REVIEW OF ANSI AND OTHER LANGUAGE DISCUSSIONS

BRIAN MARKS
IBM

Preamble added April 1991, was not part of the original handout.

In addition to the caveats embedded in this document, note that:

The reaction of SHARE members included rejection of some of these suggestions and ideas for variations which were improvements.

The examples tend to be examples of the simplest case. It is a reasonable rule to assume that the generalizations which seem natural to you were also intended.

This Handout.

This handout accompanies the SHARE session number A636 - REXX Design Dialogue, speaker Dr Brian Marks, on Tuesday February 26th 1991.

Preamble

The Procedures Language Architecture Review Board is an IBM committee that defines the programming language and the interfaces that make up Procedures Language. The Procedures Language Interface Owner, Linda Green, selects particular parts of the committee's output as the SAA Procedures Language levels; so far there have been two levels, SAA level 1.0 announced in March 1987 and level 2.0 announced in June 1990. The committee members all work for IBM and I am the chairman.

The work of the committee naturally divides into work on reconciliation, (wherever there is doubt about what implementations should do to honour the architectural definition), and work to continue the original design principles of REXX into extensions. Today we are going to discuss the design of extensions. The design I will present is at a very early stage; it is in the minds of the Board members. There are no implementation plans and only one part of it is being prototyped. So IBM is making no commitments that any of this will be delivered, or that if it is it will be in the form that we are going to discuss today.

It is a fair question to ask "Why do any work on extensions?". REXX was well designed in the first place, so perhaps extension will do more harm than good by making REXX more complicated.

The argument in favor of change is that the world of computing has changed - what was an optimum design earlier may not be optimum for today or for the next decade. REXX has already benefited from some evolution over the years. The Review Board feels that further evolution may be justified by the trends of the 1990s. Such an evolution is prompted by, and built around, the views expressed by our customers.

The purpose of today's session is to begin a dialog with you about the Board's view of what might be appropriate; we call this 'Architected REXX'. The remainder of this session will be split into parts. In each of the parts I will describe a component of 'Architected REXX' and you will have the opportunity to discuss it. In order to keep to the timetable it may be necessary to guillotine discussion but I am sure there will be other opportunities.

The first of these parts relates to the trend for the problems that programmers are solving to be more complex. Although programmers today have better equipment than ever before, there is still a challenge in programming because our ambitions have increased. In the case of REXX, writing big programs exposes some limitations in the way variables are handled. As the number of variables in a program increases it becomes difficult to control the scope of variables with simple PROCEDURE EXPOSE statements. The recent extension of EXPOSE to allow a list of variables to be named is an improvement but users still tell us that better facilities for sharing and scoping are required.

Architected REXX postulates the addition of external procedures that share variables and parameter passing 'by reference'. Here is a foil with syntax. In the example the variable ABC is shared by the main program and the subroutine ALPHA.

```
/* Main Program */                    /* External procedure */
  .                                   ALPHA:PROCEDURE EXPOSE ABC
  call ALPHA                             .
  say ABC                                ABC=66
  .                                      .
  exit                                  return
```

This is the same syntax and meaning as is currently used for internal subroutines, but external routines today cannot start with a PROCEDURE statement. So this is a 'clean' extension - no correct existing program is 'broken', ie given a different meaning, by the extension. Also it introduces little in the way of new terminology and concepts.

By-reference addressing is not so 'clean'.

```
/* Calling code */                    /* Called routine */
  call BETA MYVAR.                    BETA:
  say MYVAR.33                         use arg GAMMA.
                                       GAMMA.33='Something'
                                       return
```

This program would do something today, but nothing very useful - because there is no 'use' statement today, that line would issue a command. That is not a serious breakage problem because the coder who wanted to do that would almost certainly have used quote signs around the word 'use'.

The use statement introduces a name for an argument, like PARSE ARG does, with the difference that the argument IS NOT COPIED. The name introduced is a second name for the same variable. In this example MYVAR and GAMMA are the same variable. This aliasing is a powerful feature and also a source of pitfalls for the unwary. There are alternative designs, mostly involving multiple values on the return statement, but the Board feels that passing By-Reference is the correct choice for execution speed, since return of multiple values could require more copying of data.

This new sort of external routine, that starts with a procedure statement, is also more like an internal procedure in the way that internal values (like the current number of NUMERIC DIGITS) are handled. Today's external routines reset these internal values when the routine starts; the new sort of procedure inherits the caller's settings in the way that an internal procedure does today.

We will take our first discussion period now. I believe the essential questions are:

- Does the expectation of more complex programming justify additions to REXX?
- Is adding External-like-Internal and By-Reference-Arguments enough to alleviate the difficulties in sharing that people have experienced?
- Are there better designs of language with the same power?

National Language Sensitivity.

Our second area for design dialog is National Language Sensitivity. Our meeting today has a majority of people whose natural language is English, and REXX is optimized to people who know American-English, so this may seem seem a minor design issue. However, there are two trends of the nineties that make it increasingly important. The first is an increasing number of non-English speaking programmers. The second is the explosion in communications which is making our world into a village and making possible individual applications which have widely spread parts.

We all know this is a hard problem to tackle - the complexities of code pages and character sets together with the variety of dialects and customs makes a daunting challenge. Fortunately we are not on our own - all the Programming Languages, the operating systems, and the components like SQL are involved. An IBM architecture is emerging - the Character Data Representation Architecture which you can hear more about at other SHARE sessions.

REXX has the advantage over some programming languages that it is defined in terms of characters rather than bytes, and the definition stands up whether the characters are physically represented as one, two or a variable number of bytes. The extensions in Architected REXX provide for:

1. Source programs written in the characters sets identified by CDRA. (Those that implementations support - we would expect that to be a large number.)
2. A set of rules for coping with the specialities of particular character sets - eg which characters are allowed in names, how substitutes are used for unavailable character sets, what Uppercasing means. (By the way, Lowercasing is in the design.)
3. Existing keywords, function results remain in English. To do otherwise would cause a lot of breakage.
4. New variations on the builtin functions allow, for example the day of the week to be returned in French. (This by retaining the same names for builtin functions but adding variety to the arguments, eg DATE('W') for the local form of weekday.)
5. Run time data which is not in the same character set as the source program is permitted. However there are no automatic conversions between character sets.

This design follows CDRA in the idea that data is 'tagged' with identification of the character set that the data is in. Whether these 'tags' or 'attributes' are actually present will depend on their operating system support. Our design allows for the character set to be given in a REXX-specific way if the operating system support for tags is not present.

Some of the NLS questions:

- Do the trends justify adding these features?
- Is the extent of the support appropriate? Or maybe we need keywords in non-English? Automatic conversions at runtime to some Universal character set?
- What is the best design, given the extent of the support?

Message Driven Processing.

Our next area for design dialog is Message Driven Processing. It will be characteristic of the nineties that many applications will be distributed, with parts of the applications on different machines and often geographically far apart. Such divisions need a clear way of specifying what data and functions belong to one part of the application as opposed to another. The message driven paradigm, also known as 'Object Oriented', has proved to be good for this. And of course object orientation has also proved good for other things, like manipulating windows on a screen, for the same reason as it is good for distributed applications - because of the 'data encapsulation'.

Architected REXX favors the Object Oriented style developed by the OO-REXX team. Simon Nash talked about this prototyping effort at SHARE74 and is giving an update on Wednesday at 0930. There will be an opportunity then for a detailed discussion. Right now, I will recap on the main features so that we can discuss how this fits with other parts of Architected REXX.

These Message Driven Programming facilities introduce only insignificant breakage so a programmer who does not want to use them need not know about them. Such a programmer can continue to program using non-object-oriented features and terminology. It will be a choice for the programmer whether to adopt the OO-REXX style.

Programs in the OO-REXX style use Methods, analogous to external procedures, with a new METHOD statement analogous to PROCEDURE. Methods provide two related facilities, the encapsulation of data (variables on a METHOD EXPOSE are shared only across the methods associated with an object) and a unit of execution that can be paralleled (the method invocation).

Methods are invoked by a new form of REXX term, the 'message term', or a new instruction, the 'message instruction'. Each of these has the syntax (in the simplest case) of a function call preceded by a term and the tilde character. eg rectarea = myrect~area; mystack~push('Bill Brown')

In the Object Oriented terminology, the object on the left hand side of the tilde responds to the message (on the right hand side) by returning a result object. The objects may be strings, in which case the newness may be solely in terminology and syntax.
'ABC'~REVERSE = = 'CBA' = = REVERSE('ABC')

However, the objects need not be strings. Objects are characterised by the methods that can be applied to them, and there are Builtin methods which will create objects and associate methods with them. In this way the usual object-oriented features of powerful objects, inheritance etc. are established mainly by the programmer. Only the essential primitives have been added to REXX in this enhancement. Any particular problem oriented solution, eg a windowing scheme, could be provided as a package of pre-programmed objects but will not be part of this extension.

The parallel nature of object activity is achieved by the addition of a REPLY statement analogous to the RETURN statement. REPLY does what RETURN does but additionally continues execution (with the statement following the REPLY). Where this might lead to unsynchronized shared access to variables the programmer should make use of 'guards'. The guard statement, with syntax GUARD expression, blocks execution until the expression evaluates to '1'.

Some of the high level message driven processing questions:

- Do the trends justify adding these features?
- Is the extent of the support appropriate? Too high because it adds a whole new set of concepts and extends the character set required for REXX? Too low because it only provides mechanisms, and does not define a comprehensive set of useful objects as a part of REXX?
- Should it be viewed as a different language, analogous to the relation of C and C++, or is it right to design it as a compatible component of architected REXX?

Calling non-REXX code - the Generic Binding.

It has always been possible to call non-REXX code from REXX code; the necessary interfaces are defined and publicized. But it is not the easiest thing to do - it requires a knowledge of parameter passing details and requires some low-level programming. The difficulty hampers the development of applications in which REXX is used to harness other facilities. This applies whether the facilities are IBM supplied, like SQL, or developed by a customer.

The design the board favors has three features:

1. A set of conventions about how to pass arguments to packages. For example, if an array is to be passed the elements of the array should be assigned to SomeName.1, SomeName.2, SomeName.3, etc. and the stemmed variable SomeName. passed.
2. A language in which the developer of a package can describe the entry points of the package. This language is essentially declarations in the programming language 'C'.
3. A mechanism in the REXX implementation to convert REXX arguments to non-REXX format and pass them to non-REXX procedures, without the need for anyone to program these conversions.

The user of a package only needs to know the conventions. Such a user will not even be aware of the language in which the package is written. The developer of a package needs to describe the entry points and make use of a utility program to convert the specification of the package into a table to be used when the package is used. Only the developer of a REXX interpreter or compiler needs to know about how arguments are actually converted and passed.

Some key questions are:

- Is the investment in such a general solution justified, or are the packages and system components that will need to be accessed sufficiently few to assume they should hand-craft their own interfaces?

- How far can the infinite variety of non-REXX arguments be accommodated? eg should it be possible to pass a REXX procedure name to non-REXX code and have the non-REXX code subsequently call that REXX procedure?
- If there have to be restrictions on what can be passed, does that make the whole approach unjustified?

Debugging paradigms.

At this point I am going to mention something that is not part of architected REXX, but is a suitable subject for dialog. There is a world of difference between debugging with current REXX trace facilities and the debugging schemes available with some other languages. The latter may have multiple windows showing relevant source, variable values, tracebacks, breakpoints - you know the sort of thing I mean. I have no specific proposal, but we can discuss:

- Should there be an ambition to debug REXX in this way?
- Should it be regarded as something for the system to provide, for all programming languages, or a REXX facility?
- What would the relation to existing TRACE be? A replacement, or in some way an evolution?
- What are the implications for existing proposals to embellish TRACE?

Other items.

Our final area for design dialog covers a selection of smaller items which are not so much driven by changes in the computing scene but are more a matter of filling gaps in the general data processing capabilities of REXX. I think you will recognise them as SHARE requirements although they may not match the exact form of submitted requests.

1. Iterating over associative arrays. Builtin function TAILS returns the number of tails. NEXTTAIL returns a tail, or the successor to a given tail. The sequence produced by NEXTTAIL is guaranteed to include all tails just once, if there is no intervening creation or deletion of tails. An example loop to traverse the tails:

```
if TAILS('Mystem.')>0 then do
 Given = NEXTTAIL('Mystem.');Current=Given
 do until Current=Given /* Process Current */
   Current=NEXTTAIL('Mystem.',Current)
 end
end
```

   (Design of this feature was made more difficult by the fact that there is no string value which cannot be a tail.)

2. String functions more symmetric; negative values for positions are no longer errors; they define the position as counted from the right end of the string. This sets direction, and lengths are counted in that direction.
3. More situations are introduced in which the result of an expression is used as a symbol:

   a) `call (expression)`

Expression evaluates to the symbol called. Note that this is not extended to functions because of the breakage; (abc)(def) is concatenation.

b) `(expression)=rhs`

Neater than using VALUE. Expression evaluates to the symbol that is the target. There is breakage in theory, but who passes 0 and 1 to the host system?

c) `A.(J+1)=99 /* Same as T=J+1;A.T=99 */`

There is breakage, but who uses procedure names that end with a dot? I should point out that this item is not as solidly supported by the board as the rest of architected REXX is. (An arbitarily complex symbol doesn't fit well with the structure of existing interpreters; there is a risk that even those who don't use the feature may suffer a performance penalty from its existence.)

4. DATE() and TIME() builtin functions are extended to have conversion, allowing time arithmetic. Syntax is DATE(outputformat,inputvalue,inputformat). There is no builtin help for 'carry' from time calculation into date calculation.

Some of the questions relevant to these features:

- Does the extra complication outweigh their usefulness?
- Is the breakage tolerable?
- And since this is the last discussion period, it would be an appropriate time for you to voice opinions on the total architected REXX design.