◆ *First Hour - Introduction to REXX*

What Is This Thing Called Rexx?

What is Rexx? Why does it exist? Why should I want to learn how to use it? All very good questions. Rexx (originally called REX) was developed by M. F. Cowlishaw of the IBM UK Scientific Centre beginning in 1979. It was first introduced into the VM/CMS environment in VM/SP Release 3, and was later introduced into other environments such as MVS/TSO, MS-DOS and now OS/2.

Rexx is available in OS/2 EE 1.2 and all releases of OS/2 1.3 and 2.0. You need to nothing special to take advantage of the Rexx language. Additionally, you can install Rexx on MS-DOS by using a product called Personal Rexx.

Rexx is a very powerful language. You can use for many programming tasks that heretofore required a BASIC or C program to perform. But at the same time, Rexx is an easy language to learn. Unlike compiler languages, Rexx is interpreted; meaning that changes you make to the program take effect immediately. There is no need to go through the compiler, link, execute, debug cycle. Rexx can be used for the very simple, to the most complex, application.

In order to remain easy for the non-computer specialist to program in Rexx, the following design goals were established:

- Ease of Use. Rexx uses natural, English-like words that do what they mean, such as **SAY**, **PULL** etc.
- Free Format. Rexx has few rules about formatting. Blank space is skipped and the user is free to format the program in any manner desired.
- Interpreted. Rexx is not compiled -- each statement is interpreted by OS/2 as it is executed. Allows for immediate feedback.
- Built-in Functions. Rexx has many built-in functions that perform operations for both numeric and character data.
- Parsing Capabilities. Rexx has powerful capabilities for the manipulation of character strings.
- Debugging. Rexx has meaningful error messages and a powerful trace facility.

From Batch To Rexx

A Rexx Program can do everything that a Batch Program can do. In this first section, we will begin by looking at how we do things in Batch Programs, and then how to do the same things in the Rexx language. In this way, we can start to convert our Batch Programs to Rexx Programs, or write new procedure programs in the Rexx language. Now, there would be no need for the Rexx language if all it could do was the same tasks as Batch Programs, and in the next section we will see how the Rexx language can be used to take us where no Batch Program has gone before.

Issuing Commands

Of of the more popular uses for the batch program is to execute a series of commands. Often you find yourself executing a number of commands, in the same order, over and over. In this case, you might create a batch program to do the same commands so that you can automate the task. For example, suppose you want to backup a number of files to floppy. You can put these commands into a file, and assign it the .CMD file extension, so that each time you want to backup your files, all you need do is enter the name of the batch program at the command prompt. In each of the example programs, the line numbers are included for notational convience only and should not be considered part of the example.

```
1 rem bkup.cmd
2 cd \work\docfiles
3 copy *.txt a:
4 cd \reports\xfiles\
5 copy *.txt a:
```

The same commands can be entered in a Rexx program, and by assigning the same extension to the file, .CMD, you can backup your work by entering the same command.

```
1 /* BKUP.CMD */
2 'cd \work\docfiles'
3 'copy *.txt a:'
4 'cd \reports\xfiles\'
5 'copy *.txt a:'
```

Immediately you will notice several similarities as well as several differences between the two files. You should recognize that the commands are identical, but that in the Rexx program we needed to put the commands into quotes. Why is this? For now, let it be sufficient to say that the Rexx language is far richer than the Batch program language. While the batch program merely passes his records on to OS/2 for processing, Rexx will attempt to interpret each line as it is executed. Only if Rexx does not understand the command, will it be passed on to OS/2. In this example here, both the backslash and the asterisk mean something special to Rexx, and so Rexx will try to perform some processing before he passes the command on to OS/2. Since we don't intend for Rexx to process these special characters himself, and indeed syntax errors would be the result, we must put the commands in quotes. By doing so, we have created a single ***token*** on each line called a ***string***. Since this token means nothing special to Rexx, he passes the command (minus the quotes) on to OS/2 for processing. The quotes may be either single or double quotes, it makes no difference to Rexx.

Also note that the first line of the Rexx program contains characters between the two special characters **`/*`** and **`*/.`** These characters mean something special to Rexx and denote a ***comment***. Any characters between the **`/*`** and the **`*/`** are considered a comment and are discarded by the Rexx processor. In this case, the comment is NOT superfluous; it has a very important role to play. Since both the Batch Program and the Rexx program have the .CMD extension, OS/2 needs a means to determine if the file should be handled by the Batch Processor or the Rexx Processor. OS/2 considers the file to be a Rexx Program, if the first record contains a comment that begins in the first column. Thus, all Rexx programs must begin with a comment. The comment can even be blank, that is **`/**/`**.

Displaying Messages

Let us expand our example above. Before we can back up our files, we must have a disk in drive A:. We want to put a message into the Batch or Rexx Program to remind us to do so. In the Batch Program we would use the

ECHO command to display messages on the screen; so our program might now look like this:

```
1 echo Put a disk in drive A
2 echo.
3 cd \work\docfiles
4 copy *.txt a:
5 cd \reports\xfiles\
6 copy *.txt a:
7 echo Backup is complete on drive A
```

Now we have a few bugs to work out yet (for example, the program doesn't stop and wait for us to put the disk in the drive), but we will get to these in a moment. Line number 2 uses ECHO followed by a period to print a blank line.

You may be familiar with another use for **ECHO** that takes either **ON** or **OFF** as a parameter. When **ECHO** is **ON** (the default), OS/2 will display each batch command at a command prompt as it is executed. There are two ways to suppress this. The first is to put the command **ECHO OFF** into the batch program as the fist command. However, this still causes the command **ECHO OFF** to be displayed by OS/2 when it is executed. The second means to suppress the display of the command is to precede the command name with an at sign. For example, **ECHO OFF** should be coded as **@ECHO OFF** to prevent any commands from being displayed. Our example now becomes (remember, we will fix the other bugs later):

```
1 @echo off
2 echo Put a disk in drive A
3 echo.
4 cd \work\docfiles
5 copy *.txt a:
6 cd \reports\xfiles\
7 copy *.txt a:
8 echo Backup is complete on drive A
```

In the Rexx language, the **ECHO** command is replaced by the **SAY** ***instruction***. The same program written in Rexx would look like this:

```
1 /* BKUP2.CMD */
2 '@echo off'
3 say 'Put a disk in drive A'
4 say
5 'cd \work\docfiles'
6 'copy *.txt a:'
7 'cd \reports\xfiles\'
8 'copy *.txt a:'
9  say 'Backup is complete on drive A'
```

Again, we note that the first record of the program must be a comment, and that the commands are coded as *strings*, and thus lines 2 and 5-8 contain a single Rexx ***token***. We still need to execute the **ECHO OFF** command to prevent OS/2 from displaying the four commands in lines 5-8 as they are executed. Again, we use a string for this to prevent the Rexx interpreter from trying to interpret the line. Lines 3, 4 and 9 show the use of the Rexx **SAY** instruction. Both lines 3 and 9 contain two *tokens*; an *instruction* and a *string*. The **SAY** instruction is not in quotes because **SAY** means something special to the Rexx interpreter. When Rexx encounters the **SAY**

instruction, it will display remaining tokens in the ***clause*** on the console. A Rexx *clause* is made up of zero or more tokens on a single line, thus the two tokens in line 3 make up a single Rexx clause. In short, line 3 displays the message 'Put a disk in drive A' on the console.

In lines 3 and 9 we have chosen to use Rexx strings, and put the message into quotes. It is not absolutely necessary to do so. The `SAY` instruction will display any number of tokens on the console. If instead we coded line 3 as: **`say Put a disk in drive A`** we now have a clause with, not two tokens, but seven. This is not an error, but Rexx will attempt to interpret each of the tokens **`Put, a, disk, in, drive`** and **`A`** before it is displayed. Since none of these tokens is special to Rexx, they are merely interpreted to be themselves translated to upper case. Therefore, this clause will display 'PUT A DISK IN DRIVE A' on the console. (I'll tell you a secret now. These six tokens are actually ***variables*** in the Rexx language. We will learn about variables later, but for now just remember that the initial value for variables is the name of the variable in uppercase).

Pausing Execution

Our example program so far has a few flaws. It displays the message to put a disk in drive A, but doesn't wait for us to do so. In the batch language we use the **`PAUSE`** command to stop the batch program from running momentarily.

```
1 @echo off
2 echo Put a disk in drive A
3 pause
4 cd \work\docfiles
5 copy *.txt a:
6 cd \reports\xfiles
7 copy *.txt a:
8 echo Backup is complete on drive A
```

Here we have replaced the second **`ECHO`** command (the one that displays the blank line) with the **`PAUSE`** command. The OS/2 pause command stops the execution of the batch program and displays the message: `Press any key to continue . . .` on the console.

It is quite permissible to use the **`PAUSE`** command in Rexx (putting quotes around the command, of course). However, you can emulate the pause action using native Rexx instructions; and even customize the message. Here we see the same program in Rexx.

```
1 /* BKUP3.CMD */
2 '@echo off'
3 say 'Put a disk in drive A'
4 say 'and press Enter when ready'
5 pull .
6 'cd \work\docfiles'
7 'copy *.txt a:'
8 'cd \reports\xfiles'
9 'copy *.txt a:'
9 say 'Backup is complete on drive A'
```

We see the use of another Rexx instruction, **`PULL`**. **`PULL`** causes the Rexx program to stop and wait for the user to enter something at the keyboard before continuing. The `PULL` instruction must be followed by one or more tokens that are known as the ***Parse Template***. While parse templates can be quite complex, we see the simplest here; the single period. The `PULL` instruction followed by the single period causes whatever the user enters at

the keyboard to be discarded. Since we only care about the user pressing the ENTER key, we use the period here as a "don't care" template. The only way to continue execution from a **PULL** instruction is by pressing enter.

NOTE: Like the batch program, the Rexx program can be terminated by pressing Ctrl-C or the Break key; except when executing the PULL instruction. If the Rexx program is halted due to a PULL instruction, the Ctrl-C or Break must be followed by the Enter key. Execution of the Rexx program will then be terminated.

Including Remarks

We have already seen the use of comments in the Rexx program. Comments take the place of the **REM** statement in the batch program. It is always a good idea to comment your programs so that others can understand them (or even so you can understand them when you look at them six months later). Unlike the **REM** command, a Rexx comment can span more than one record and can be included anywhere in the record, even in the middle of the clause (though this is *not* recommended).

Let us add some comments to the Rexx program we have been developing. Also, let's simplify the program so that we eliminate the Change Directory statements.

```
 1 /* BKUP4.CMD
 2    Backup daily working files to diskette at the end of the day
 3 */
 4 '@echo off'
 5 /* Clear the screen and remind user to put a disk in */
 6 'cls'
 7 say 'Put a disk in drive A'
 8 say 'and press Enter when ready'
 9 pull .
10 'copy \work\docfiles\*.txt a:'  /* Backup working doc files */
11 'copy \reports\xfiles\*.txt a:' /* Backup daily report files */
12 say 'Backup is complete on drive A'
```

So, we now have a complete working Rexx program to backup our working files every day. All we need to do now is remember to run it -- and I'm afraid that Rexx can't help you there.

Running one program from another

One Batch Program can be run from another Batch Program by using the **CALL** command. By including the name of the second Batch Program on the **CALL** command in the first Batch Program, control will be passed to the second program and return to the next statement in the first program. Here is a (useless) example where one program executes a second.

```
1 rem primo.cmd
2 @echo off
3 echo This is the first batch program
4 call secundo
5 echo Back in the first batch program
```

The same process is used with Rexx Programs. The **CALL** command must be used to execute the subordinate Rexx program. It is important that the **CALL** command be included in the quoted string to avoid confusion with

the **CALL** Rexx instruction -- which we will look at later.

```
1 /* PRIMO.CMD */
2 '@echo off'
3 say 'This is the first batch program'
4 'call secundo'
5 say 'Back in the first batch program'
```

You may recall that a Batch Program can be executed from another batch program just by using the name of the second Batch Program. In this case, when the second program completes, control is not passed back to the calling program. This is called ***chaining***. Chaining is not possible from Rexx Programs. You must use the **CALL** command, and control must return to the calling program.

There are other ways to execute Rexx Programs from within other Rexx Programs. We will look at these in the next section.

Using Replaceable Parameters

It is possible to pass parameters to a Batch Program from the command line. For example, the following two commands could be used in a Batch Program to move a file from the current subdirectory to a GAMES directory.

```
copy kye.exe c:\games
del kye.exe
```

We would like to make this batch file more useful by allowing us to pass the name of the file and the name of the target subdirectory to the Batch Program. We do this by using replaceable parameters, **%0** through **%9** in the Batch Program. **%1** will be replaced by the first parameter, **%2** the second, and so on. Therefore, our example program becomes:

```
1 rem moveit.cmd
2 @copy %1 %2
3 @del %1
```

Here, we've suppressed the display of the command by using the at sign before each command. Now we can move a file by typing `MOVE KYE.EXE C:\GAMES`. (NOTE: OS/2 has a MOVE command. We could use that command, but would rather build our own for exercise).

The Rexx language has a much more powerful parameter handling ability. Rather than use the predefined names **%1..%9**, Rexx allows use to use meaningful name, or *variables*, for each of the command line parameters. We need to look at the Parse Template again.

```
1 /* MOVEIT.CMD Usage: MOVEIT file.ext subdir */
2 arg fileid subdir .
3 '@copy' fileid subdir
4 '@del' fileid
```

Well, that seems to be a much clearer program to understand. Even without the comment you should probably be able to figure out what this Rexx Program does, and what parameters must be passed. Let us break down each of the lines.

Line 2 shows us the equivalent of the Batch Replaceable parameters. Rexx uses the **ARG** instruction to indicate input from the command line. Like **PULL**, **ARG** requires a Parse Template. This example shows a simple parse

template that consist of three tokens. The first two tokens give the names of Rexx ***symbols*** that are to be assigned the values of the first two arguments on the command line. A *symbol* becomes a Rexx ***variable*** when it has a value assigned to it. Therefore, if we invoke the Rexx Program with the command line `MOVEIT KYE.EXE C:\GAMES`, the variable **`fileid`** takes the value '`KYE.EXE`' and the variable **`subdir`** takes the value `C:\GAMES`. The third token is the single period again that we saw previously with the **`PULL`** instruction. The period is here to 'catch' any extra parameters the user might have included on the command line. It is always a good idea to include the period at the end of the parse template, because if Rexx runs out of tokens in the parse template before he runs out of parameters to assign, he will assign all the remaining tokens to the last symbol. For example, if we omit the period from the parse string, and the use enters `MOVEIT KYE.EXE C:\GAMES FOO`, then **`fileid`** takes the value `KYE.EXE` as before, but **`subdir`** now has the value `C:\GAMES FOO`, clearly not what we intended. The next section has more on parse templates.

Lines 3 and 4 show the **`COPY`** and **`DEL`** commands as before, except this time the tokens **`fileid`** and **`subdir`** are not part of the quoted string. Line 3 has three Rexx tokens, so Rexx will try to interpret each token from left to right. The first token is a string, so Rexx leaves it alone. The next token is the symbol **`fileid`**. Because the symbol **`fileid`** has a value assigned to it as the result of the **`ARG`** instruction, it is treated as a variable, so the value is substituted for the symbol. The same is true for the symbol **`subdir`**. After this substitution we have `@copy KYE.EXE C:\GAMES`. Since this doesn't mean anything to Rexx, he passes it along to OS/2 for execution.

If you were to remove the at signs from the commands you would notice that the fileid and subdirectory names echoed before execution would be in upper case, regardless of how they where typed on the command line. Both **`ARG`** and **`PARSE`** uppercase before assigning values to symbols in the parse template. Later we will see how to preserve case.

Program Flow and Conditional Execution

There is a major problem with the above example. If for some reason the **`COPY`** command fails (say the `C:\GAMES` subdirectory does not exist), the **`DEL`** command will still execute -- leaving you scrambling for your backup diskettes. It is necessary to check on the success or failure of the **`COPY`** operation before proceeding to annihilate the data file. In the Batch Program, we do this be means of the **`IF`** command and the **`ERRORLEVEL`** number. **`COPY`** will return an errorlevel greater then zero if the operation fails. We can check for this, and only execute the delete command if **`ERRORLEVEL`** is zero.

```
1 rem moveit2.cmd
2 @copy %1 %2
3 @if errorlevel 1 goto end
4 @del %1
5 :end
```

So now, if the **`COPY`** operation fails, we skip the **`DEL`** operation. The **`IF`** command causes the **`GOTO`** to be executed if the previous command returns an errorlevel *greater than or equal to* 1. There is no way to test for specific errorlevels.

The Rexx language has a special variable **`rc`** that works in the same manner as errorlevel. The variable **`rc`** contains the return value of the previous command. We can check the value of **`rc`** in a similar manner to

errorlevel.

```
1 /* MOVEIT2.CMD */
2 arg fileid subdir .
3 '@copy' fileid subdir
4 if rc \= 0 then signal end
5 '@del' fileid
6 end:
```

EGAD! That sure looks different from the Batch Program. If we look at the new line (line 4) we see **RC** has taken the place of errorlevel -- but then we have some funny hieroglyph after that (**\=**). In Rexx, we use a set of *comparative operators* to test values of variables. This operator stands for NOT EQUAL TO. The backslash represents NOT. We could also have used the GREATER THAN **(>)** operator. Comparative Operators are discussed more fully in the next section, but for now it's important to note that comparative operators result in a value of 1 (for true) or 0 (for false). Note the use of the Rexx instruction **THEN**. In an **IF** clause, the compared (the stuff being compared) can be quite complex. The **THEN** instruction is *required* to indicate the introduction of the clause to be executed if the **IF** clause evaluates to TRUE. Here, we are checking IF RC NOT EQUAL TO ZERO, do something.

Do *something*? What? **SIGNAL**? The **signal** instruction is equivalent to the Batch Program **GOTO** statement. It causes control to be passed to the label specified. The bad news is that use of the **SIGNAL** instruction is considered bad form. With the Rexx language, it is usually always possible to write a program so that the use of **SIGNAL** is not required. The above program could be written

```
1 /* MOVEIT2.CMD */
2 arg fileid subdir .
3 '@copy' fileid subdir
4 if rc = 0 then '@del' fileid
```

which is much preferred. It is possible to execute more that a single instruction with the **IF** clause, but, you guessed it, that is covered in the next section.

Beyond Batch

If that's all Rexx could do, there would be no need for it. But we have only started. You have enough now to rewrite most of your BATCH Programs in Rexx line for line. But there is much more in Rexx. We now go beyond the Batch Language and explore some of the more advanced elements of the Rexx language.

Symbols

Groups of any characters, selected from the set of alphanumeric characters and some special symbols, are called *symbols* in Rexx. Rexx does not distinguish between upper- and lower-case characters in symbols. What a Rexx symbol stands for is implied by the context in which it appears. A symbol can be a Rexx *keyword* (or *instruction*), a *function* or a *variable*.

- Constant Symbols Any symbol that starts with a digit is a constant symbol. As the name implies, these symbols can not be assigned to. Note that a constant symbol may be a

number, but is not necessarily a number.

```
1 /* CONSTANT.CMD - Examples of Constant Symbols */
2 say 77 ;
3 say 154.45 ;
4 say 12e5
5 say 3D ;   /* A constant symbol that is not numeric */

[C:\]constant
77
154.45
12E5
3D
[C:\]
```

- Simple Symbols A symbol that does not contain a period and does not start with a digit is a simple symbol. By default, a simple symbol represents the characters of the symbol translated to upper case. A symbol can be assigned a value, at which point it becomes a variable, which is discussed in the next section.

```
1 /* SIMPLE.CMD - Examples of Simple Symbols */
2 say GEORGE
3 say WhatAGoodExample /* Same as WHATAGOODEXAMPLE */
4 say $12 /* Not valid in OS/2 Rexx */

[C:\]simple
GEORGE
WHATAGOODEXAMPLE
<error message>
[C:\]
```

- Numbers A constant symbol that is comprised of digits. A number may optionally have a leading plus or minus sign and a decimal point. A number my have the character E (or e) followed by one or more digits representing a power of the exponent. Numbers may contain leading and trailing spaces. Numbers may also be represented by literal strings following the same rules. Note that literal strings numerics that end with 'x are taken to be

hexadecimal number and may contain the characters A-F.

```
1 /* NUMBERS.CMD - Examples of valid numbers */
2 say 15 ;
3 say -12.4 ;
4 say 126.0375 ;
5 say 73E+143 ;
6 say ' + 4.9e5' ;
7 say 'AF0d'x ;

[C:\]numbers
15
-12.4
126.0375
73E+143
 + 4.9E5
<funny hex stuff>
[C:\]
```

Variables

A variable is any symbol that is not a constant symbol that has been assigned a value.

- Uninitialized. By default, a simple symbol will evaluate to the characters that comprise the symbol, in uppercase. Note that technically, such as symbol is not a variable, but is often referred to as a variable for convenience sake.

- Symbols are variables. If a value has been assigned to a simple symbol, then the symbol will evaluate to the value that is assigned. In other words, the symbol becomes a variable by the act of assigning it a value. Variables can become simple symbols (uninitialized) by using the `DROP` instruction.

```
1 /* SYMBOLS.CMD - Symbols are variables */
2 say name bark
3 name = 'Snoopy'   /* This is an assignment */
4 bark = 'Woof Woof'
5 say name bark
6 drop bark
7 say name bark

[C:\]symbols
NAME BARK
Snoopy Woof Woof
Snoopy BARK
[C:\]
```

- Stems A Stem is a symbol that contains a single period as the last character. It may not start with a digit. As with all symbols, its default value is the characters that comprise the symbol, in uppercase.

If a stem symbol is used as the target of an assignment, then all possible Compound Symbols (see below) that have the same stem will be given that value. A reference to any Compound Symbol with that stem will evaluate to this value until either the Stem is assigned a new value or a individual Compound Symbol is assigned.

- Compound A Simple Symbol becomes a Compound Symbol if the symbol contains at least one period and the last character is not a period. The symbol may not start with a period. A compound symbol may be assigned a value, at which time it becomes a variable. Before it is assigned, its default value is the same as the simple symbol; that is the characters that comprise the symbol, in upper case. Compound Symbols are often used for arrays. The compound symbol consists of the Stem and the ***Tail***. The tail may be composed of Constant Symbols or Simple Symbols. If the tail is a variable, the variable is substituted before the compound variable is evaluated.

    - Subscripts Numeric Tails that are numerics or variables that contain numeric values are often used as arrays in Rexx. The tail can be used to index any element.

```
1 /* STEMS1.CMD - Example of numeric stems */
2 stem. = 'BOB'          /* All stems are BOB */
3 say stem.1 stem.2      /* says BOB BOB */
4 x = 2
5 stem.x = 'CARROLL'     /* Assign specific var*/
6 say stem.1 stem.x      /* says BOB CARROLL */

[C:\]STEMS1
BOB BOB
BOB CARROLL
[C:\]
```

    - Subscripts Non-Numeric Tails need not be numeric and variables that evaluate to non-numeric can be used.

```
1 /* STEMS2 - Examples of non-numeric stems */
2 foo.bar = 2
3 foo.2 = 10
4 say foo.bar     /* says 2 */
5 bar = 2
6 say foo.bar     /* says 10 */
7 say foo.'bar'   /* says foo.bar */

[C:\]
2
10
FOO.bar
[C:\]
```

Continuation Characters

If the last non-comment token on a record is a comma, the comma is replaced by a blank and the clause is assumed to continue on the following non-blank line. The comma acts as the continuation character.

```
1 /* LONGLINE.CMD - Example of the Continuation Character */
2 say 'Long lines can be continued by' ,
3     'using a comma at the end'

[C:\]longline
Long lines can be continued by using a comma at the end
[C:\]
```

Operators

Operators may be subdivided into three groups:

Concatenation

These operators are used to combine two strings to form one string by appending the second string to the end of the first.

- With Blanks. One or more blank characters is the simplest operator in the Rexx language. It causes two strings to be concatenated with one blank between them.
- With the Operator. The concatenation operator `(||)` can be used to concatenate two string with no blank between them.
- Abuttal. If a literal string and a symbol are abutted, they will be concatenated as if the Concatenation Operator had been used. For example, if `TOTAL` had the value `37.5`, then `TOTAL'%'` would evaluate to `37.5%`, the same as `TOTAL||'%'`.

```
1 /* CONCAT.CMD - Concatenation Examples */
2 say 'With'     'Blanks'
3 say 'With the Cancat' || 'enation operator'
4 foo = 'With Abut'
5 say  foo'tal'

[C:\]concat
With Blanks
With the Concatenation operator
WithAbuttal
[C:\]
```

Arithmetic

Characters strings that evaluate to valid numbers may be operated on by using one of the arithmetic operators.

- Addition, Subtraction, Multiplication and Division. Are, respectively, `+`, `-`, `*` and `/`
- Integer Division (`%`). Divide and return the integer part of the result.
- Remainder (`//`). Divide and return the remainder. Note this is not the same as modulo, as the result my be negative (thus `3 ** 0.5` is invalid).
- Power (`**`). Raise the number to a whole number power.
- Prefix plus and minus.

```
 1 /* OPERATOR.CMD - Examples of Rexx Operators */
 2 say "3+4="                3 + 4
 3 say "7-'4'="              7 - '4'
 4 say "4 * 3="              4 * 3
 5 say "5/3="                5 / 3
 6 say "1.4 + 0.3="          1.4 + 0.3
 7 say "5%3="                5 % 3
 8 say "5//3="               5 // 3
 9 say "-5//=3"              (-5 // 3)
10 say "3**2="               3 ** 2
11 say "12>5="               (12 > 5)
12 say "15<10="              (15 < 10)
13 say "'bark' > 'bite'="    ('bark' > 'bite')
14 say "' 3' = '3'="         (' 3' = '3')

[C:\]operator
3+4= 7
7-'4'= 3
4 * 3= 12
5/3= 1.66666667
1.4 + 0.3= 1.7
5%3= 1
5//3= 2
-5//=3 -2
3**2= 9
12>5= 1
15<10= 0
'bark' > 'bite'= 0
' 3' = '3'= 1
[C:\]
```

Comparative

Comparative operators return the value of `1` for true and the value of `0` for false. In comparative operators, if both operands are numerics, then a numeric comparison is performed. Otherwise, the operands are treated as

character strings. The comparative operators are the traditional symbols. Note that the backslash (\) can be substituted for the logical not symbol if that character is not in the character set (which is true for OS/2).

In the case of character comparison, leading and trailing blanks are ignored and the shorter character string is padded with blanks to the length of the longer string. The comparison is case sensitive.

Interactive Input and Output

Rexx procedures have the ability to receive data from the keyboard and display data on the console interactively. It is possible to write interactive applications using the Rexx language. We have already seen use of the SAY, PULL and ARG instructions in the previous section. We will now extend our ability to handle input by looking at more closely at the Parse Template.

Parsing Templates for PULL and ARG

Both the PULL and ARG instructions are used to break up strings and assign values to variables (symbols). The only difference is the source of the string. For the PULL instruction, the string is read from the console (keyboard) and for the ARG instruction, it is read from the command line that was used to invoke the Rexx procedure.

The simplest parsing template is a list of variable (symbol) names. The string being parsed is broken into to words, delimited by blanks, and each word is assigned to a variable in sequence. If there are more words than variables to be assigned, the last variable will receive all the remaining words. If there are more variables that words to be assigned, unused variables will be assigned a zero length string (a NULL value). Before being assigned to the variables, the blanks are stripped from both ends of the word, except to the last variable. The last variable may contain both leading and trailing spaces. Note that user input in this example is denoted by the underlined text.

```
1 /* PARSE.CMD - Example of Simple Parsing */
2 pull w1 w2 w3  /* example Any Good News */
3 say w1          /* prints ANY */
4 say w2          /* prints GOOD */
5 say w3          /* prints NEWS */
6 pull w1 w2 w3   /* example Not yet */
7 say w3          /* print a NULL string */
8 pull w1 w2 w3   /* example Where are we going home */
9 say w3          /* prints WE GOING HOME */
[C:\]
Any Good News
ANY
GOOD
NEWS
Not Yet

When are we going home
WE GOING HOME
[C:\]
```

After parsing on blanks, the next easiest to use is a literal string in the template, which is called a ***pattern***. The literal string could be a single character, or a string. The text to be parsed is broken into two string at the point

where the text matches the pattern, and each half is then parsed based on the template to the left or right of the pattern. The pattern itself is removed from the string. An example should make this clear (Again, user input is underlined):

```
1 /* PATTERN.CMD - Parsing with a pattern */
2 pull w1 ',' w2  /* To be, or not to be? */
3 say w1          /* prints TO BE */
4 say w2          /* prints OR NOT TO BE? */
5 pull w1 ',' w2 w3 w4 /* same input */
6 say w2          /* prints OR */
7 say w4          /* print BE? */
[C:\]pattern
To be, or not to be?
TO BE
OR NOT TO BE?
To Be, or not to be?
OR
BE
[C:\]
```

On occasion, you have the need to parse a string and don't need every phrase or word assigned to a variable. The single period symbol can be used as a "dummy variable". It will act just like a variable, but the word that would have been assigned to this variable is discarded and no memory needs to be allocated. The place older is very useful at the end of a parsing template to remove the leading and trailing blanks from the last word.

```
1 /* DUMMY.CMD - Using placeholders in templates */
2 Arg first . third . fifth .
3 say third','first fifth

[C:\] dummy Wally Willisford Wainwrite went westward Wednesday
WAINWRITE,WALLY WESTWARD
[C:\]
```

PARSE ARG and PARSE PULL

We have seen in the previous examples that Rexx will uppercase before assigning values to symbols when the `PULL` or `ARG` instruction is used. Both `PULL` and `ARG` are a shortened form of the `PARSE` instruction because they are so often used. The `PARSE` instruction is used with a parse template to assign values to variables. `PULL` is a shortened form of `PARSE UPPER PULL` and `ARG` is a shortened version of `PARSE UPPER ARG`. The `UPPER` keyword is used with the `PARSE` instruction to cause Rexx to uppercase before assigning values to tokens. If we use `PARSE` without the `UPPER` keyword, the input string will be unchanged, and left in mixed

case.

```
1 /* PARSEPUL.CMD - Example of Parse Pull */
2 Parse upper pull string
3 say string
4 Parse pull string
5 say string

[C:\]parsepul
The rain in Spain
THE RAIN IN SPAIN
The rain in Spain
The rain in Spain
[C:\]
```

PARSE VALUE (expression) WITH

The `PARSE VALUE` instruction takes the result of an expression (such as a function call) as the source of its input string. We will look at an example of this in a moment.

PARSE VAR

The `PARSE VAR` instruction takes a variable as the source of its input string. Again, we will see this in the following example programs.

Parsing with positional patterns

A numeric can be used in the template to specify an absolute column in the string. Columns up to the specified column number (but not including it) will be parsed into the template to the left of the numeric. The string will continue to be parsed beginning with the specified column. If the column is less than the current column pointer, the remaining unparsed string will be used, and the column pointer reset to the specified column. Thus it is possible to backup in the string to be parsed.

Signed numerics can be used in templates to indicate movement relative to the current column pointer. The absolute column is calculated by Rexx by adding (or subtracting) the numeric from the current column pointer.

```
1 /*POS.CMD - Example of positional parsing */
2 parse value 'ABCDEFG' with w1 3 w2 +3 w3 4 w4
3 say w1          /* says AB */
4 say w2          /* says CDE */
5 say w3          /* says FG */
6 say w4          /* says BCDEFG */

[C:\]
AB
CDE
FG
DEFG
[C:\]
```

Parsing with Variable patterns

Finally, it is possible to parse with a literal string, even if that string is not known when the procedure is written. A pattern can be specified as a variable by putting the variable name in parentheses.

```
1 /* VARPARSE.CMD - Example of Variable patterns */
2 string = '/abc def/'
3 parse var string delim +1 string (delim) .
4 say string

[C:\]
abc def
[C:\]
```

More on the IF Instruction

Previously we saw the use of the `IF` instruction to conditionally execute a clause. In addition to the `THEN` clause, it is possible to code a `ELSE` clause that will be executed if the condition if false. Please note that in the `IF` construct, the expression (here x = 5) must evaluate to either 0 or 1. By the way, are you surprised that the last execution in the example returned FALSE? See if you can explain why this would be. Code this example to find the answer.

```
1 /* IF.CMD - Use of the IF instruction */
2 arg x
3 if x = 5 then say 'TRUE'
4          else say 'FALSE'

[C:\]if 5
TRUE
[C:\]if 10
FALSE
[C:\]if 5 10 15
FALSE
[C:\]
```

Much Ado about DO

Have no fear folks, the Rexx language has a powerful iterative instruction, the ubiquitous `DO`. There are several flavors of the `DO` instruction, we will look at only some of the more popular versions here.

Simple Do Group

The simple `DO` group is used to group instructions together to be executed as if they were a single instruction. The primary use of the Do Group is as the instruction to be executed as the result of a `THEN` or `ELSE` instruction

in an IF clause. The DO group is executed once. The DO group is terminated with the END instruction.

```
1 /* SIMPLEDO.CMD - Example of a DO Group */
2 Parse Arg name .
3 If name = '' then do
4    Say 'Please enter your name'
5    Parse Pull name .
6 End
7 Say 'Hello' name
@Example@[C:\] simpledo
Please enter your name
Abraham
Hello Abraham
[C:\] simpledo Daisy
Hello Daisy
[C:\]
```

Simple repetitive loops

A Do Group can be made to execute more than a single time by including a numeric *repetitor* after the DO keyword. The group will be executed that number of time. The repetitor may be an expression that evaluates to a whole number. For example, here is a DO group that displays a simple aphorism 7 times:

```
1 /* TOOLONG.CMD - Proof that I've been coding Rexx too long */
2 times = 7
3 do times
4   say 'Rexx is better than SEX'
5 end

[C:\] toolong
Rexx is better than SEX
Rexx is better than SEX
Rexx is better than SEX
Rexx is better than SEX
Rexx is better than SEX
Rexx is better than SEX
Rexx is better than SEX
[C:\]
```

Controlled repetitive loops

The controlled form specifies a *control variable* which is given an initial value before the execution of the DO group and is incremented by 1 until the final end value is reached. The DO Group is executed for each value.

Either the initial value or the end value or both can be an expression and they do not need to be whole numbers.

```
1 /* CONTROL.CMD - Controlled Do Loop */
2 x = 1
3 do z = x to 5
4    say z
5 end

[C:\]control
1
2
3
4
5
[C:\]
```

DOing things FOREVER

The iterative `FOREVER` causes the `DO` group to be executed forever; what else did you expect it to do? Naturally, there needs to be a method to break out of a `DO` loop such as this. You can, of course, use `RETURN`, but we will see in a moment a couple of other Rexx instructions that control breaking out of a `DO` loop.

```
1 /* FOREVER - Example of the FOREVER iterative */
2 do forever
3 say 'This is a silly loop!'
4 end                              /* Ctrl-C to the rescue! */

[C:\]forever
This is a silly loop!
This is a silly loop!
This is a silly loop!
^C
```

Conditional Phrases WHILE and UNTIL

The phrases `WHILE` and `UNTIL` can be used to modify the behavior of any iterative `DO` loop. Both `WHILE` and `UNTIL` take a conditional expression. `WHILE` will cause the loop to be executed as long as the expression

evaluates to TRUE. `UNTIL` will cause the loop to be executed as long as the expression evaluates to false.

```
1 /* WHILE.CMD - Example of the WHILE condition
2    NOTE The time(r) resets an internal elapsed
3    time timer.  time(e) retrieves this    value */
4 junk = time(r)
5 do while time(e) < 30
6 say 'Gonna do this for another'
7     30 - time(e) 'seconds'
8 end
@Example@[C:\]while
Gonna do this for another 29.5432274 seconds
Gonna do this for another 29.1343254 seconds
<etc.>
```

Conditional Phrases with WHILE or UNTIL

The Conditional Phrases WHILE and UNTIL can be used with controlled repetitive loop. The loop will execute until either the control variable exceeds its ending value, or the conditional expression becomes true.

ITERATE

The `ITERATE` instruction is used to transfer control to the end of a `DO` loop. The ending condition is evaluated and the loop executed again if the ending condition has not been met.

```
1 /* ITERATE.CMD */
2 do ix=1 to 10
3     say 'Always say this'
4     if ix < 3 then iterate
5     say 'but only this if x > 2'
6 end
@Example@[C:\]iterate
Always say this
Always say this
Always say this
Always say this
but only this if x > 2
<etc.>
```

LEAVE

The `LEAVE` instruction is used to transfer control out of a `DO` loop unconditionally.

```
1 /* LEAVE.CMD */
2 rep = ''
3 do forever
4     say 'Contiune? (Yes/No)'
5     pull rep .
6     if abbrev('YES', rep) | abbrev('NO', rep)
7           then leave
8     say 'Please enter YES or NO'
9 end

[C:\]leave
Continue (Yes/No)
Sure
Please enter YES or NO
NO
[C:\]
```

Basic Debugging Techniques

Rexx provides a mean of tracing execution to enable the user to interactively debug a program. In most cases, the `TRACE R` instruction (which stands for TRACE RESULTS) is used to display each clause before it is executed and the final result of its evaluation. For hard cases, the `TRACE I` (for Intermediate) instruction can be used. It works like `TRACE R,` but also displays the intermediate results of each expression as it is evaluated.

When Rexx encounters the `TRACE` instruction, it causes all subsequent lines encountered in the program to be display, in some format, as they are executed. The `TRACE` instruction is followed by a character that indicates the 'level' of tracing that is to be performed. `TRACE` is used primarily for debugging. In the following table, only the first letter of each argument needs to be specified.

TRACE All	When this clause is encountered, all subsequent clauses are displayed on the console as they are executed. The clauses are shown as entered, whithout variable substitution, and comments are not displayed.
TRACE Commands	All OS/2 commands are displayed as they are executed, both as entered in the program, and after variable substitution. If the command results in an error or failure, the return code is also displayed. No other clauses are displayed.
TRACE Intermediates	Traces all clauses as they are executed and displays all intermediate results. This produces a lot of debugging information, but can be useful for really gnarly problems.
TRACE Off	Nothing is traced
TRACE Results	All clauses are displayed before they are executed, along with the results of the execution. Variable assignemnt of PARSE clauses are also displayed. This is a very good way to start debugging.

Interactive Tracing

For those really gnarly problems, Rexx provides yet another level of tracing. One of the best features of the Rexx language is the ability to single step through a program as it is executing. Rexx includes a mechanism for interactively controlling the execution of a program. Changing the `TRACE` setting to one with a prefix of '?' (for example `TRACE ?All`) turns on interactive tracing, and informs the user that tracing is now active. The language processor will ignore further trace instructions, and will stop after each instruction is traced. When stopped, three actions are possible:

- Enter a NULL line. A null line (no spaces even) will make the program continue to the next statement. Repeatedly entering a null will step from statement to statement in the program, that is single-stepping.
- Entering an Equal Sign Causes the processor to re-execute the last clause traced. This is useful if an `IF` statement is about to take an incorrect branch. The value of a variable can be changed and the clause re-executed.
- Anything else. Will be treated like part of the Rexx program and be interpreted. No tracing takes place. Once interpreted, the processor again pauses. A `TRACE` instruction can be entered to alter the current trace, and it will take effect.

A numeric form of the trace instruction can be used to allow several clauses to be executed before a pause. The clauses will be traced. `TRACE n` will cause 'n' clauses to be traced before the process again stops. `TRACE -n` will cause 'n' clauses to be executed, without tracing, before the procedure again stops.

SET RXTRACE=ON

In OS/2, the environment variable RXTRACE can be set to ON to enable interactive tracing of a Rexx command file. When this variable is SET (and ON is the only allowable value), command files written in Rexx are traced as if they contain the `TRACE ?R` instruction. It can be turned off again by clearing this variable. (`SET RXTRACE=`).

◆ *Second Hour - Rexx in the OS/2 World*

Using Functions in Rexx

The Rexx languages has a rich set of built-in functions. A function (call) is a symbol or string that is immediately followed by a left parenthesis. Even if the function requires no parameters, the parentheses must be included. Rexx can be extended by developing internal or external function, that can be written in Rexx or a high-level language (development of high-level language functions is not discussed in this paper.).

Built-in Functions

There are a number of built-in functions that can be used in Rexx. These are documented in the Reference Guide. All functions return a value and thus must either be assigned to a symbol, or evaluate to a valid expression, such as an OS/2 command.

```
1 /* BUILTIN.CMD - Example of a built-in function */
2 say date()
3 now = time('C')
4 say now

[C:]BUILTIN
27 Jul 1992
12:23pm
[C:\]
```

Internal Functions

Additional functions can be written in the Rexx language and included within the Rexx procedure file. These functions are called in the exact manner as built-in Rexx functions. Functions that are contained in the same Rexx procedure file are defined in the same manner as a Rexx procedure, but must return a value on the RETURN instruction.

```
 1 /* ORDINAL.CMD - Example (incomplete) of an internal function */
 2 x = 2
 3 say ordinal(x)       /* says 2nd */
 4 say ordinal(10)
 5 return
 6
 7 ordinal:
 8    ord.  = 'th'
 9    ord.1 = 'st'
10    ord.2 = 'nd'
11    ord.3 = 'rd'
12    arg num .          /* Note use of ARG */
13    return num || ord.num

[C:\]ORDINAL
2nd
10th
[C:\]
```

External Functions

Addition functions can be written as Rexx procedure files and stored on disk. These functions can be called in the exact same manner as internal Rexx functions. The name of the procedure file (e.g. TIMEDIFF.CMD) is taken to be the name of the function and thus the user is limited to eight characters for a function name. (Unless HPFS is used). The Rexx procedure must return a value on the RETURN instruction. In the example above, the function could be in file ORDINAL.CMD.

It is also possible to write an external Rexx functions in a high-level language like C and store them in a Dynamic Link Library. While an example of this is beyond the scope of this paper, there are two external Rexx libraries that are of interest. The first is the RexxUtil, which offers the Rexx programmer many functions, primarily dealing with:

- OS/2 system level commands
- User and text screen input and output.
- OS/2 INI file input and output.

The second library contains routines to allow Rexx programs to communicate with Extended Services/2 3270 emulation windows using HLLAPI.

Before any external library can be used from Rexx programs, it must be registered to the system so that the Rexx processor knows in what dynamic link library a specific function is located. External Rexx functions are registered using the **RxFuncAdd** function. The **RxFuncAdd** routine requires three parameters; the name of the function being registered, the name of the dynamic link library in which the function resides, and the name of the routine in the library that handles the function. This name is often the same as, though not required to be, the function name. The next example shows how to register the **SysSleep** function that is located in the RexxUtil library. Once a function is registered, it is registered to the system and need not be registered again (unless it has been deregistered). The supplier of the dynamic link library should supply you with the necessary information to

register its functions. Many libraries provide a function that registers their remaining functions.

```
1 /* SNOOZE.CMD */
2 Call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
3 Call SysLoadFuncs
4 Call SysSleep 60
```

File Input and Output Statements

Yes, Rexx can do input and output to disk files. These three function are most often used to read and write disk files.

LINEOUT

The LINEOUT function is used to write a string to a disk file. If the file does not exist, it is created. If the file exists, data is appended to the end of the file. The function normally takes two arguments, the name of the files, and the string to be written. A third argument can be specified, and if specified must be 1 to indicate that the string should be written as the first position in the file. This has the effect of overwriting an existing file. LINEOUT is called with only the filename to close an open file (either input or output). If no filename is specified, then STDOUT, that is the console, is used.

Because LINEOUT is a function, it returns a value; specifically, the number of lines remaining to be writing after attempting to write the string. Therefore, LINEOUT returns a 0 if the write is successful, or 1 if it was not. Because a value is returned, LINEOUT must either be used in an expression, or be assigned to a variable.

NOTE: Rexx functions can also be CALLed as procedures, in which case they do not return a value (actually, they return the value in a special variable RESULT). CALLing a function rather than using the function as a function often obviates the need for a dummy variable.

LINEIN

The LINEIN function is the opposite of the LINEOUT function in that it reads lines from a text file. LINEIN takes from zero to three arguments; most often a single argument, the name of the input file. If the file name is not specified, STDIN, the keyboard, is used. The second argument is used to specify which line is to be read from the input file (if not specified, the next available line is read). The final argument must be 0 or 1 (the default is 1) and specifies the number of lines to read from the input file. A file can be opened by specifying 0 lines to be read.

LINEIN returns the line read from the file.

LINES

LINES is a function that returns 1, or true, if any more lines (or characters) remain unread in an input file. It returns 0, or false, when the file is at end of file. Thus, LINES can be used to control a DO loop that is reading

data from a file.

```
1 /REVERSE.CMD  Example of file input and output */
2 arg filein fileout .
3 call LINEIN filein,,0       /* Opens filein */
4 do while lines(filein)
5    text = linein(filein)
6    call LINEOUT fileout,reverse(text)
7 end
8 call LINEOUT fileout
9 call LINEOUT filein

[C:\]REVERSE poepoem.txt reverse.txt
[C:\]
```

PMREXX - Using Rexx on the Desktop

Most Rexx programs are run either in full-screen or windowed OS/2. It is possible to run a Rexx procedure from the desktop in a Presentation Manager session by using the PMREXX command. The PMREXX command gives you the following features:

- A window for Rexx output from:
  - The **SAY** instruction
  - Output and error messages from OS/2 commands
  - Rexx TRACE output
- An input window for :
  - The **PULL** and **PARSE PULL** instructions
  - Input to OS/2 programs and commands
- Rexx output browsing, scrolling and clipboard capability
- A selection of fonts for the output window
- Menu options to trace or end a Rexx program.

To run a Rexx program under the PMREXX interface, just add the PMREXX command in front of the command line. For example, **PMREXX FILES**. The output from this may result in more that a screen full of information. If executed from the command line, the directory listing will quickly scroll off the top of the screen. However, if

executed under the PMREXX interface, you can scroll the output area back and forth to see all of the information.

```
1 /* FILES.CMD - To be run under PMREXX */
2 do forever
3     say 'Please enter a file or directory name'
4     parse pull filename
5     if filename = '' then leave
6     '@dir' filename
7 end

[C:\]pmrexx files
```

One of the functions in the RexxUtil function package provides a means for displaying a message under Presentation Manager. It is also possible to solicit a response from the user by using a Presentation Manager Message Box. The **RxMessageBox** function only works when run under the PMREXX interface. The example show the use of the **SysLoadFuncs** helper function for registering all the routines in the RexxUtil library. The **RxMessageBox** returns a numeric value depending on which button the user selects. .

```
1 /* RXMSG.CMD - A fragment to verify replacement of file */
2 call RxFuncAdd 'SysLoadFuncs', RexxUtil , 'SysLoadFuncs'
3 call SysLoadFuncs
4 do
5     reply = RxMessageBox("Do you want to replace the file", ,
6                 "Replace File?", "YesNo", "Question")
7     if reply = 7 then return
8 end
```

The following table shows the possible returned values depending on the button pressed by the user.

OK Button	1	CANCEL Button	2
ABORT Buttton	3	RETRY Button	4
IGNORE Button	5	YES Button	6
NO Button	7	HELP Button	8
ENTER Button	9	error occured	-1

Managing the Desktop with Rexx

The RexxUtil external Rexx function package provides a number of functions that permit the Rexx programmer to manipulate objects on the desktop. It is possible to create folders, program objects, programs groups and a number of other objects as well. Enhancements since the general release of OS/2 also provide for destroying and changing objects on the desktop. These routines provide the user with a means to dynamically alter the desktop programatically. Because these functions are in an external function package, they must be registered to the system prior to use. The examples show the correct call to register the functions.

SysCreateObject

This RexxUtil function can be used to create various object, like folders, programs and shadows, in the Workplace Shell using Rexx.

```
result = SysCreateObject(classname, title, location, [setup], [duplicate])
```

Classname

The classname determines what kind of object is to be created in the shell. There are several classes that are predefined to the system, but the four most useful are listed in the following table. A complete list of available predefined objects can be found in the Presentation Manager Programming Reference Volume II, Chapter 9. You can, of course, create you own object classes and use them, but that is beyond the scope of this paper.

WPFolder	Use this class to create a new folder
WPProgram	To create a program object used to launch application programs
WPProgramGroup	To create a program group, which is also a container class. User Profile Management is an example of a program group
WPShadow	Create a shadow object of another object.

Location

The location on the desktop can be specified either as a descriptive path (for example OS/2 system Folder\System Configuration) or by using an ObjectId. Not all objects have ObjectId's and the ObjectId is assigned when the object is created. The following table shows some of the predefined objects that exist on my system. You should specify an ObjectId for any objects created from Rexx programs, since the RexxUtil function can only specify existing objects by objectid.

<WP_DESKTOP>	The Desktop	<WP_INFO>	Information Folder
<WP_TEMPS>	Template Folder	<WP_GLOSS>	Glossary
<WP_VIEWER>	Minimized Icon Viewer	<WP_TUTOR>	Tutorial
<WP_SHRED>	Shredder	<WP_CMDREF>	Command Reference
<WP_MINDEX>	Master Index	<WP_REXREF>	Rexx Language Reference
<WP_OS2SYS>	System Folder	<WP_DRIVES>	Disk Drives Folder
<WP_START>	Startup Folder	<WP_CONFIG>	System Setup Folder
<WP_CLOCK>	System Clock	<WP_KEYB>	Keyboard Settings Notebook
<WP_KEYB>	Keyboard Settings	<WP_INST>	Setup and Install
<WP_MOUSE>	Mouse Settings	<WP_DDINST>	Device Driver Installation
<WP_MIGAPP>	Migrate Applications	<WP_SOUND>	Sound Settings Notebook
<WP_SYSTEM>	Systems Setting Notebook	<WP_CNTRY>	Country Settings Notebook
<WP_FNTPAL>	Font Pallette	<WP_CLRPAL>	Color Palatte
<WP_SCHPAL>	Scheme Pallette	<WP_SPOOL>	Spool Settings Notebook
<WP_PROMPTS>	Command Prompts	<WP_TOOLS>	Productivity Tools

<WP_Nowhere>	Hidden Folder		

Setup Strings

Setup strings consist of zero or more field, in the format 'keyname=value', separated and terminated by semicolons. The keynames that are valid are dependent on the object being created. The following tables show some the keynames and values common for (a) all objects, (b) Folder objects and (c) Program objects.

This table shows the Keyname and values that are common for all objects in the system. Every keyname has a safe default and thus not all keys need to be specified. The default value is shown hilighted.

KEYNAME	**VALUE**	**DESCRIPTION**
ICONFILE	filename	Sets the object's icon.
TEMPLATE	YES	Create the object as a template
	NO	Object is not a template
NODELETE	YES	Object can not be deleted
	NO	Object can be deleted
NOCOPY	YES	Object can not be copied
	NO	Object can be copied
NOMOVE	YES	Object can not be moved
	NO	Object can be moved
NOSHADOW	YES	Can not create a shadow of the object
	NO	Object can have a shadow
NOTVISIBLE	YES	Object is not visible
	NO	Object is visible
NOPRINT	YES	User can not print the object
	NO	User can print the object
ICONRESOURCE	id,module	Set the object icon in a resource library
ICONPOS	x,y	Set the icon's position in the folder where x and y represent percentage coordinates of the parent folder.
OBJECTID	<name>	Sets a persistent name fo the object. The OBJECTID is used to reference the object from the Rexx routines, or to obtain the handle to the object.
NORENAME	YES	Can not rename to object
	NO	Object can be renamed
NODRAG	YES	Object can not be dragged
	NO	Object is draggable
VIWBUTTON	HIDE	Views have a hide button
	MINIMIZE	View have a minimize button
MINWIN	HIDE	View are hidden when minimized

	VIEWER	Views are put into the minimized window viewer when minimized
	DESKTOP	Views are place on the desktop when minimized
CONCURRENTVIEW	YES	New object is created when opened
	NO	Resurface existing object when opened
OPEN	SETTINGS	Open setting notebook by default
	DEFAULT	Open the default view when opened

In addition to the above keynames and values, the following table shows those keynames and values that are added when the object is a folder object.

KEYNAME	VALUE	Description
OPEN	ICON	Open icon view when the object is created.
	TREE	Open the tree view when the object is created
	DETAILS	Open the details view when object is created
ICONVIEW	s1[,s2,...sn]	Set icon view the specified style
TREEVIEW	s1[,s2...sn]	Set tree view to the specifeid style
DETAILSVIEW	s1[,s2..sn]	Set detail view to specified style
view styles	FLOWED	columnar list of items
	NONFLOWED	non-columnar list of items
	NONGRID	put 'em anywhere icons
	NORMAL	normal icon size
	MINI	min-icon size
	INVISIBLE	no icons
	LINES	use lines in the tree view
	NOLINES	no lines in the tree view
BACKGROUND	filename	sets the folder background to the bitmap specified by filename in the \OS2\BITMAP directory of the boot drive.

In addition to the keynames and values common for all objects (except as noted in the table), the following table show those keynames and values that are added for program objects.

KEYNAME	VALUE	Description
ASSOCFILTER	filters	Sets the filename filter for files assoicated with the program.
ASSOCTYPE	type	Sets the type of files associated with this program.
EXENAME	filename	The name of the program
PARAMETERS	params	The parameter list, which can include substitution characters
PROGTYPE	FULLSCREEN	OS/2 fullscreen session
	PM	Presentation Manager window
	SEPARATEWIN	WIN-OS2 windows session running in a separate VDM
	VDM	DOS full screen session
	WIN	Fullscreen WIN-OS2 session
	WINDOWABLEVIO	windowed OS/2 session
	WINDOWEDVDM	windowed DOSS session
	WINDOWEDWIN	windowed WIN-OS2 session
NOAUTOCLOSE	YES	Leaves the window open after the program ends
	NO	Closes the window when the program ends
STARTUPDIR	pathname	The working directory
VIEWBUTTON		not supported
WINDOWPOS		not supported

Duplicateflag

There are three possible actions to take if an attempt is made to create an object with an object-id that already exists. The action is specified by this parameter.

- FailIfExists - No object will be created if an object already exists with the same ObjectId. This is the default.
- ReplaceIfExists - If an object with the same ObjectId already exists, it will be replaced with the given object.
- UpdateIfExists - If an object exists with the same ObjectId, it will updated with the given information.

The example uses the SysCreateObject external function to define a new folder on the desktop, create a program object in this folder and then create a shadow of this object on the desktop.

```
 1 /* FLDSHAD.CMD - Builds a folder and add a program object to it */
 2 call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
 3 call sysloadfuncs
 4 classname = 'WPFolder'
 5 title     = 'Test Folder'
 6 location  = '<WP_DESKTOP>'
 7 setup     = 'OBJECTID=<TEST2_FOLDER>'
 8 call BldObj
 9 classname = 'WPProgram'
10 title     = 'SYSLEVEL-VIO'
11 location  = '<TEST2_FOLDER>'
12 setup     = 'OBJECTID=<TEST2_SYSLVIO>;' ||,
13             'EXENAME=\OS2\SYSLEVEL.EXE;' ||,
14             'PROGTYPE=WINDOABLEVIO;'
15 Call BldObj
16 classname = 'WPShadow'
17 title     = 'SYSLEVEL-VIO'
18 location  = '<WP_DESKTOP>'
19 setup     = 'SHADOWID=<TEST2_SYSLVIO>'
20 call BldObj
21 return
22 BldObj:
22    SysCreateObject(classname, title, location, setup, 'R')
23 return
```

SysSetObjectData

The SysSetObjectData RexxUtil function can be used to change the settings of a specified object, if the ObjectId (or fully specified path) is known. It can also be used to open or resurface an object.

ObjectId

This must be the ObjectId of an existing object. There are several preexisting object (such as <WP_OS2SYS>) or it can be an object that was created with the SysCreateObject function. See the discussion of ObjectId in the preceding section.

Setup String

A string that contains the data to be changed in the object. This is the same setup string that is documented above for SysCreateObject. Only the values that are to be changed need be specified. To open an instance of an object, specify OPEN=DEFAULT; for the setup string.

The example shows how to open the program object created in the previous example.

```
1 /* OPOPEN.CMD */
2 call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
3 call sysloadfuncs
4 objectid = '<TEST2_SYSLVIO>'
5 call SysSetObjectData objectid, 'OPEN=DEFAULT;'
```

SysDestroyObject

The SysDestroyObject RexxUtil function removes an object from the Workplace Shell. The only parameter is the ObjectId of the object to be destroyed; see the preceding section for a discussion of ObjectId. Yes, it is possible to destroy your desktop with this function.

The example shows how to remove the folder created in previous examples. What happens to the shadow when it's related program object is destroyed?

```
1 /* OPByeBye.CMD */
2 call rxfuncadd sysloadfuncs, rexxutil, sysloadfuncs
3 call sysloadfuncs
4 objectid = '<TEST2_FOLDER>'
5 call SysDestroyObject objectid
```

Rexx and 3270 HLLAPI

If you have Extended Services/2 1.0, you have all the tools necessary to use the Rexx language as a HLLAPI programming tool. The Rexx EHLLAPI dynamic link library (SAAHLAPI.DLL) is installed whenever 3270 or 5250 emulation is installed. The documentation describing the available functions is not installed by default, unless the API Data Structures are already installed. If you wish to install the Rexx EHLLAPI documentation without the API Data Structures, the document is found in the GRP_0584.ZIP file on Extended Services Disk 4.

The Rexx EHLLAPI document should be sufficient if you already have a knowledge of the EHLLAPI interface. More information about EHLLAPI programming can be found in the IBM OS/2 EHLLAPI Programming Reference (S01F-0297).

The EHLLAPI service are provided as external functions in a dynamic link library and thus must be registered to the system, with the RxFuncAdd routine, before use. The example shows the correct call to register the service.

This example uses EHLLAPI to get the time from a VM/CMS system. It could be used to set the clock on the Personal Computer.

```
 1 /* QTIME.CMD - Use EHLLAPI to get date and time from VM system */
 2 session = 'A'
 3 call RxFuncAdd 'HLLAPI','SAAHLAPI','HLLAPISRV'
 4 rc = hllapi('Connect', session) if rc \= 0 then do
 5     say 'Did not connect to host RC='rc
 6     signal quit
 7 end
 8 rc = hllapi('Wait') if rc \= 0 then do
 9     say 'Host keyboard locked.'
10     signal quit end
11 rc = hllapi('Sendkey', '@C')
12 rc = hllapi('Wait')
13 rc = hllapi('Sendkey', 'CP Query Time @E')
14 rc = hllapi('Wait')
15 pos = hllapi('Search_ps', 'TIME IS ', 1)
16  if pos=0 then do
17     say 'Host could not process QUERY TIME command'
18     signal quit
19 end
20 time = hllapi('Copy_ps_to_str', pos + length('TIME IS '), 8)
21 say time
22 rc = hllapi('Sendkey', '@C') rc = hllapi('Wait')
23 quit:
24 call hllapi('disconnect')
25 call hllapi('reset_system')
26return
```