REXX AND UNIX PANEL DISCUSSION

JEFF LANKFORD, NORTHRUP; SAM DRAKE, IBM; JONATHAN JENKINS, AMDAHL;
SCOTT OPHOF, CONSULTANT; ALAN THEW, UNIVERSITY OF LIVERPOOL

REXX and UNIX Panel Discussion

Moderator: Jeff Lankford
Panel: Sam Drake, Jonathan Jenkins, Scott Ophof, Alan Thew

Moderator's Note: The complete session was taped, but the recordings are being withheld pending filing of formal charges. What follows is a collection of prepared notes, not necessarily as full of wit as the delivered presentations; you simply had to be there.

Introduction

Jeff Lankford

Good Afternoon, Ladies and Gentlemen...

On behalf of the participants, welcome to the panel discussion "REXX and the UNIX Environment". We are fortunate today to have panelists outstanding in their fields and who have occasionally been found out standing in other people's fields.

A common theme is addressed by all the panelists: why use REXX with UNIX — and how to do so effectively. The first speaker, Alan Thew of the University of Liverpool Computer Laboratory, will discuss "REXX and awk: how does REXX fit in with existing programming tools?", in which he compares the various interpreted languages commonly used in the UNIX environment and contrasts their capabilities with those of REXX. The next speaker, Jonathan Jenkins of Amdahl Corporation will present a practical perspective on "UNI-REXX use at Amdahl". The third panelist, Scott Ophof, formerly of Delft Hydraulics, will discuss portability concerns when using "REXX on any System". The concluding speaker, Sam Drake of IBM Research in San Jose will raise the question "REXX and UNIX ... what's the point?", and examine the issue of finding REXX's proper niche in the UNIX environment.

Before proceeding, I'd like to exercise my prerogative as moderator to discuss something completely different, by raising a perennial question that has converted many doctors of philosophy into cabbies (and vice versa): "What am I doing here?" The special case is clearly more interesting, namely: "What am I PERSONALLY doing here?"

I am neither a chronic nor habitual user of REXX; my use is best characterized as recreational. In fact, I haven't touched a piece of REXX code in nearly six months; if I stay clean a whole year, the doctors tell me I'll be cured. My first experience with REXX occurred a few years ago when I undertook a project to implement international standard networking protocols in the IBM VM environment. Entering the VM environment from a UNIX background was traumatic, partly due to the paucity of convenient-to-use program development tools.

I soon learned of REXX and found it unlike any other standard VM utility: it was easy to learn and to use, it supported rapid prototyping, it supported personal tailoring of the system command language, and the string processing functions promoted its use as a macro processor. Using REXX, in about a month I built the core of a UNIX-like program development environment that provided networked hierarchical file reference, compatible file, device and interjob I/O, asynchronous job initiation, and implementation of nearly one hundred UNIX-like commands front-ending either VM commands or custom built REXX functions. Without REXX, program development in a heterogeneous networked environment targeting applications for compilation and execution in the VM environment would have been much less productive.

In his 1984 paper published in the IBM Technical Journal, Mike Colishaw succinctly described one of the major reasons for REXX's popularity in the IBM world: "The design of REXX is such that the same language can be effectively and efficiently used for many different applications that would otherwise require the learning of several languages." While certainly true of many IBM environments, this is less true of the UNIX environment, where several stream editors, command language and macro processors offer complimentary and compatible features.

Hence, there are potential barriers to acceptance of REXX in the UNIX environment. A rudimentary classification scheme distinguishes between barriers of style and barriers of substance. The former category includes the stylistic difference between the UNIX philosophy of making each tool do one thing well, together with the anticipation that tools should interact via "piped" data streams, versus the typical VM practice. Substantive barriers include differences between external I/O models, for example between the UNIX system's three distinct data streams for input, output, and error messages versus REXX's single-threaded data buffer chains or stacks. Also, the lack of regular expression manipulation built-in functions as a standard part of the language could be considered a barrier to the acceptance of REXX. Another barrier is the rudimentary signaling and event handling mechanism. Stylistic barriers can be addressed by acculturation of REXX application programmers to the UNIX environment, but substantive barriers require innovative implementations or even extensions to the evolving standard to provide REXX program accessibility to standard, popular UNIX features. While there are many excellent reasons supporting the use of REXX in the UNIX environment, the real challenge, for you the audience, as much as for the panelists, is to seize this opportunity to cooperate in the uncovering of potential barriers and to begin to formulate reasonable solutions.

On behalf of the sponsors of this second REXX Symposium I want to thank the panelists for subjecting themselves to the mercy of the crowd and most especially to thank you, the

crowd, for making this session memorable.

REXX and AWK/ksh - Can the former learn from the later?

A. J. Thew

I am an applications programmer at the University of Liverpool Computer Laboratory, U. K. We currently run a VM/CMS service for most users but are moving to Unix and by late 1993 will base all our user service on Unix.

I have used REXX for about 5 years. At present this is primarily in conjunction with the SQL/DS RDBMS using RXSQL. REXX is used extensively in data manipulation along with CMSPIPES (a major contribution to REXX on CMS). This represents the bulk of my job. My Unix experience started about 2 years ago and now that I've passed some of the worst part of the learning curve, my main activities here are investigation of public domain (and other) tools, e.g. an e-mail for a new Unix system and editors. I have used all major shells to some degree. Some work with C has been done and familiarity with other tools is increasing. I have also used to some degree all major Unix's (BSD, SunOS, HP-UX, System V Releases 2 and 3).

I want to try to say something about some existing tools on Unix and how they might relate to REXX and vice-versa. These tools are the other interpreted programming languages. This emphasis partly reflects my job and interests at this present time.

My first reactions on seeing the shells from the programming point of view was that they seemed very primitive compared to REXX. It was as if REXX was removed from CMS and EXEC1 was the command interpreter and general programming language. No free format and plenty of chances for error.

David Korn (of AT&T) (1) and Morris Bolsky say "no unquoted spaces or tabs are allowed before the '=' or after". Poor manipulation of strings and almost no arithmetic were additional first impressions. Others coming from a CMS background around the world felt similarly but were met with unsympathetic responses such as "**sh** does all you need" from the existing Unix community. The obvious "problems" were visibly demonstrated by Neil Milsted (2) during last year's symposium.

Times change and I now feel I can use **vi** faster for some operations than XEDIT (I should point out that this was partly out of necessity). In addition I have had plenty of opportunity to look at what interactive programming tools Unix provides. This has mainly centered on AWK abut also the Bourne Shell (**sh**) and the Korn Shell (**ksh**).

My attention was grabbed by AWK since on face value it offers a concise simple syntax like C, but without data-typing, semi-colons, memory management, pointers but with real arrays, *proper* string manipulation functions, arithmetic, simple assignments (no dollar signs in most cases), and some ability to interact with Unix. Arrays are associative which seemed similar to REXX compound variables. It seemed to offer the best of C and shells without any of the pain and with functionality that I'm used to with REXX on CMS.

I have attempted to take a serious look at shells, and have recently standardised on the Bourne Shell, in particular the version dating from System V Release 3. Apart from BSD systems and older System V releases, this is reasonably widespread. It offers more compatibly with other tools, allows redirection on the read statement, more built-ins for better performance and improved parameter checking/substitution and above all functions (though *not* recursion without pain).

In addition David Korn has produced a shell that is largely compatible with the System V Release 3 **sh** but enhanced a great deal. This offers:

- much better performance than existing shells,
- greater functionality,
- more general I/O,
- built in arithmetic,
- some string functions,
- local variables in functions (i.e. recursion),
- limited array capabilities,
- co-processing features,
- and better security.

This shell is gaining in popularity. To illustrate its capabilities, Morris and Korn's book present a powerful subset of the Rand Mail message handler. This is not the "Word processor in FORTRAN" type application but a realistic project. They even claim that the **show** and **next** commands are faster than the C equivalents. However, the code is not nice to look at in my opinion (although it is probably easier to understand than the "real thing" in C) but it's very compact, being less than a 1000 lines of code and comments.

AWK derives its name from its designers Alfred Aho, Brian Kernighan and Peter Weinberger. An additional interest was that the Free Software Foundation provided a version that worked on a PC which was exactly the same apart from pipe support and was constrained by 640K. This is also available for Unix and implements all the latest functionality.

I should stress that I'm referring to what is commonly called "new AWK" or **nawk** and references to AWK will refer to **nawk** unless stated. This version has been available since 1985 and is available as standard (with the old **awk**) from most vendors.

The original language was written in 1977 and its basic action was to "scan a set of input lines in order, searching for lines which match a set of patterns which the user has specified"(3). An action (code section) can be taken when lines match a pattern (default is print). The "patterns may be more general than those in grep, and the actions allowed more involved than merely printing the line". The language was designed for ease of use rather than speed.

AWK provides implicit and explicit data input, the latter not being required for a working program and some books do not present any treatment of the explicit input until toward the end. When given a file to read as an argument AWK reads it sequentially, as records which where the record separator defaults to a newline. The original was designed for short programs of one or two lines. They designers "knew" what the language was designed for but many users, often first time computer users found that the ease of use made it a general programming language and used instead of others. This "shocked and amazed" the designers who assumed that compiled languages (presumably) would be used for anything longer than a few lines. Users often seem willing to sacrifice

performance when ease of use is available, BASIC was/is an example of this.

The AWK users had their demands met and the 1985 version contained dynamic regular expressions, new built in variables and functions, multiple input streams with more explicit I/O, and user functions. The AT&T System V Release 4 version makes some additional enhancements in the spirit of the 1985 release but not so numerous. Function libraries do not have explicit support but are easy to implement.

Examples of major applications which use **awk** are a **nroff** type text formatter (written for early versions of Unix which did not bundle nroff with them) and small Lisp interpreter.

It is interesting that Aho, Kernigan and Weinberger mention REXX in a discussion about "similar" languages (mentioning SNOBOL4 and ICON) (4). This statement was a small prod for my topic, I'll have to admit.

There are built in software limitations (4) [to AWK]:

- 1 open pipe,
- 15 open files,
- 100 fields,
- 3000 chars per input record,
- and 1024 chars per field.

These were designed in or not designed out since the performance would have degraded substantially for one thing. *But* the main thing that AWK users miss/need is that the lack of any debugging facilities. **nawk** and the Free Software Foundation's **gawk** give better error diagnostics when the program fails which is an improvement to old AWK's "bailing out..." error. Debugging has to be done on the lines of any other language without built in tracing or an available interactive debugger.

The Unix shells both provide builtin tracing, **ksh** allows command scanning without execution and something like REXX's *TRACE R* command although no interactive tracing. AWK is poor at replacing the shell command interpreter functions since it's use of pipes is restricted to either input or output (remember the one open pipe limit). It has no interrupt handling facility like the [**sh**] **trap** command, and there are other limitations but this is not "pushing" the language as much as abusing it. AWK's real strength is as a data processing language.

AWK passes arrays to functions by reference and not by value and scalars by reference. It is possible to have local variables by effectively hiding them on the function line:

function fred(a, b, loc1, loc2)

Functions can be recursive.

Users requiring extra performance can buy the **awkcc** program from AT&T which converts the program to C and then compiles it. **awkcc** does not come with any vendor versions of Unix that I know of. A company even provides a proper compiler but only for the PC.

The design goals [of REXX] were/are very different. They make REXX a bigger language than AWK, the latter is often referred to as one of Unix 's little languages. REXX was "designed for generality" (5) which makes it suitable for many tasks, one of which is a command processor:

- readability,
- no explicit data-typing,
- good diagnostics via limited span syntactic units,
- low astonishment factor (predictable results even when features accidentally misused),
- language kept small in sense of number of commands,
- and no defined size or shape limits.

A feature of REXX that has always impressed me as a user was the debugging facilities, just add the required trace command and go.

Goal was to have good performance as well. CMSPIPES as well as performance tips has enabled us to double performance of critical execs on CMS.

REXX 4.0 goals(2): Still to "keep the language small", enhancements chosen on "high power-to-complexity ratio". This last phrase sums up REXX for me in that it is small in some aspects and when interpreted slow on very large programs but otherwise without limit.

Some additional functions are required by REXX to allow it to communicate in the most basic way with the shell which exec'd it and pass commands to a shell to execute [on UNIX]. These were listed by Neil Milsted of the Workstation Group at the last Symposium (2) and provide enough, **cwd**, **getenv/putenv**, etc.

One missing feature is currently regular expression support, a feature of many Unix tools which do some pattern matching. This allows the shells to have some ability to manipulate strings without any inherent string functions. This "lack" is not so apparently bad when one examines the wealth of functions available with REXX, many more than AWK. It was an interesting exercise to see which functions could be implemented as user functions in AWK. Only *JUSTIFY* and *VERIFY* looked hard. REXX's parser is more generalised than AWK's but AWK's use of a simple user definable field separator which itself can be a regular expression should not be underrated. AWK has math functions and substitution functions (type of line edit) that REXX does not have although only the math functions would need a function package to get good performance.

Some shell features are missing such as interrupt handling and some of the advanced features of the **ksh** but these could be provided possibly by function packages without making the core language on Unix non-standard and the manuals twice the size. REXX is easily the language to replace some shell programs and become a major command language for for Unix. However, if REXX becomes *just* another language for scripts that would seem rather limiting since it was designed to be a general purpose language. C and **ksh** (where available) can do a better job in some cases, although a REXX compiler for Unix would be very interesting.

The standard should prevent REXX becoming a monolithic Unix tool which could be tempting but dangerous in my opinion for the above reasons and most importantly would go against proven design objectives. Unix already has a public domain tool which possibly provides this monolithic functionality in Larry Wall's PERL but his design goals were different.

AWK is a programming language in its own right, even taught on some software engineering courses apparently. It was designed to fit in with the Unix philosophy of being a tool to do a job, a tool of many. This philosophy lets other languages do other jobs such as sorting, or better handling of command line arguments (shell wrappers). While REXX

is better suited to cover more ground than AWK and "who wants to learn many programming languages when they can learn one?", REXX could be seen as providing more/better facilities rather than just a replacement language which ignores as much of Unix as possible and re-invents the wheel many times.

References:

(1) Bolsky, M. I. and D. G. Korn. *The Korn Shell - Command and Programming Language*, Prentice-Hall, 1989.

(2) *Proceedings of the 1st REXX Symposium for Developers and Users*, SLAC, 1990.

(3) Aho, A. V., B. W. Kernighan and P. J. Weinburger. *AWK - A Pattern Scanning and Programming Language*, Unix Programmer's Manual, Vol. 2, 1978.

(4) Aho, A. V., B. W. Kernighan and P. J. Weinburger. *The AWK Programming Language*, Addison-Wesley, 1988.

(5) Colishaw, M. F. *The REXX language A practical Approach to programming*, Prentice-Hall, First Edition, 1985.

Uses of REXX under Unix at Amdahl's Corporate Computer Center

J. L. Jenkins

Hi! My name is Jonathon Jenkins and I work for Amdahl Corporation. I'd like to talk to you today about some of the practical applications that we created under our Unix systems using REXX. Please feel free to contact me if you have further questions, or would like to see copies of this code.

Currently 43 uses have been determined for REXX under Unix. Categories include:

- System Monitoring
  - Check for the completion of system accounting
  - Dump Management
  - Checking console logs for system and device errors
  - Daily cleanup of temporary filesystems
  - Continuous montoring of permanent filesystem usage
- Security
  - Checking for unauthorized superuser access
  - Checking for incorrect users in the /etc/passwd file
  - Checking for unauthorized members in system groups
- Disaster Recovery
  - Backing up of critical system files to root and /usr

When I attended the 1990 REXX Symposium, I learned of a product, called Uni-REXX that is a REXX interpreter for Unix systems. After returning to work I recommended that we purchase this product. I was asked to provide a justification as to why we needed this product and how we could use it on our UTS systems at the Corporate Computer Center. I began to make a list of ways to use REXX under UTS. Since that time, the list has grown to contain 43 separate items, some of which I have begun to write. These items seem to fall into three separate categories.

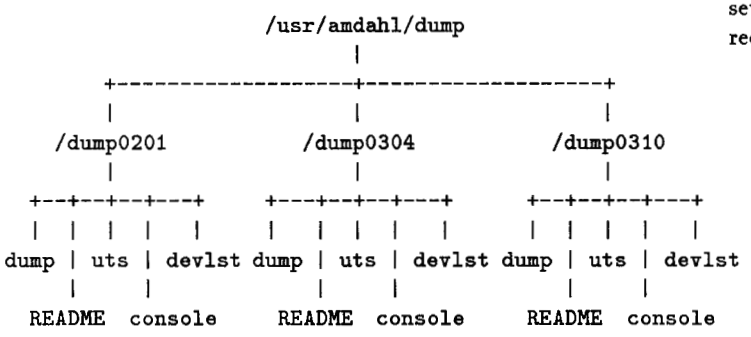
System Monitoring

Periodic checking of the system. Items which fall into this category are:

- Checking the status of System Accounting processing. System accounting generates usage reports on UTS each night. Sometimes the processing software encounters unrecoverable errors. Stewards should check daily to ensure that all of the accounting data from the previous day has been processed. If this is not done, critical charge-back data is not processed to bill CCS UTS customers.
- Check for successful completion of accounting. This can be automatically checked by a program which notifies the system steward only if problems were detected.
- Dump Management:
  The /dump and /usr/amdahl/dump filesystems are used to hold the current dump file and previous system dumps. When a UTS system panics (abends), an image of system storage is written into a file named dump in the /dump directory. Due to the storage sizes of some of the UTS systems, this filesystem can typically hold only one dump at a time. Previous system dumps are copied into the /usr/amdahl/dump filesystem for examination. Here are some of the things that may be checked automatically using REXX are:
  - Make sure that a dump file is on this directory. Create one using the *makedump(1m)* command if dump not found.
  - Make sure that the dump file is the only file in this directory. All other files associated with dumps should be placed in the /usr/amdahl/dump directory. Files found which are not associated with dumps should be removed.
  - Check the %full (blocks and inodes) of the filesystems.
  - Ensure that the following directory scheme is adhered to for each of the files found in the /usr/amdahl/dump directory: a README file that contains a description of why the system was down, the dump file, a copy of the related kernel (/uts at the time of the dump), a copy of the console log (/usr/spool/console/<dump_date>), a copy of the /etc/devicelist.

Following is an example of how the directory structure underneath /usr/amdahl/cump could be organized. Note that the "0304" in the "/dump0304" represents the date of the dump 03/04 of the current year.

- Checking console logs for system and device errors. The system console log contains information about current and significant events on the UTS system. Sometimes it contains sensitive information such as passwords, administrative commands, and system operation information which is not suitable for clients. Some of the important pieces of information contained in the console process stack error messages, unsuccessful logon attempts.



- Unsuccessful login attempts
- Device Data Checks, Equipment Checks, and Unit Checks
- Line timeouts and restarts (PVM, 3274e)
- Missing Interrupts
- Process error messages (ie. Stack too Large)
- Tape mount request information

- Following are some the things which may be checked automatically and responded to/reported on via programs.
  - Permissions, owners, and groups of files in the */usr/spool/* directory. These files/directories may be checked daily and may correct the problem as well as report it to the system steward.
  - System Error Messages such as those listed below, may be collected and delivered to the appropriate groups. Note that the possible groups are listed after each message: Data Checks (UTS/SSS), Unit Checks (UTS/SSS), Equipment Checks (UTS/SSS), Unsuccessful logon attempts (Computer Security), RSCS shutdown/restarts (UTS/TSG), PVM shutdown/restarts (UTS/TSG), Line timeouts (UTS/TSG), Process stack too large (UTS/TSG), Out of paging/swap space (UTS/TSG, UTS/SSS), Ethernet network unavailable (UTS/TSG)
  - The console log contains information about tape mounts. It shows when a mount request was received, when the request was satisfied, and when the tape user completed use of the tape. From this information, we can generate the following types of reports: How many tapes were requested for the day, How long (including average times) tapes were mounted, How many tapes were mounted over 3 shifts (grave, day, swing), How long it took to satisfy mount requests (including average mount times), Flagging of tape mounts which take longer than a predetermined threshold (currently 10 min.).
- Daily cleanup of temporary filesystems like /tmp, /usr/tmp, and /free. These directories are used to hold temporary information on our UTS systems. All users are able write to these directories, and don't always remove their temporary files when they are done. Because of this, these directories run out of space. Following are several things to be checked daily, concerning these directories:
  - Remove all files and directories older than a predetermined amount of time.
  - Warn system stewards and operators when %full (blocks and inodes) is greater than a predetermined threshold and again at 90%.
  - When 90% full (blocks and inodes) and files greater than 3 days old have been removed, remove those over 2 days and then those over 1 day old. If it is still full, then start removing files in reverse time order. (**ls -lt** ... older files first)
- Continuous montoring of permanent filesystem usage. From time to time (at least one per day), filesystems on UTS run out of blocks or inodes. The kernel only places a message on the console once the filesystem is full. Through the use of programs and the **df(1m)** command, we can continually monitor the usage of filesystems and alert the appropriate personnel when they began to become full.

Security

- Checking for unauthorized superuser access. On UTS, the super-user account has complete authority over the system. This user can read or write any file on the system, it can change anyone's password without security restrictions, it can kill any process on the system, modify the kernel and system source code, and write directly to any device on the system. Each of these priviledges is something that should only be available to a select number of system users, therefore access to super-user should be monitored daily to make sure that only authorized users have this ability. Information about superuser access is logged by the system. It is possible to check this log for unauthorized accesses and unsuccessful attempts. This information can be delivered daily to the system stewards for action.
- Checking for incorrect users in the /etc/passwd file. The /etc/passwd file contains a list of the users who are valid to UTS. This file should be checked for inconsistancies and potential security holes.
  - Find users with home directories and .login, .cshrc, and .profiles that are accessible to others. Notify these users of the problem, and of the potential problems of having these files open to others. Change these settings after two weeks/month of notification.
  - Make sure that expired users have a login shell of /usr/dirm/bin/bye
  - Check users no passwords.
- Checking for unauthorized members in system groups. The group permissions of files play a large part in determining who can access them. It is important that the permissions of these files are set correctly and that the members of certain groups are checked regularly.
  - Valid system groups and users kept in the control file.

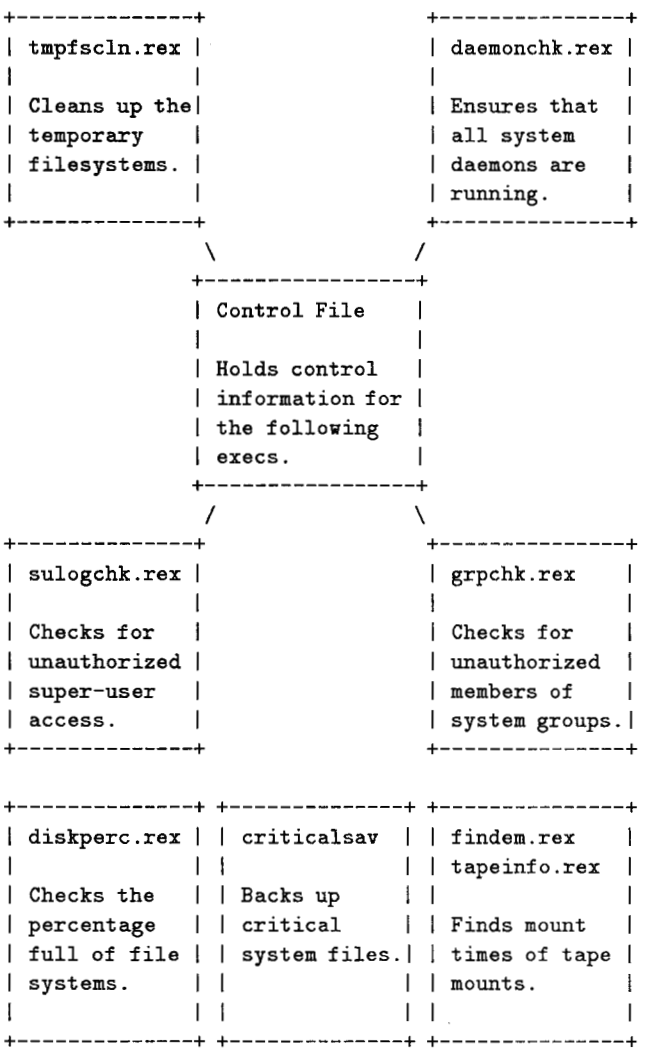
– Groups to check are: bin, mail, adm, oper, sys, tape, dev, uidadm

Disaster Recovery

Backing up of critical system files to root and /usr Due to the importance of some system files, it is necessary to have a backup online in case the file is inadvertantly changed or destroyed. It is possible to have this function done via a program which runs daily. These files should be backed up onto two separate disk volumes to minimize the chances of both copies being destroyed.

- Should be kept backed up both on root and usr volumes. (/.critsave and /usr/dirm/tsg/.critsave)
- The files to be backed up are as follows: /etc/devicelist, /etc/passwd, /etc/group, /etc/identity, /etc/identitydefs, /etc/inittab, /etc/hosts, /etc/services, /etc/netinfo.db, /etc/netstart, /etc/profile, /etc/rc, /etc/bcheckrc, /etc/brc, /etc/local.rc, /usr/spool/cron/crontabs/root
- These should be written to tape weekly.

Examples



The next example demonstrates how the system is put together. These routines are called from the **cron(1m)** command. They are executed at regular intervals and typically produce exception reports. I have included a copy of the source code of the *daemonchk.rex* program which checks to make sure that all of the system daemons listed in the control file are actually running on the system. I have also included the lines from the control file which this exec uses to initialize the daemonlist variable as well as a copy of the *crontab* file used to automatically execute these execs. This exec demonstrates the power of using the functions of Unix along with the power of the REXX language.

Sample crontab file:

```
#
# min     hour    day     month   day_of_week  command
#
#(0-59)   (0-23)  (1-31)  (1-12)  (0-6
#                                 Sunday=0)
#
#-------------------------------------------------
#
# System Steward
#-------------------------------------------------
#
# -> Check disk full percentages and notify contacts.
0 6,16 * * 1-5 rexx /autoops/stewards/diskperc
#
# -> Check disk full percentage every hour during day
0 7-14 * * 1-5 rexx /autoops/stewards/diskperc
0 16-18 * * 1-5 rexx /autoops/stewards/diskperc
#
# Check disk full percent every hour during off shift
0 20,22 * * 1-5 rexx /autoops/stewards/diskperc
0 0,2,4 * * 1-5 rexx /autoops/stewards/diskperc
#
# -> Check to make sure that system daemons are running
# -> every 5 min. during the day.
0,5,10,15 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
20,25,30 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
35,40,45 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
50,55 6-18 * * 1-5 rexx /autoops/stewards/daemonchk
#
# -> Check to make sure that system daemons are running
# -> every 15 min. during off shift.
0,15,30,45 19-5 * * 1-5 rexx /autoops/stewards/daemonchk
#
# Automated Operations
#-------------------------------------------------
0 0 * * 0-6 rexx /autoops/tpmnts/findem
#
# Security
#-------------------------------------------------
31 2 * * 1 rexx /autoops/.security/passwdchk mail nomsg
35 0 * * 1-5 rexx /autoops/.security/criticalsav
0 5 * * 0-6 rexx  /autoops/.security/grpchk
#0 5 * * 0-6 rexx /autoops/.security/tmpfscln
#0 3 * * 0,3,5 rexx /autoops/.security/sulogchk
```

Control file lines:

```
steward jlj50
daemon <swap> <init> <wss_dmn> <wss_steal> <flush>
```

```
daemon <sys_0> <sys_1> <sys_2> <sys_3> osmcat
daemon cron nadaemon admin802 tpdaemon
daemon lpsched reread spls tdmr dbspvsr
daemon portmap
daemon adminllc ipadmin tacomad biod
daemon mountd nfsd sendmail inetd sts
daemon slink mr dbcopyd rscsd
daemon errdemon
```

DAEMONCHK.REX Program Code:

```
/* REXX ***************************************
*                                             *
*  Name: daemonchk                            *
*  Date: 02/04/91                             *
*  Time: 03:59:40                             *
*  Auth: Jonathon Jenkins                     *
*                                             *
*  This exec will check to make sure that all *
*  of the system daemons listed in the control *
*  file are running on the system.  It will   *
*  print them if more than 5 copies are running *
*                                             *
***********************************************
*            Change History                   *
*------------+--------------------------------*
*    Date    |  Description of Changes        *
*------------+--------------------------------*
*                                             *
*                                             *
*                                             *
***********************************************/

address unix
arg .

found.=0
daemonlist=''
control_file='/autoops/control_file'

do queued(); pull; end
x=popen('/bin/grep system_daemons 'control_file)
do while queued()>0
   parse pull keyword . 1 rest_of_line
   if keyword='system_daemons' then do
      parse value rest_of_line with . daemons
      daemonlist=daemonlist daemons
      end
end

do queued(); pull; end
x=popen('/bin/ps -e | /bin/grep -v getty')
do while queued()>0
   parse pull . . . daemon
   daemon=translate(daemon,'_',' ')
   if wordpos(daemon,daemonlist)/=0 then
      found.daemon=found.daemon+1
end

do count=1 to words(daemonlist)
   daemon=word(daemonlist,count)
   if found.daemon=0 then
      say 'daemonchk: 'daemon' was not found',
          ' running on the system.'
   if found.daemon>25 then
      say 'daemonchk: 'found.daemon daemon' were',
          ' found to be running in the system.'
end

exit
```

My management is currently planning to purchase the Unix-REXX product from Wrk/Grp pending the availability of funds. I used a copy of the product that was used to port the interpreter over to UTS to test this code.

REXX on any system

F. S. Ophof

I'm basically a CMS user on the systems maintenance side, trying to find out what UN*X means (kicking and screaming all the way...).

My introduction to REXX was on an IBM 4331 running CMS a few months after starting to program in EXEC2. This was at DELFT HYDRAULICS, my employer at the time. EXEC2 did not really seem an attractive one to write XEDIT macros in, so I took to REXX like a fish to water and never looked back.

Personal REXX and Kedit made the PC a more attractive tool for the CMS users. This led to problems porting applications from CMS to the PC and more people were using *both* versions of REXX, one on the mainframe, the other on their PC.

A new policy at DELFT HYDRAULICS dictated that VMS, CMS, NOS-VE, and PCs were to be replaced by UN*X where possible. The CMS users wouldn't really be happy with **vi** (which they spelled Y-U-K). So from the user support point of view I looked around for a UN*X version of REXX and XEDIT, mainly via e-mail.

It was Alan Thew who mentioned uni-REXX and uni-XEDIT. My hope was the implementors hadn't followed the "include everything but the kitchen-sink" philosophy. Well, the Workstation Group was distributing a very clean, CMS-like version of REXX and XEDIT. Cleaner than I first thought. But it wasn't available yet for the HP 9000 series 800, the machine in use at DELFT HYDRAULICS.

In the meantime Alan and I were discussing REXX (and XEDIT) on UN*X with Ed Spire. My main contention was, and still very much is, that what functionality doesn't belong in a program should not be included. This is in line with the UN*X philosophy of making each tool do its own thing well. I would like to add "and *only* its own thing".

Uni-REXX and uni-XEDIT for the 800 model arrived, were installed at DELFT HYDRAULICS, and have been in use a couple of months. The last I heard is that they were very happy with these UN*X versions.

REXX on any system — What a wonderful idea.

REXX has been implemented on a respectable number of operating systems. And there is no problem using REXX, as long as programs built for a specific operating system stay there.

But when interoperability is needed...

Each operating system has its own peculiarities. So each REXX implementation needs to be adapted to that environment. The result is that no two implementations are identical. The main differences are in:

- I/O models (byte streams on UN*X, records on CMS),
- file specification (*/dir/subdir/part1.part2.etc* on UN*X, *FILENAME FILETYPE FMD* on CMS),
- and operating system philosophy (filters in UN*X, *eierlegende MilchSau* in CMS, "hog all memory and do it *my* way" in DOS).

So when an application is copied from one operating system to another, it comes down to virtually rewriting the whole thing.

This is not my idea of "REXX on any system"...

To be able to have interoperability apply to REXX, the following might need to be done:

- Make REXX programs completely independent of operating systems.
- Modify REXX so it can recognize for which opsys the program was originally written and interpret accordingly.

Opsys independency — Beautiful! Can it be done? If so, what is needed? Externalize the I/O model? Modify the I/O model so the syntax is valid for any conceivable system?

Would a complete rewrite of REXX be necessary? Or could it be done with a number of additions/extensions? What about current users of REXX?

A lot of questions...

Recognize the opsys: this *could* be done with an external set of functions and procedures. Con: Each new implementation of REXX would create the need for new sets of translators (being twice the number of already existent implementations). Pro: One would only need the translation set(s) for those opsyses on expects to translate from.

Any change to REXX itself to achieve this would of course need to be independent of implementation, since one cannot expect the user to buy a new version of REXX for each new implementation to become available.

A logical extension would be to create a "neutral" set of functions and procedures to bring down the number of translations sets. And so we are just about back at the first possibility (independency of opsys).

As to the Uni-REXX implementation, it's quite "clean" (in uni-REXX the **cd** function is necessary). Some of the other implementations could use a bit of clean-up as to modularity.

Statements and functions which are not REXX specific should be relegated to external programs [or] function packages.

The manuals should state clearly which are standard REXX statements/functions, which are implementation dependent, and which are add-ons the implementor offers.

Examples:

- DIAG() in the CMS implementation.
- The whole hardware and DOS groups of functions in Personal REXX.

Add-on due to presentations already here:

- UN*X has multi-tasking. How does this affect the SIGNAL statement? Would SIGNAL need to be enhanced for UN*X? And, how does this affect interoperability? Would implementing the enhancements in UN*X (even as NOP) be a good idea to copy to other implementations?
- Replacing, inserting, deleting a line within a CMS file is very easy without destroying the rest of the file. But using LINEOUT() looses everything after the last line worked on. My reaction was major panic. So on the PC I use EXECIO, not the REXX I/O facilities.
- Since regular expressions are dependent on the operating system, why include it in REXX? It's not part of REXX itself.

REXX on UNIX ... what's the point?

S. Drake

Moderator's note: Sam's well-groomed slides appear separately in these proceedings.

Moderator's Note: The floor was opened for audience comment, and a lively discussion ensued. A splendid time was had by all.

REXX in UNIX

What's the Point?

Sam Drake

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

BITNET: DRAKE at ALMADEN
Internet: drake@almaden.ibm.com

REXX Symposium

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

My Predjudices

- Former "VM Bigot"
  - Used REXX as programming language, macro language under XEDIT and other programs
  - Couldn't survive without it
- Now an "AIX Bigot"
  - Tried to make a "clean break"
  - After four years, I still can't write a shell script
  - And I'm darned proud of it

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

UNIX state-of-the-art

- Two "classic" shell script languages
  - Bourne Shell, C-Shell
- One "classic" data manipulation language
  - AWK
- Two "modern" languages
  - Korn shell
  - Perl
- No unified macro languages

All are powerful, cryptic, unfriendly

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

Korn shell example

```
case $1 in
1)      # keep current dir
   print -r - "$PWD"
   return
   ;;
[2-9]|[1-9][0-9])
   n=x+${1}-1 type=2
   if ((type<3))
   then x=4
   fi
   ;;
*)
   #default
   ;;
esac
```


IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Perl

- Relatively new language
- By Larry Wall
- Implementation, documentation publically available
- Combines:
  - Good interpreted shell script language
  - Good data manipulation language
  - Excellent access to native UNIX facilities

I can think in REXX and write PERL!!!!!!!!

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

Perl Example (in REXXish Style)

```
$name = "";
while (<>) {
   $line = $_;
   chop($line);
   @words = split($line);
   $lastword = $words[$#words];
   print "Last word = $lastword";
}
```

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Why REXX in UNIX

- Existing shell script languages are very arcane
- Port existing REXX programs to UNIX
- Universal macro language ... *a REXX exclusive*

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099

REXX Symposium

Issues with REXX in UNIX

- Existing shell languages are rich, powerful, universal
- REXX "looks foreign" in UNIX
  - The C heritage of UNIX pervades everything.
  - REXX doesn't look like C
- EXECCOMM ... difficult!?!?!?
- What should the default subcommand environment look like?
- Access to UNIX built-in features

IBM Almaden
Research Center

650 Harry Road

San Jose CA 95120-6099

Summary

REXX in UNIX can play two key roles:

- Portable, easy to use shell script language
- Common embedded macro language

There is stiff competition for the former.

REXX could dominate the latter.

IBM Almaden Research Center

650 Harry Road

San Jose CA 95120-6099