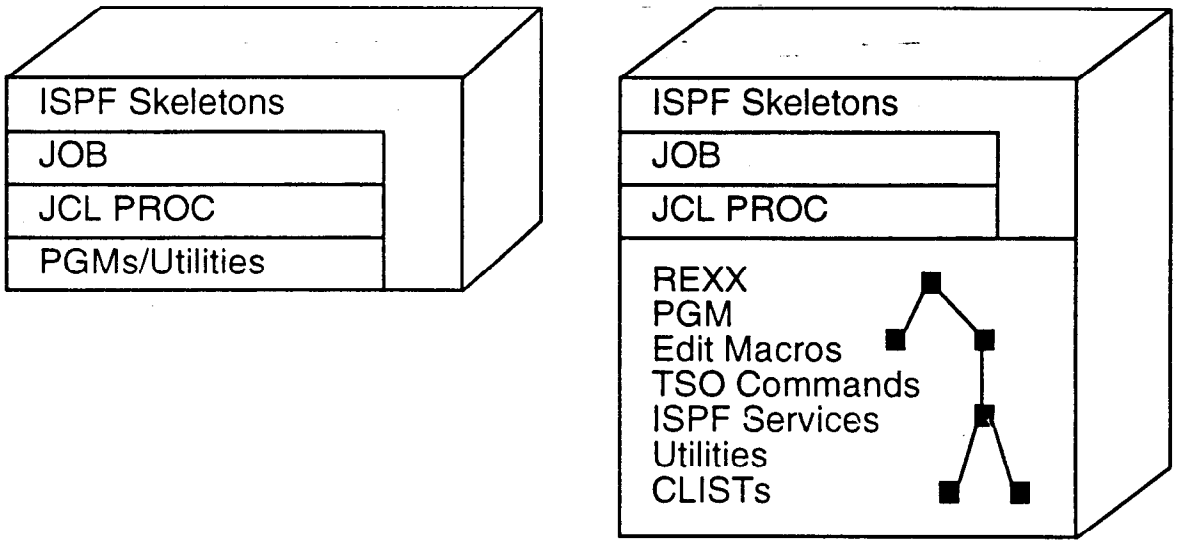
Analysis of Actual Batch System

NYCTA's CCM Release 2.1 had one PROCLIB consisting of 51 members containing 191 steps, calling 19 programs. Using 1:1 as an approximate ratio for JCL statement to REXX host command ratio, these 191 steps should be replaceable by 70 (51+19) REXX EXECs of approximately the same length, one for each // EXEC. In practice, the results were much better as most of the 51 JCL PROCs were replaced by a few REXX main modules plus about 10 driver modules to call language processors (LKED, COBOL, COBOL II, CICS, etc.).

Conversion is Ideal for Initial Usage of REXX

- Nearly all JCL has direct REXX or TSO equivalents.
- Low exposure.
- Low cost.
- Process can be automated.
- High benefit.
- Probability of success is high.
- Required software is already in-house in most shops.

Batch Comparison



Batch REXX:

- Allows Integration of Software: Allocation, Utilities, Application.
- Supports Multiple Levels of Invocation and Common Modules.
- Removes Most Requirements for Manual Intervention: Overrides, Control Cards, etc.

Summary

- Brings Batch into '90s - Portably and Productively
- Provides Real Programming Constructs
- Enables Application Based Scheduling, Recovery, etc.
- Keeps Pace with PCs and Minis
- Breaks Down "Wall" Between Control and Software Functions
- Opens Exciting New Possibilities for Batch Processing

Araceli Adriano
IBM, 40-D1-02
1 East Kirkwood Blvd
Roanoke, TX 76299-0015

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

Doug Benson
Lotus
1000 Abernathy Road
Suite 1700
Atlanta, GA 30328

Eric M. Bitterman
Teachers Insurance (TIAA)
67-85 223rd. Pl
Bayside, NY 11364
212-916-6157
Fax: 212-867-9075

Tom Brawn
IBM Corporation, Dept. G79
1701 North St.
Endicott, NY 13760
607-752-5166
tombrawn@vnet.ibm.com

Alex Brodsky
S/SE
154 Hillcroft Way
Netwown, PA 18940
215-579-2537
brodsky@netaxs.com

Shu Chen
246 W. 102nd St., Ste. 4B
New York, NY 10025

Ina Chien
Putnam Investments
859 Willard St.
Quincy, MA 02169

Anders Christensen
Sintef Runit
N-7034 Trondheim
Norway

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

Mike Cowlishaw
IBM
Hursley Park
Winchester, S021 2JN
England
mfc@vnet.ibm.com

James Crosskey
IBM Corporation, Dept. G79
1701 North St.
Endicott, NY 13760
crosskey@gdlvm7.vnet.ibm.com

Cathie Burke Dager
Stanford Linear Accelerator Center
P.O. Box 4349
Stanford, CA 94309
415-926-2904
Fax: 415-926-3329
cathie@slac.stanford.edu

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

Chip Davis
7254 Pommel Dr.
Sykesville, MD 21784-5931
410-549-3596
chip@clark.net

Rafael Fessel
Ammonoosuc Tech
131 Ridge Rd.
Franconia, NH 03580
603-823-8461

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

Hal German
GTE Labs
40 Sylvan Road
Waltham, Ma 02254
617-466-2290
Fax: 617-890-9320
hhg1@gte.com

Eric Giguere
WATCOM
415 Phillip St.
Waterloo, Ontario
Canada
519-886-3700
Fax: 519-747-4971

Klaus Hansjakob
IBM Austria, VSDL
Lassallestrasse 1
A-1020 Vienna
Austria
+43-1-21145-4243
Fax: +43-1-21145-4490
hansjako@vabvm1.vnet.ibm.com

David Hergert
Textron Defense Systems
201 Lowell St, Rm 3124
Wilmington, MA 01887
508-657-2953
Fax: 508-657-2776

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

Marc Irvin
100-01 Hope St.
Stamford, CT 06906-2500
203-852-3584
Fax: 203-852-3570

Kevin Kearney
Mansfield Software Group
P.O. Box 532
Storrs, CT 06268
203-429-8402
Fax: 203-487-1185

Lee Krystek
Boole & Babbage Inc.
8000 Commerce Pky
Mt. Laurel, NJ 08054
609-778-7000
lee@boole.com

Luc Lafrance
Simware
2 Gurdwara Rd
Ottawa, Ontario, Canada K2E 1A2
613-727-1779
lafrance@simware.com

Bill Langlais
Percussion Software
222 Berkeley, St. Ste 1620
Boston, MA 02116

Linda Littleton
Pennsylvania State University
214 Computer Building
University Park, PA 16802

John Lynn
Rohm and Haas
Independence Mall West
Philadelphia, PA 19105
215-592-3000

Ray Mansell
IBM
H4-A06
30 Saw Mill River Road
Hawthorne, NY 10532
914-945-3000
Fax: 914-784-6201

Alan Matthews
Percussion Software
222 Berkeley, St. Ste 1620
Boston, MA 02116
617-267-6700
Fax: 617-266-2810

Rohan Menezes
515 W. 59th St., No. 19B
New York, NY 10019

Patrick Mueller
IBM, MS 4B-G
11000 Regency Parkway
Cary, NC 27512
919-469-7242
Fax: 919-469-6948

Donna Murphy
Putnam Investments
859 Willard St.
Quincy, MA 02169

Simon Nash
IBM UK Laboratories Ltd
Hursley Park
Winchester
Hants S021 2JN
England
nash@vnet.ibm.com

Matthew Plager
CTPS
10 Park Plaza
Suite 2150
Boston, MA 02116
617-973-7075
Fax: 617-973-8855

Joe Player
IBM
12200 Dancrest Dr.
Clarksberg, MD 20871
301-564-2022
Fax: 301-564-2580

Edmond Pruul
R.D. 1, Box 632
Afton, NY 13730

Peter Ricciardiello
Carrier Corporation
Building TR5
Carrie Parkway
Syracuse, NY 13221
315-433-4014

Sara Rogers
Mansfield Software Group
P.O. Box 532
Storrs, CT 06268
203-429-8402
Fax: 203-487-1185

Roger Root
2963 Tillinghest Trail
Raleigh, North Carolina 27653
919-846-7101
70353.2753@compuserve.com

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

Jonathan Schulman
John Hancock Mutual Life
John Hancock Place
P.O. Box 111
Boston MA 02117
usjhcpu6@IBMMAIL.com
617-572-8410

David Shriver
IBM, 40-D1-02
1 East Kirkwood Blvd
Roanoke, TX 76299-0015

Timothy Sipples
IBM Corp.
One IBM Plaza (07/SS4)
Chicago, Ill 60611
312-245-4003
usib58c5@ibmmail.com

Hobart Spitz
MTA New York City Transit
130 Livingston St, Rm 5041 A
Brooklyn, NY 11201
5520808@mcimail.com

Stan Stocker
IBM Canada
1150 Eglinton Ave East
Toronto Ont. M3C 1H7
416-448-4197
Fax: 416-448-4414
stocker@torolab2.vnet.ibm.com

David Sutter
IBM Corp
4912 Green Rd
Raleigh, North Carolina 27604
919-301-2196
Fax: 916-301-2052

Peter Szabaga
1 Madison Ave. Area 6-F
New York, NY 10010
212-578-2691
Fax: 212-578-7198

Pam Taylor
The Workstation Group
6300 N. River Road
Rosemont, IL 60018
708-696-4450
Fax: 708-696-2277
pjt@wrkgrp.com

Chuck Turco
Monsanto
800 No. Lindbergh O2J
St. Louis, MO 63167
314-694-4227
Fax: 314-694-7545

Melinda Varian
Princeton University CIT
87 Prospect Ave.
Princeton, NJ 08544
609-258-6016
melinda@pucc.princeton.edu

Heather Wassel
524-101 Benner Road
Allentown, PA 18104
215-653-8067

Tom Wassel
524-101 Benner Road
Allentown, PA 18104
215-653-8067

Robert Wilcox
New World Technologies
85 Jones Hollow Rd
Marlborough, CT 06447
203-295-0680