Introduction

The REXX langauge was created in the late 1970's by Mike Cowlishaw of IBM as a command programming language for IBM mainframes. Because the available languages of the time were quite cumbersome to use, Cowlishaw set out to create a language that was more like a human language than a computer language. Thus was born what would later be called REXX, the Restructured Extended Executor. Because of its ease of use, REXX quickly became popular both inside and outside of IBM. Today REXX is available for all of the major operating systems, and for some, like IBM's OS/2 Warp, it is integrated seamlessly into the operating system.

Because of its design, REXX excels at dealing with the two things people usually work with: words (strings) and numbers. REXX has some very powerful string-handling features and the precision of its numerical functions can be controlled by the programmer. If you need 20 digits of precision in your program, one simple command can be used to set it. If you later need, say, 40 digits of precision, only the one line in the program has to be changed.

The following is an introduction to REXX, with an emphasis on its use with MainActor. It is not intended to be an exhaustive description of the language, but to provide new users with some of the basics of the language and to point out common errors that new users make.

REXX Basics

This section describes the basics of the REXX language for those with little or no REXX proramming experience.

Creating a simple REXX program

Most people find that learning a new programming language is most easily accomplished by looking at example code to accomplish a specific task. That is the approach we will take in this document. To start out, let's create a REXX program that prints a message to the screen.

REXX programs must begin with a comment line. This is a convenient place to describe what the program does, so that when you don't use it for a long time, you can look at the program source code and find out the program's purpose. Comments in REXX begin with a forward slash and asterisk, and end with an asterisk and forward slash like this:

```
/* This is my very first REXX program. */
```

Comments can extend across multiple lines. The following examples are all valid REXX comments:

```
/*
   This is my very first REXX program.
*/
```

```
/* This is my very first REXX program.
   The program prints a message to the screen. */
```

Now, how do we print something on the screen? The **SAY** instruction is used to accomplish this. By supplying the **SAY** instruction with a string, you can print a message to the screen (or more correctly *STDOUT* which is usually the screen unless you have redirected the output). So, our simple REXX program would look like this:

```
/* This is my first REXX program. All it does is
   print a message on the screen. */
Say "Hello! This message is from REXX."
```

Type in the above code and save it to a file. Under the **Edit** menu in MainActor, choose **Start Rexx** and then select the file in the file dialog. An output window will appear as well as an input window. Type in the requested data and the greetings will be printed to the output window.

Note that the string to be printed is surrounded by quote marks. Quote marks are used to indicate literal strings. You can use either double or single quotes. Our program could just as well be written this way:

```
/* This is my first REXX program. All it does is
   print a message on the screen. */
Say 'Hello! This message is from REXX.'
```

Congratulations! You are now a REXX programmer!

Getting input from the user

Now let's modify our simple program from the previous section to ask the user for the string to be printed to the screen. Again we use the **SAY** instruction to print a question on the screen:

```
/* Modified program to ask the user for a string, and
   then print it on the screen */
Say "What message would you like me to print?"
```

Now we need to get the string from the user. To do this we use the **PARSE PULL** instruction. Parsing is the act of processing a string, and we are "pulling" a string from the user, hence the syntax of this instruction. (As we will see later **PARSE** is a very powerful instruction that allows you to do all sorts of manipulations on strings.) To get an answer from the user, we use this:

```
Parse Pull Answer
```

Answer is a variable that will contain the string that the user entered. Once we have that, all we have to do is use the **SAY** instruction to print it to the screen:

```
/* Modified program to ask the user for a string, and
   then print it on the screen */
Say "What message would you like me to print?"
Parse Pull Answer
Say "Your message was:"
Say Answer
```

Now, if you have some experience with other programming languages like BASIC, you might be wondering how REXX nows that the variable *Answer* is supposed to hold a string. Nowhere have we told REXX what *type* of variable *Answer* is. The reason is that *all* variables in REXX are strings, and we will discuss this more in the next section.

Data Types (or lack thereof) and Numerical Operations

Very often the first stumbling block people run into with REXX, if they have experience with other programming languages, is the lack of data types. In REXX everything is a string, so there is no need to declare variables as integer, floating, string, etc. as in other languages. This makes REXX much easier to use, but it also results in a performance hit.

So, if everything is a string, how can REXX perform mathematical functions? If you perform a mathematical operation on a REXX variable, REXX handles everything for you. There is no need to do any conversions. For example, let's write a program that asks the user for two numbers and then prints out their sum and product:

```
/* Input two numbers from the user and print
   the results to the screen */
Say "Enter the first number:"
Parse Pull FirstNumber
Say "Enter the second number:"
Parse Pull SecondNumber
Product=FirstNumber*SecondNumber
Sum=FirstNumber+SecondNumber
Say "The product is" Product
Say "The sum is" Sum
```

If you enter numbers, the program will work as expected. But try entering a letter or word instead of a number when you run the program. REXX will accept the input, but it will generate an error when it tries to do the multiplication. This is the infamous "Bad arithmetic conversion" error. Remember this one because you are sure to see it. It means that you have tried to perform a mathematical operation on something that is not a number.

By default REXX uses nine digits of precision. This can be changed by using the **Numeric Digits** instruction before you begin any numerical operations. To set the precision to twenty digits use:

```
Numeric Digits 20
```

Unfortunately REXX does not have built-in support for more advanced functions like the trigonometric functions. Only the basic functions (addition, subtraction, multiplication, and division) are built into the language. However, there are usually freely available add-on libraries that provide these functions.

Conditionals and Boolean Operators

REXX has the often-used **IF ... THEN ... ELSE** operator and it works just like other languages except for one important point- there is no **ENDIF** marker. In its simple form, you use **IF** to decide whether to execute one statement or another (or perhaps do nothing). For example, let's write a program that asks the user to enter either 1 or 2 and then prints the word "One" or "Two":

```
/* Asks the user for a 1 or a 2 and then prints out the
   corresponding word. */
Say "Please enter 1 or 2 and press return"
Parse Pull Number
If Number=1 then
   Say "One"
Else
   Say "Two"
```

The statement following the **IF** statement is executed if the test is true. If the test statement is false, then the statement following the **ELSE** statement is executed. The **ELSE**statement is optional. If you want to execute multiple statements, you have to enclose them in a **DO ... END** loop like this:

```
If Number=1 then
   Do
      Say "One"
      Single=1
      Double=0
   End
Else
   Do
      Say "Two"
      Single=0
      Double=1
   End
```

You can use the usual Boolean operators in the test statement. In REXX, as in many other languages, the operators are

- = equal to
- < less than
- > greater than
- <= less than or equal to
- >= greater than or equal to
- not equal to
- <> greater than or less than (same as)
- not less than
- not greater than

The above operators can compare numbers and strings, although in the latter case the comparison will be done after leading and trailing blanks are stripped away. To use strict comparison with strings different operators are used. (In the normal comparison 0 and 0.0 are equal, but in a strict comparison they are not.). The strict comparison operators are

- == equal to
- << less than
- >> greater than
- <<= less than or equal to
- >>= greater than or equal to
- = not equal to
- < not less than
- >not greater than

The boolean operators for combining test statements are

- &amp. and
- | or
- &amp.&amp. xor
- not (used to reverse one of the above)

The following is an example of using the boolean operators:

```
/* Test to see if two variables have particular values */
If Number=1 &amp. Product="MainActor" then
   Say "Customer ordered one copy of MainActor."
If Number>1 and Product="MainActor" then
   Say "Customer ordered" Number "copies of MainActor."
```

Loops

REXX has some very flexible capabilities when it comes to executing instructions multiple times. The heart of REXX loops is the **DO ... END** structure. If you know ahead of time how many times you want to execute the instructions, the loop is very simple. For example, to make the computer beep 5 times, you would use this:

```
/* Make the computer beep five times at a frequency of 440 Hz
   (middle A note) for half a second each beep */
Do 5
   Call Beep 440,500 /* Last parameter is the duration in milliseconds */
End
```

The number after the **DO** statement is the number of times that the loop will be repeated. Should you wish to have the loop execute indefinitely, simply use **FOREVER** after **DO**.
In most cases you will want to have the loop controlled by a variable. If we wanted to make the beeps change pitch each time, we could use something like this:

```
/* Make the computer beep five times and change the frequency of
   the beep each time */
Delta=50
Do i=1 to 5
   Call Beep 390+(Delta*i),500
End
```

Using the **BY** statement we could write the above program a bit more efficiently:

```
Do i=390 to 640 by 50
   Call Beep i, 500
End
```

Although more efficient, this method requires that we calculate the last value for the frequency. The most efficient and easiest way to do it is to use a third form of the **DO** loop:

```
Do i=440 For 5 by 50
   Call Beep i, 500
End
```

Breaking Out Of Loops or Iterating

There are times when you need to either break out of a loop or go on to the next iteration. REXX provides the **LEAVE** and **ITERATE** instructions for these purposes. You would most likely use **LEAVE** in a **DO FOREVER** loop. Here is a program that loops until the user enters the answer "Y" to a question:

```
/* Ask the user a question repeatedly until they enter "Y" as the answer */
Do Forever
   "Does MainActor support more file formats than any other program?"
   Parse Pull Answer
   If Answer="Y" then
      Leave
   Else
      Say "You did not answer Y. Try again."
End
```

ITERATE is used when you want to continue to the next iteration rather than leaving the loop altogether. Here is a program that prints the numbers 1 to 10 except for the number 7:

```
/* Print the numbers 1 to 10, except 7 */
Do i=1 to 10
   If i=7 then
      Iterate
   Else
      Say i
End
```

Stems (Arrays)

Stem variables in REXX behave in many ways like arrays in other languages, but they have some powerful advantages in certain situations. Stem variables, like arrays, are useful when you want to group pieces of data together. For example, you might be working with data that involve employees- name, identification number, salary, *etc.*. In that case, it would be useful to have variables like *Name*, *ID*, and *Salary* and be able to loop over them using an index number. REXX stems can do just this.

A stem variable is created using a valid variable name with a dot, as in *Name.*, with the index coming after the dot: *Name.1*, *Name.2*, *etc* One very nice feature of stems is that the index need not be a number. It is perfectly valid to use strings like this:

```
Employee.Name="Bob Smith"
Employee.ID="1234567"
Employee.Salary="36000"
```

(One thing to be careful of when using strings as indices is to make sure that you aren't using a variable of the same name. For example if you have Fred=1 and Name="Fred" and then use Employee.Name, that is the same as Employee.1. This is a very powerful feature but it can lead to unexpected results if you don't pay attention to it.)

If you need multiple dimensions, just use multiple dots, such as Name.1.1, Name.1.2, Name.2.1, Name.2.2 and so on.

Unlike some other languages, REXX stems need not be "declared" before being used. Also, stem variables can have the same name as other non-stem variables. *Name.* and *Name* are two different variables as far as REXX is concerned.

Manipulating Strings

Some of the most powerful capabilities of REXX involve operations on strings. This section discusses some of the string-handling functions built into REXX.

Basic String Functions

Below are some of the basic string-handling functions of REXX. In the calling methods, a parameter enclosed by square brackets is optional.

LENGTH Returns the length (in characters) of a string. Length("A string") would return 8.

POS Returns the position of the first occurence of one string in another. The calling format is

```
Pos(Target,Source,[Start])
```

where *Target* is the string we are looking for, *Source* is the string that we are searching in, and *Start* is an optional point (in characters) in the source string to begin the search (defaults to the beginning of the source string).

SUBSTR Returns part of a string. The calling form is

```
Part=Substr(string,start,length)
```

where *Part* is the part of the string, *string* is the main string we want a part of, *start* is the starting position (in bytes) in the main string, and *length* is the length of the substring that we want.

LASTPOS is similar to **POS** except that it returns the location of the *last* occurrence of the target string in the source string.

WORDS returns the number of blank-delimited words in a string. Words("A string") returns 2, for example.

WORD returns a particular word in a string. The syntax is

```
Word(string,index)
```

where *string* is a REXX string and *index* tells which word in the string we want. The first word is 1, the second is 2, and so on. Word("A string",2) would return *string*.

STRIP strips leading and/or trailing characters from a string. This function is useful for removing spaces at the beginning and end of a string. The calling form is

```
Strip(string,[option],[character])
```

where *string* is the string we are operating on, *option* is either B (both leading and trailing characters, the default), L (leading characters only), or T (trailing characters only), and *character* is the character to strip with the default being a space. To strip leading and trailing spaces from a string simply use the form *Strip(string)*. To strip non-printing characters such as tabs, use the **D2C** (decimal to character) function which allows you to specify characters by their ASCII codes. *Strip(" Some string with leading and trailing tabs ","B",D2C(9))* would strip both leading and trailing tabs (ASCII 9) from the string.

Concatenating Strings

REXX provides several ways of concatenating strings. The explicit concatenation operator is **||** as in

```
"A " || " String"
```

which results in *A String* You can also simply list one string after another as in this exmaple:

```
String1="This is"
String2="an example"
String3="of string concatenation."
Say S  tring1 String2 String3
```

You can also abutt literal strings with other literal strings or variables like this:

```
String1="This is"
Say String"a string."
```

The output would be *This isa string.* because no space is placed between the strings when they are abutted. You would get the same result if you used

```
Say String||"a string"
```

Using the PARSE Function

In REXX, *parsing* is the process of splitting a source string into parts. For example, you might have a list of people's names containing their first and last names, and need to split the first and last names from each other. The **PARSE** instruction does just this. We have already used **PARSE** to get input from the user. Let's write a program that asks the user for their name (first and last) and then greets them informally and formally:

```
/* This program asks the user for his or her full name and
   then greets them informally and formally */
Say "Enter your full name (e.g. Bob Smith):"
Parse Pull FullName
Parse Var FullName FirstName LastName
Say "Hi" FirstName
Say "Hello Mr./Ms." LastName
```

In the **PARSE** instruction we split the full name that the user input into first and last names and then used that create the greetings. The *Var* part of the instruction tells REXX that the source string we want to operate on is going to be found in a variable. The name of that variable comes after the *Var* option. When we asked the user for the full name, we stored it in the variable *FullName*, so that is the variable name that should come after *Var*. What comes after the source variable is called the *template*. The template is where the real power comes in because that is where you tell REXX exactly how you want it to split the source string. In our case we have a string that consists of two words- the first and last names. By default REXX uses the space as the separator, and that is what we need in this example. So our template, *FirstName LastName*, tells REXX to put the first word (up to the first space) into the variable *FirstName* and the second word into the variable *LastName*.

What happens in the above example if the user enters a name like *Joe Bob Smith*? That is, what happens when the number of words in the source string is larger than the number of variables in the template? In that case REXX puts one word in each of the first variables and when it gets to the last template variable, it throws the rest of the string into the last template variable. So if Joe Bob Smith runs the program above, *FirstName* will contain Joe and *LastName* will contain "Bob Smith".

What about the situation where the separator is some other character besides the space? For example, what if our source string is the name of an image file, say *image1.tif* and we want to get the file extension so that we can perform operations appropriate to the file type? In that case we have to specify the separator in the parse template. Our separator is the dot, so our **PARSE** line would look like this:

```
/* This program asks the user for a file name and then splits it
   into the leading part plus the file extension */
Say "Enter a file name (e.g., image1.tif):"
Parse Pull FileName
Parse Var FileName FirstPart "." Extension
Say "The first part is" FirstPart
Say "The file extension is" Extension
```

By putting the *"."* between the two variables in the template, we are telling REXX to use the dot as the separator.

A third variation of the template is useful when the separator character is either unknown at the time the program is written, changes from one run to the next, or is not a printable character (such as a tab or carriage return). In these cases, we would like to have the separator be specified by the contents of a variable. To accomplish this, put the name of the variable in parentheses:

```
/* This program does the same thing as the file name splitter, but
   uses the contents of a variable as the separator */
Say "Enter a file name (e.g., image1.tif):"
Parse Pull FileName
Separator="."
Parse Var FileName FirstPart (Separator) Extension
Say "The first part is" FirstPart
Say "The file extension is" Extension
```

This program gives the same results as the previous one. A better example of the usefulness of this method is a situation where the separator is an unprintable character, say, a tab. Just replace

```
Separator="."
```

with

```
Separator=D2C(9)
```

The ASCII code for a tab is 9 and the **D2C** function (**D**ecimal to **C**haracter) creates a string given a character's ASCII code. The variable *Separator* now contains the tab character, and you can parse tab-delimited strings using *(Separator)* in the template. There are other functions similar to **D2C** such as **C2D** which is the inverse of **D2C** (returns the ASCII code of a given character), **C2X** (character to hexadecimal; **X2C** is the inverse), and **B2X** (binary to hexadecimal; **X2B** is the inverse).

File I/O

REXX has several functions for reading from and writing to a disk and for determining various properties of disk files. This section covers these aspects of file I/O operations.

The STREAM Function

The **STREAM** function is used to get information on a file (or in general, a data stream, hence the name) or to perform certain operations on the file. The calling form is

```
Stream(Filename,[Option],[Command])
```

where *Filename* is the name of the file, *Option* is either C (command), D (description), or S (state, the default value), and *Command* is a command name to perform a specific operation on the stream. The state and description operations are similar in that they return information on the status of the file, but the description option will generally give more detailed information should an ERROR or NOTREADY status be returned. The Command option has several useful variations (which are specified in the third parameter passed to the function).

Passing the *open* command to **STREAM** as in

```
Stream("C:\CONFIG.SYS","C","Open")
```

would open the file *CONFIG.SYS* in read-write mode (*i.e.*, you can perform read and write operations on the file. Generally, it is **not** necessary to explicitly open a file for reading or writing as in other languages. Performing a read or write operation on a closed file will implicitly open it. To open a file in read-only or write-only mode simply append *read* or *write* onto the *open* command as in

```
Stream("C:\CONFIG.SYS","C","Open read")
```

When doing file I/O you must remember to close a file when you are finished with it. If you run into a situation where you are doing operations on many files, but your program mysteriously just stops after doing a few of them, it is very likely that you are not closing the files and are running out of file handles. **This is a very common mistake that people waste a lot of time tracking to track down!** To close a file after you are finished reading or writing, simply call **STREAM** like this:

```
Stream("somefile.txt","C","Close")
```

One very useful stream command is **QUERY**. You can use this command to determine if a file exists, what its size is, and get its date and timestamps. To determine if a file exists, use

```
File=Stream("somefile.txt","C","Query Exists")
```

If the file exists, the full pathname of the file will be returned in the variable *File*. (You don't have to use the name *File*. Any valid variable name will work.) If the file does not exist, it will return a null (empty) string.

To get the size of a file (in bytes), you would append *size* onto the command:

```
Size=Stream("somefile.txt","C","Query Size")
```

The variable *Size* would then contain the size of the file in bytes.

Finally, to get the date and time associated with a file, use *datetime* as in

```
Timestamp=Stream("somefile.txt","C","Query Datetime")
```

The date and time would be returned in a form like *09-15-97   14:26:38*

Reading Data From Files

There are two REXX functions for reading data from a file depending on whether you want to read the file byte-by-byte or line-by-line. To read a file byte-by-byte, you use the **CHARIN** function whose calling form is:

```
result=Charin(filename,start,length)
```

where *result* is the variable that will hold the data that are read (any valid REXX variable name can be used), *filename* is the name of the file to read from, *start* is the position (in bytes) to begin reading from (The default is the beginning of the file.), and *length* tells how many bytes to read (The default is one byte.). To read the first byte of a file, you would use this form:

```
FirstByte=Charin("somefile.txt")
```

Or you could explicitly specify the start and length parameters:

```
FirstByte=Charin("somefile.txt",1,1)
```

It is not necessary to keep track of the current position if you are reading bytes sequentially from a file. If you wanted to read an entire file two bytes at a time, you would make multiple calls to **CHARIN** like this:

```
FirstByte=Charin("somefile.txt",,2)
```

leaving the start parameter blank. You can use the **CHARS** function to determine how many bytes are left to be read:

```
BytesLeft=Chars("somefile.txt")
```

If you want to read text files line-by-line, REXX provides the **LINEIN** function. This function reads in one line at a time putting the line (minus the end-of-line character or characters) into a variable:

```
Result=Linein("somefile.txt")
```

Combining **LINEIN** with the **LINES** function makes it possible to read in the contents of a text file. **LINES** has different return values under different REXX implementations. Under some, it returns a 0 (zero) if there are no more lines to be read in and 1 (one) if there are more lines to be read. In some versions of REXX, **LINES** will return the number of lines that remain to be read. To avoid portability problems, you can just test the result to see if it is greater than zero, if so, you can continue to read from the file.
The following program would read in all of the lines of a file and print them to the screen (like the *type* command in DOS and OS/2 and *cat* in Unix):

```
/* This program prints the contents of a text file to the screen. */
Do While Lines("somefile.txt")>0
   Result=Linein("somefile.txt") /* Read in a line */
   Say Result /* Print the line to the screen */
End
/* Close the file */
ReturnCode=Stream("somefile.txt","C","Close")
```

Writing Data To Files

REXX provides **CHAROUT** and **LINEOUT** to perform byte-by-byte and line-by-line output. The calling form for **CHAROUT** is

```
count=Charout(filename,string,start)
```

where *count* is the number of bytes remaining to be written after the function call (0 if all bytes were written), *string* is a sequence of bytes to write to the file specified in *filename*, and *start* is the byte position in the file to begin writing (defaults the beginning of the file on the first call and the current position thereafter).

LINEOUT is used when you want to write lines of text to a file. **LINEOUT** will write a specified string to the file and add the appropriate end-of-line sequence for your operating system (*e.g.* a carriage return/linefeed pair on DOS/Windows and OS/2 machines, linefeed for Unix). The calling form is

```
ReturnCode=Lineout("somefile.txt",string)
```

One thing to watch out for when writing data to a file is whether or not the file already exists. Under some systems, using **LINEOUT** (or **CHAROUT**) on an existing file will replace the contents of the file. Under others (OS/2 for example), the data will be appended to the existing file, so if you want the old data to be replaced you have to delete the file before you attempt to write to it.

Calling System Functions or External Programs

Often it is necessary to perform some function that REXX itself is not capable of doing or is not efficient at doing. In these cases it is necessary to call on functions built into the operating system or perhaps another program. Copying a file is an example of the former. There is no built-in REXX function to copy a file. It could be done purely in REXX using the file I/O functions, but why not just call the copy command of the operating system? Another example would be a situation where you need to do some intensive numerical calculations. Since REXX is interpreted, it is slow compared to compiled languages like C or Fortran. It would be more efficient to call on a compiled program to do the calculations and then process the output with REXX.

Fortunately, calling external programs from REXX is very easy. Any statement that REXX does not recognize, it simply passes to the operating system. There is no copy command in REXX, so if you have a line that reads

```
copy "c:\config.sys" "c:\config.bak"
```

REXX will not recognize *copy* and therefore it will pass it to the operating system, which, if you are running OS/2 or Windows, it will understand.

A bit of advice that will probably save you some headaches is to put external commands in quotes to ensure that REXX doesn't interpret the command as a variable. The copy command above would be bettter written as

```
"copy c:\config.sys c:\config.bak"
```

to avoid any problems. Of course if you want to pass the contents of variables to the operating system or an external program, do not put the variable names inside the quotes. An example that accomplishes the same thing as the copy command above, but uses variables for the filenames, would be

```
SourceFile="c:\config.sys"
BackupFile="c:\config.bak"
"copy" SourceFile BackupFile
```

If you need to get a return code from a call to an external function or program, check the value of the variable RC right after the call. REXX will automatically put the return code there.

Using REXX With MainActor

REXX was chosen as the scripting language for MainActor because it is a simple, but powerful language. In this section we will look at some applications of REXX within MainActor.

Running REXX Scripts in MainActor

REXX scripts in MainActor are executed by selecting *Start Rexx...* in the *Edit* menu or by passing the name of a REXX script file on the command line as in

```
mactr Script.rex
```

Creating an Animation File

To start, let's write a simple REXX script that will create an animated GIF from a set of TARGA files. Suppose we have created a series of images of the Earth as it rotates and that the file names are *earth01.tga*, *earth02.tga*, ... *earth20.tga*. First we need to load the images into MainActor. This is accomplished with the **LoadPictureList** instruction. The calling form is

```
rc=LoadPictureList(BasePicture,Number)
```

where **BasePicture** is the first picture to load (in our case this is earth01.tga) and *Number* is the number of pictures to load. If you specify 0 for *Number* then MainActor will load all of the appropriate images. The variable *rc* will contain the return code from the call to the function. If the return code is 0, then the function call was successful. If the return code is any other value, a problem was encountered.

Assuming that the call to **LoadPictureList** was successful, we now need to save the animation to disk. The **Save** function enables you to create the animation file. The calling form is

```
rc=Save(File,Module,Codec,Frames,Width,Height)
```

where *File* is the filename (with path information if desired), *Module* is the animation module to use (in our case "GIF-Anim" since we want an animated GIF), *Codec* is the codec you want to use (for an animated GIF, the only value is "GIF-LZW"), *Frames* is either "AllFrames" or "SelectedFrames" (in our case we want "AllFrames"), *Width* is the desired width of the animation, and *Height* is the height of the animation. To keep the width and height the same as the source images, enter 0 for these parameters.

So, our script to create an animated GIF boils down to two lines of code! Of course, it is always a good idea to at least look at the return codes, so we will add a couple of lines to print them out. Here is the source code for our animation creator:

```
/* MainActor script to load the Earth images and
   make an animation out of them
*/
rc=LoadPictureList("F:\Images\earth\earth01.tga",0)
Say "RC from LoadPictureList:" rc
rc=Save("F:\Images\earth\earth.gif","GIF-ANIM","GIF-LZW","AllFrames",0,0)
Say "RC from Save:" rc
```

A More Flexible Animation Creation Script

Now let's look at the MainActor sample script CONVMPEG.REX which also converts a picture list into an animation, namely an MPEG-I animation with the PAL codec. This script is more flexible than the simple animated GIF script we did in the last section because it prompts the user for the project name rather than having it hardcoded into the script. The function used to load a project is **LoadProject(ProjectName)** where *ProjectName* is the name of the project. In this case we want to prompt for the project name, so we enter an empty (null) string for *ProjectName*, that is, "". This script checks the return code from the call to **LoadProject(ProjectName)** to make sure that the project actually exists. If the project does not exist (*i.e.*, the return code is not 0), then the script prints an error message and stops executing by using the REXX **EXIT** command.

This script also provides a nice touch by displaying the time that it took to create the animation. This is done using the built-in REXX function **TIME**. One of the features of **TIME** is that it can act as a stopwatch. To use it, you simply call **TIME** with the "R" option to reset the timer just before entering the section of code that you want to time. Then call **TIME** with the "E" option to get the elapsed time right after exiting the section of code. In this case, we want to know how long a MainActor **Save** operation takes, so that is the only code between the two **TIME** calls.

Here is the source code of the script:

```
/*************************************************************
 *                                                           *
 *                   MainActor Rexx Script                   *
 *                                                           *
 *  Loads a project and converts it to MPEG-I (PAL)          *
 *  Can be customized in any way                             *
 *                                                           *
 *  Last modified: 09/09/97, Written by: Markus Moenig       *
 *                                                           *
 *************************************************************/

  say "Select Project ..."
  rc=LoadProject("")                              /* Load project ... */
  IF rc <> "0" THEN DO
   say "No project loaded! Exiting ..."           /* Failed, exiting ... */
   exit
  END

  filename="c:"                /* Use "" to get a file-requester */
  format="MPEG-I"                  /* Change this to use another format, like
"AVI" */
  codec="PAL"                      /* Use the codec which has "PAL" in its
identifier */
  frames="AllFrames"               /* Use "SelectedFrames" if you only want to
convert */
                                   /* the selected frames of the project */
  width=0                          /* Use original width */
  height=0                         /* Use original height */

  /* You could for example use width=352 and height=288 to make sure that the
*/
```

```
/* created MPEG has the default PAL dimensions */

say "Converting project to" format "..."

TIME("R")                       /* Reset the timer */
Save( filename, format, codec, frames, width, height ) /* Save it ... */

say "Finished, needed" TIME("E") "seconds !"

CloseProject()                  /* Closes the loaded project */
```

Converting an Entire Directory of Projects

This script converts all of the projects in a directory to a particular format (AVI in this example). It illustrates the use of the **LoadProjectPattern** function which loads all projects matching a wildcard pattern. The calling form is

```
rc=LoadProjectPattern(Pattern)
```

where *Pattern* is a wildcard pattern for file names such as "*.tif" or "*.*". This script will process all projects, so we will use "*.*". We also use the **GetProjectInfo** function to retrieve the name of the project by passing "NAME" as the parameter. We then use **PARSE** to split the project name so that we have the first part and the extension. The first part serves as the basis of the new name, which will have the ".avi" added to it. Here is the commented source code:

```
/************************************************************
 *                                                          *
 *                   MainActor Rexx Script                  *
 *                                                          *
 *  Converts all projects inside a directory to AVI. Shows  *
 *  how to use LoadProjectPattern for automatic loading and  *
 *  converting of whole directories.                        *
 *                                                          *
 *  Last modified: 09/19/97, Written by: Markus Moenig      *
 *  Modified slightly by Dirk Terrell on 10/23/97           *
 *                                                          *
 ************************************************************/

  filepattern="h:\gfx\anims\Quick\*.*"   /* Substitute any other directory
here */

  DO WHILE LoadProjectPattern(filepattern) =0  /* This loop runs over all
files */
   BEGIN
    name=GetProjectInfo("NAME")     /* Name of the project */
    Parse Var name newname "." Ext  /* Cut off the extension from name */
    filename="c:\"newname           /* filename, write new files to c:\ */
    format="AVI"                    /* Format */
    codec="Intel Indeo"             /* Use Ultimotion */
    frames="AllFrames"              /* Save all frames of the anim */
    width=0                         /* Use original width */
    height=0                        /* Use original height */

    say "Converting" GetProjectInfo("NAME") "to" filename "as" format "..."
    Save( filename, format, codec, frames, width, height ) /* Save it ... */
  END
```

Getting Project Information

The sample script PROJINFO.REX returns information on the currently loaded project. If none is loaded, then it will prompt the user for a project name. To determine if a project has been loaded, you can use the **GetGlobalInfo** function with "LOADEDPROJECTS" as the parameter, as in

```
rc=GetGlobalInfo("LOADEDPROJECTS")
```

This function returns a 0 if no projects are loaded.

This program also illustrates how to print messages to the status line of the MainActor window rather than the REXX output window. Use the **SAY** instruction to send messages to the REXX output window and **SetInfoText** to send messages to the MainActor window. The calling form for **SetInfoText** is

```
rc=SetInfoText(string)
```

where *string* is the string to be displayed in the status area.

To get information about a loaded project, use the **GetProjectInfo** function. The calling form is

```
GetProjectInfo(option)
```

where *option* is one of the following:

- **AUDIOMODE** - returns "Stereo", "Mono", or a null string if the project has no audio data
- **BITSPERSAMPLE** - returns the bitsize (8 or 16) of the audio data, or a null string if there are no audio data
- **FRAMES** - the number of frames in the project
- **FORMAT** - the format of the project such as AVI, Quick Time, *etc.*
- **FULLNAME** - the file name of the project with full path information included
- **HEIGHT** - the height of the project in pixels
- **NAME** - the name of the project as displayed in the project list
- **SAMPLESPERSECOND** - returns the number of samples per second for audio data or a null string if there are no audio data
- **SIZE** - the size of the project in bytes
- **TIMECODEMODE** - returns "Global Timecode" if the project has a global timecode or "Local Timecodes" if local timecodes are used.
- **TYPE** - returns either "Animation", "Picture List", or "Audio" indicating the type of the project
- **WIDTH** - returns the width of the project in pixels

Here is the commented source code for PROJINFO.REX:

```
/***********************************************************
 *                                                         *
 *                   MainActor Rexx Script                 *
 *                                                         *
 *  Returns information about the currently selected project  *
 *                                                         *
 *  Last modified: 09/04/97, Written by: Markus Moenig     *
 *                                                         *
 ***********************************************************/
```

```
  IF GetGlobalInfo("LOADEDPROJECTS")= "0" THEN DO     /* Check if there are */
   BEGIN                                              /* any projects loaded
*/
    say "No project loaded! Trying to load project..."/* No: try to load one
*/
    rc=LoadProject("")                                /* If succeeded, rc = 0
*/
    IF rc <> "0" THEN DO
      say "No project loaded! Exiting ..."            /* Failed, exiting ...
*/
      exit
    END
   END

  SetInfoText("Printing project information ...")     /* Set a descriptive
info text */

                    /* Use the GetProjectInfo() command to list the project
info */
  say "General Information:"
  say "* Project Name: " GetProjectInfo("NAME")
  IF GetProjectInfo("TYPE") <> "Picture List" THEN   /* If project is no
picture list */
    say "* Full Name: " GetProjectInfo("FULLNAME")     /* print complete name
*/
  say "* Size: " GetProjectInfo("SIZE") " bytes"
  say "* Type: " GetProjectInfo("TYPE")", Format: " GetProjectInfo("FORMAT")
  say "* Width: " GetProjectInfo("WIDTH")", Height: " GetProjectInfo("HEIGHT")
  say "* Frames: " GetProjectInfo("FRAMES")
  say "* Timecode: " GetProjectInfo("TIMECODEMODE")

  audiomode=GetProjectInfo("AUDIOMODE")              /* Check if project has
audio */
  IF audiomode <> NULL THEN DO
    say "Audio Information:"
    say "* Audio Mode: " audiomode                     /* Print audio info */
    say "* Samples per Second: " GetProjectInfo("SAMPLESPERSECOND")
    say "* Bits per Sample: " GetProjectInfo("BITSPERSAMPLE")
  END
```

Writing Frame Information to a File

FRAMINFO.REX gets the available information on the frames of a project and writes it to a file. The **GetFrameInfo** function returns information on frames and is used like this:

```
GetFrameInfo(FrameNumber, Option)
```

where *FrameNumber* is the frame number within the project and *Option* is one of the following:

- **AUDIOSIZE** - returns the size (in bytes) of the audio data or a null string if there are no audio data.
- **COMPRESSION** - the compression method used for the frame.
- **OFFSET** - the offset of the image data of the frame inside the animation. Returns a null string if the frame is a picture.
- **PICTURENAME** - If the project is a picture list, this returns the full path and file name of the picture for this frame. Otherwise, a null string is returned.
- **SIZE** - returns the size (in bytes) of the image data for the frame.
- **TIMECODE** - returns the timecode (in milliseconds) for the frame.

The script uses a **DO** loop to iterate over all of the frames and makes calls to **GetFrameInfo** to get the various pieces of information. The information is then written to a file using the **LINEOUT** instruction. This script illustrates an alternative to using the **STREAM** function to close a file. The file is closed by calling the **LINEOUT** function with a null string as the string to write to the file. Here is the commented source code:

```
/***************************************************************
 *                                                             *
 *                    MainActor Rexx Script                    *
 *                                                             *
 *  Writes all available information for the frames of the    *
 *  current project to a file                                  *
 *                                                             *
 *                                                             *
 *  Last modified: 09/06/97, Written by: Markus Moenig         *
 *                                                             *
 ***************************************************************/

  filename="c:.txt"                            /* Set output filename */

  IF GetGlobalInfo("LOADEDPROJECTS")= "0" THEN DO     /* Check if there are */
   BEGIN                                              /* any projects loaded
*/
    say "No project loaded! Trying to load project..."/* No: try to load one
*/
    rc=LoadProject("")                                /* If succeeded, rc = 0
*/
    IF rc <> "0" THEN DO
      say "No project loaded! Exiting ..."            /* Failed, exiting ...
*/
      exit
    END
   END

  SetInfoText("Writing frame information ...")     /* Set descriptive info 
text */
  say "Writing frame information to" filename "..."

  hasAudio="0"
  IF GetProjectInfo("AUDIOMODE") <> NULL THEN hasAudio="1"  /* Check for audio 
*/

  isPictureList="0"                  /* Check if project is picture list */
  IF GetProjectInfo("TYPE") ="Picture List" THEN isPictureList="1"

  frames=GetProjectInfo("FRAMES")
  i=1;
  DO WHILE i <= frames                            /* Iterate through all 
frames */
   CALL LineOut filename, "Info for frame" i ":"
   CALL LineOut filename, "  Image Data Size:" GetFrameInfo(i, "SIZE")
   CALL LineOut filename, "  Image Data Offset:" GetFrameInfo(i, "OFFSET")
   CALL LineOut filename, "  Timecode:" GetFrameInfo(i, "TIMECODE") "ms"
   CALL LineOut filename, "  Compression:" GetFrameInfo(i, "COMPRESSION")
   CALL LineOut filename, "  Keyframe:" GetFrameInfo(i, "KEYFRAME")
   IF hasAudio <> "0" THEN CALL LineOut filename, "  Audio Size:" 
GetFrameInfo(i, "AUDIOSIZE")
   IF isPictureList ="1" THEN CALL LineOut filename, "  Picture Name:" 
GetFrameInfo(i, "PICTURENAME")
   i=i+1
  END

  CALL LineOut filename                /* Close stream */

  say "Ready."
```

Calculating Frame Statistics

The FRAMSTAT.REX script starts off by getting the number of frames in the project, and then uses a **DO** loop to iterate over all of the frames and get their information. Calls to **GetFrameInfo** are made with the **SIZE** and **TIMECODE** options and running totals of each are kept. (Notice the initialization of the sum variables *totalSize* and *totalTime* to values of 0. Failing to do this will result in the infamous "Bad arithmetic conversion" error.)

After the loop is finished, the average size and number of frames per second are computed by dividing the totals by the number of frames. Here is the commented source code:

```
/*************************************************************
 *                                                           *
 *                   MainActor Rexx Script                   *
 *                                                           *
 *  Returns information and statistics about the size and    *
 *  timecodes for all frames of the current project          *
 *                                                           *
 *  Last modified: 09/04/97, Written by: Markus Moenig       *
 *                                                           *
 *************************************************************/

  IF GetGlobalInfo("LOADEDPROJECTS")= "0" THEN DO     /* Check if there are */
   BEGIN                                              /* any projects loaded
*/
    say "No project loaded! Trying to load project..."/* No: try to load one
*/
    rc=LoadProject("")                                /* If succeeded, rc = 0
*/
    IF rc <> "0" THEN DO
      say "No project loaded! Exiting ..."            /* Failed, exiting ...
*/
      exit
    END
   END

  SetInfoText("Printing frame information ...")     /* Set descriptive info
text */

  frames=GetProjectInfo("FRAMES")
  i=1; totalSize=0; totalTime=0;
  DO WHILE i <= frames                      /* Iterate from 1 to number of
frames */
   currentSize=GetFrameInfo(i, "SIZE")
   currentTime=GetFrameInfo(i, "TIMECODE")
   totalSize=totalSize + currentSize        /* Sum up size for statistics */
   totalTime=totalTime + currentTime        /* Sum up time for statistics */
                                            /* Print the frame line */
   say "Frame" i ":: Imagesize:" currentSize ", Timecode:" currentTime "ms"
   i=i+1
```

```
END

SetInfoText("Printing statistics ...")

say ""; say "Statistics:";                /* Print statistics */
say "* Average image datasize per frame: " totalSize/frames "bytes"
say "* Average frames per second:" 1000/(totalTime/frames) "fps"
```

Other Sources of Information

- A good starting point for finding information about REXX on the World Wide Web is the IBM REXX page at http://www2.hursley.ibm.com/rexx/rexx.htm
- *The REXX Files*, a monthly column in OS/2 e-Zine! at http://www.os2ezine.com, covers the application of REXX to various programming problems.
- The Enterprise REXX web site at htpp://www.winrexx.com provides information on the version of REXX used in the Windows version of MainActor, as well as links to general REXX information.
- Dave Martin's REXX FAQ page can be found at http://www.mindspring.com/~dave_martin/RexxFAQ.html