OS/2 REXX:
From Bark to Byte

Document Number GG24-4199-00

December 1993

International Technical Support Organization
Boca Raton Center

Take Note!

Before using this information and the product it supports, be sure to read the general information under “Special Notices” on page xix.

First Edition (December 1993)

This edition applies to OS/2 2.1.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address given below.

An ITSO Technical Bulletin Evaluation Form for reader's feedback appears facing Chapter 1. If the form has been removed, comments may be addressed to:

IBM Corporation, International Technical Support Organization
Dept. 91J, Building 235-2 Internal Zip 4423
901 NW 51st Street
Boca Raton, Florida 33431-1328

Abstract

This document describes OS/2 REXX from a usage and application scenario basis. It includes OS/2 REXX interfaces to CM/2, DB2/2 and MMPM/2.

This document is intended for IBM system engineers, IBM technical advisors, IBM authorized dealers, IBM customers and others who require a knowledge of OS/2 2.1 REXX and its interfaces.

A working knowledge of OS/2 2.1 and REXX is assumed.

PS (336 pages)

Contents

Figures

Tables

Special Notices

This publication is intended to help customers and IBM technical professionals understand the capabilities and interfaces of OS/2 2.x REXX. The information in this publication is not intended as the specification of any programming interfaces that are provided by OS/2 2.x. See the PUBLICATIONS section of the IBM Programming Announcement for OS/2 2.x for more information about what publications are considered to be product documentation.

References in this publication to IBM products, programs or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program, or service is not intended to state or imply that only IBM′s product, program, or service may be used. Any functionally equivalent program that does not infringe any of IBM′s intellectual property rights may be used instead of the IBM product, program or service.

Information in this book was developed in conjunction with use of the equipment specified, and is limited in application to those specific hardware and software products and levels.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, NY 10577.

The information contained in this document has not been submitted to any formal IBM test and is distributed AS IS. The information about non-IBM (VENDOR) products in this manual has been supplied by the vendor and IBM assumes no responsibility for its accuracy or completeness. The use of this information or the implementation of any of these techniques is a customer responsibility and depends on the customer′s ability to evaluate and integrate them into the customer′s operational environment. While each item may have been reviewed by IBM for accuracy in a specific situation, there is no guarantee that the same or similar results will be obtained elsewhere. Customers attempting to adapt these techniques to their own environments do so at their own risk.

Any performance data contained in this document was determined in a controlled environment, and therefore, the results that may be obtained in

other operating environments may vary significantly. Users of this document should verify the applicable data for their specific environment.

Reference to PTF numbers that have not been released through the normal distribution process does not imply general availability. The purpose of including these reference numbers is to alert IBM customers to specific information relative to the implementation of the PTF when it becomes available to each customer according to the normal IBM PTF distribution process.

The following terms, which are denoted by an asterisk (*) in this publication, are trademarks of the International Business Machines Corporation in the United States and/or other countries:

Advanced Peer-to-Peer Networking	AIX
CUA	DATABASE 2
DB2	DB2/2
DISTRIBUTED DATABASE CONNECTION SERVICES/2	IBM
ISSC	Multimedia Presentation Manager/2
OS/2	OS/400
Personal System/2	Presentation Manager
SAA	WIN-OS/2
Workplace Shell	EHLLAPI

The following terms, which are denoted by a double asterisk (**) in this publication, are trademarks of other companies:

Amiga	DeskMan/2
GPF	1-2-3, Lotus, Freelance, Freelance Graphics
Hockware	VisPro/REXX
Watcom	VX-REXX

Preface

This document describes OS/2 REXX from a usage and application scenario basis. It includes OS/2 REXX interfaces to CM/2, DB2/2 and MMPM/2.

This document is intended for IBM technical professionals, IBM technical advisors, IBM authorized dealers, IBM customers and others who require a knowledge of OS/2 2.1 REXX and its interfaces.

A working knowledge of OS/2 2.1 and REXX is assumed.

How This Document is Organized

The document is organized as follows:

- Chapter 1, “Why REXX?”

  This is a comprehensive look at the advantages and disadvantages of the REXX language.

- Chapter 2, “OS/2 REXX Specifics”

  This chapter gives a brief overview to the features of OS/2 REXX and describes some of the major differences between OS/2 REXX and the other platforms.

- Chapter 3, “External Functions”

  This chapter focuses on external functions written in compiled languages that are accessible by REXX programs.

- Chapter 4, “ REXX Utilities External Function Package (REXXUTIL)”

  This chapter will focus on some of the most useful features in RexxUtils although some of the utilities are used in examples throughout this book.

- Chapter 5, “The Workplace Shell and REXX”

  This chapter takes a look at the REXXUTIL functions, and different ways that they can be used to manipulate the Workplace Shell.

- Chapter 6, “ REXX and C”

  This chapter will discuss in detail how to write C functions that are accessible by REXX, as well as an in depth look at how to call REXX functions from C programs.

- Chapter 7, “Multimedia REXX”

In this chapter the REXX interfaces to the MMPM/2 software application are explained in detail. Examples are provided.

- Chapter 8, “REXX Interfaces to CM/2 EHLLAPI”

  This chapter will focus on REXX EHLLAPI APIs and their interaction with 3270 sessions.

- Chapter 9, “REXX Interfaces to DB2/2”

  This chapter discusses how the DB2/2 interfaces can be used to create useful REXX programs. Example programs are provided and explained. The installation and setup required for REXX programs to access remote workstation databases is also provided.

- Chapter 10, “Visual REXX Builders”

  In this chapter we take an existing REXX application and convert it into a VX-REXX program, as well as a VisPro/REXX program.

Related Publications

The publications listed in this section are considered particularly suitable for a more detailed discussion of the topics covered in this document. A general listing of other REXX related documentation can be found in Appendix E, “Published Books, Manuals, and Papers on REXX” on page 309.

- *Procedures Language 2/ REXX Reference*, S10G-6268
- *Procedures Langauge 2/ REXX User's Guide*, S10G-6269
- *SAA CPI REXX Level 2 Reference*, SC24-5549
- *IBM Database 2 OS/2 Programming Reference*, S62G-3666
- *IBM Database 2 OS/2 SQL Reference*, S62G-3667
- *IBM Communications Manager/2 EHLLAPI Programming Reference*, SC31-6163
- *OS/2 2.1 Unleashed*, SR28-4318
- *The REXX Language: A Practical Approach to Programming*
- *VisPro/REXX for OS/2 2.x*
- *Watcom VX-REXX for OS/2 Programmer's Guide and Reference*

International Technical Support Organization Publications

A complete list of International Technical Support Organization publications, with a brief description of each, may be found in:

Bibliography of International Technical Support Organization Technical Bulletins, GG24-3070.

Acknowledgments

The advisor for this project was:

Jerry A. Stegenga II
International Technical Support Organization, Boca Raton

The authors of this document are:

Richard Rogers
IBM United States

Ilari Rönnberg
IBM Finland

This publication is the result of a residency conducted by the International Technical Support Organization at the Boca Raton Center.

Thanks to the following people for the invaluable advice and guidance provided in the production of this document:

Tim Sennitt, International Technical Support Organization, Boca Raton

Niels Gylling, IBM Denmark Responsor

Richard Kurtz, IBM Poughkeepsie

Mike Lamb, ISSC Kingston

Bret Curran, IBM Personal Systems Support Center Dallas

Juan van der Breggen, ISM South Africa

And the various contributors to the REXX Fora

Chapter 1. Why REXX?

REstructured eXtended eXecutor language (REXX) was developed with the intention of providing a language that makes programming easier. It is a language designed for the computer administrator, whether the administrator is an experienced programmer or just a beginner. Because of this, learning how to program effectively in the REXX language is considerably easier than in many other languages. No compiler is required. It is an interpreted language, meaning each step of the program is evaluated and then executed each time the program runs. REXX evolved out of the EXEC and EXEC 2 languages that provided a way to bundle Conversational Monitor System (CMS) commands together. REXX took that concept a step further by providing the CMS command interface along with the syntax and function of a more robust programming language. REXX was developed and used internally at IBM* for a few years before being made a part of the Virtual Machine/System Product (VM/SP) in 1983. In 1987 REXX was chosen by IBM to be the Systems Application Architecture (SAA*) procedural language, paving the way for the important role it has assumed in the Operating System/2* (OS/2*) product.

A strength of the REXX language is that it is very easy for you to write REXX programs that interface with the operating system that your program will be running on. REXX is compatible with the VM, Multiple Virtual Storage (MVS), Advanced Interactive eXecutive (AIX*), and OS/2 operating systems, among others. REXX has been a part of OS/2 since Version 1.2. If you have OS/2, then you have all that you need to write REXX programs. Since REXX is packaged with OS/2, there are many interfaces to OS/2 functions provided for REXX programming. Given that REXX is an inherent part of OS/2, it is a natural way to explore OS/2 function as well as products that integrate with OS/2.

The intent of this book is to focus on OS/2 specific features of the REXX language, and to provide information on how to use REXX to interact with OS/2 and some OS/2 compatible functions. This redbook is not a tutorial on how to program in REXX. The REXX syntax and command reference are provided in Appendix A, “REXX Syntax Diagrams” on page 225 as a convenience. There are many good books and technical references that are very helpful for the beginning REXX programmer. (See “Related Publications” on page xxii.) We have also included syntax and programming reference information for each of the OS/2 interfaces we focus on in this book.

1.1 Power of OS/2 2.1 REXX

Since REXX is a part of the OS/2 2.1 package, most of the features of OS/2 2.1 are accessible through REXX. For example, there are functions contained in the REXX Utilities package that provide for manipulation of the Workplace Shell*, desktop and file directories. OS/2 2.1 REXX has the capability of using dynamic link libraries (DLL) to interface with other software applications, for example Ultimedia*, Communications Manager for OS/2 (CM/2), and Database 2* for OS/2 (DB2/2*). All of these languages provide application programming interfaces (APIs) that allow REXX programs to interface with them. There are numerous Visual REXX products on the market that link REXX programs to OS/2 Presentation Manager* (PM). Because REXX is not compiled, the performance in terms of execution speed of a REXX program does not compare to compiled languages running under OS/2 2.1. However, it is obvious that REXX is a strategic part of the OS/2 world of products and is often the fastest way to create, or interface to, an application.

Here is a more comprehensive look at the advantages and disadvantages of the REXX language.

Advantages:

- System independent coding for:
  - VM
  - MVS
  - Operating System/400 (OS/400*)
  - Advanced Interactive Executive (AIX)
  - OS/2
  - Amiga**
- Character strings are the only data type. Handling depends on usage. For example, numbers can be manipulated by string operations:

```
Number = '5678'
LastOne = Substr(Number,4,1)
Say LastOne                    /* Will Display   8    */
```

  or the same number can be used for numeric calculation:

```
Number = '5678'
LastOne = Substr(Number,4,1)
Multiplied = LastOne * Number
Say Multiplied                 /* Will Display   45424 */
```

- Powerful string handling functions. For example:

```
TextLine = 'This is a line of text'
PartOfFourth = Substr(Word(TextLine,4),3,2)
Say PartOfFourth                /* Will Display   ne   */
```

- Associative arrays. For example:

```
Do I = 1 to 4
  Data.I = 'DATA 'I
End
Do I = 1 to 4
  Say Copies(" ",I) Data.I
End
/* Will say
 DATA 1
  DATA 2
   DATA 3
    DATA 4
*/
```

- Decimal arithmetic, precision decided by programmer
- Interpretive language for faster development
- Programming styles to fit anybody, from one line code to Common Business Oriented Language (COBOL) style
- Natural, english like syntax
- SAA language
- Future recognition by American National Standards Institute (ANSI) and the International Organization for Standardization (ISO)
- Access to operating environment commands
- REXX enabled products such as:
  - LOTUS** Notes (INTERFLOX)
  - LOTUS 1-2-3**
  - Personal Application System/2 (PAS/2)
  - Enhanced editor for PM (EPM)
- REXX API interfaces provided to environments such as:
  - DB2/2 Query Manager CM/2
  - Advanced Program-to-Program Communication (APPC)
  - Emulator High Level Language Application Programming Interface (EHLLAPI*)

  - Common Programming Interface for Communications (CPI-C)
  - Multimedia Presentation Manager/2* (MMPM/2)
- External function packages, existing and possibility to build your own
- Can replace command lists (batch languages) with structured approach
- Visual builders (VisPro/REXX** by HockWare**, VX-REXX** by Watcom**, GPF* * REXX tool by GPF) allow easy access to common user access (CUA*) objects and event programming
- Packaged with OS/2 and other operating systems, so no separate purchase necessary to start programming with REXX
- Evolving language, moving towards Object REXX

Disadvantages:

- Interpreted, not compiled for OS/2. Faster and easier development and version independence but slightly slower performance
- Differences between operating systems especially file I/O
- REXX data type, the string, not supported well in other languages thus requiring extra conversions when invoking programs or functions written in other languages

1.2 Example

Following is an example showing structured programming in REXX and the use of REXX variables in both string and arithmetic operations. The example is used for converting temperatures from Fahrenheit to Celsius and vice versa. The names of places in the example are purely coincidental and have only been selected to show selection logic in REXX.

All examples in this book are also included in the REXX Samples diskette shipped with the book.

1.2.1 Sample 1 FAH2CEL.CMD

```
/* Calculate Fahrenheit in Boca Raton to Celsius             */
/* or Celsius in Helsinki to Fahrenheit                      */
Do Until Pos(Ans,'CF') > 0 /* Loop until answer either C or F */
  'CLS'                    /* Clear the screen              */
  Say 'Do you wish to convert Celsius in Helsinki or'
  Say 'Fahrenheit in Boca Raton?'  /* Display question      */
  Say 'Answer C for Celsius of F for Fahrenheit'
  Pull Ans                 /* Get the answer from keyboard  */
  Ans = Translate(Ans,'CF','cf') /* Translate to uppercase  */
End
If Ans = 'F' then Call Fah /* Translate Fahrenheit to Celsius */
Else Call Cel              /* Translate Celsius to Fahrenheit */
Exit                       /* The Exit point for program      */
```

Figure 1. Example FAH2CEL.CMD Part 1

The first part of the program, Figure 1 provides screen output via the **Say** function as well as input via the **Pull** function. It uses the string function **Pos** to search if the keyboard input is valid and **Translate** to translate the input to uppercase. **CLS** is used to show how standard OS/2 commands can be used from within REXX. **Call** is used to show how separate procedures can be called within the program to provide a more structured approach. **Do Until** is shown as one way of using loops in a REXX program.

```
Fah:

Do Until Datatype(Fahrenheit) = 'NUM'
                          /* Do until datatype = numeric     */
  Say 'Give temperature in Boca Raton in Fahrenheit:'
  Pull Fahrenheit
  If Datatype(Fahrenheit) <> 'NUM' then Iterate
                          /* If not numeric, loop again      */
  Celsius = -32*5/9+Fahrenheit*5/9
/* 0 Celsius = -32 Fahrenheit                                 */
/* 1 Fahrenheit = 5/9*Celsius                                 */
  Select                  /* Select from various situations  */
    When Celsius < -273.15 then Do
      Say 'Impossible, below absolute freezing point!!'
      Say 'Otherwise answer would be:' Celsius
      Exit
    End
    When Celsius < 20 then Do
      Say 'Are you sure you are talking about Boca Raton'
      Say 'and not Helsinki or the North Pole?'
      Say 'This low temperature is:' Celsius
    End
    When Celsius > 90 then Do
      Say 'Say, are we talking about Venus here?'
      Say 'This temperature is:' Celsius
    End
    Otherwise             /* None of the above               */
      Say 'Temperature in Boca Raton in Celsius is:' Celsius
  End
End

Return
```

Figure 2. Example FAH2CEL.CMD Part 2, Function Fah

The second part of the program shows the use of **Iterate** to provide control inside a loop by going to the beginning of loop without executing the rest of the instructions within the loop. **Select** is used to show how REXX can be instructed to select an instruction from various alternatives. **Datatype** can be used to determine if the variable can be used as a valid numeric value. **Celsius = -32*5/9+Fahrenheit*5/9** shows how REXX can handle calculations with its string variable.

```
Cel:

Do Until Datatype(Celsius) = 'NUM'
  Say 'Give temperature in Helsinki in Celsius:'
  Pull Celsius
  If Datatype(Celsius) <> 'NUM' then Iterate
  Fahrenheit = 32+Celsius*9/5
  Select
    When Celsius < -273.15 then Do
      Say 'Impossible, below absolute freezing point!!'
      Say 'Otherwise answer would be:' Fahrenheit
      Exit
    End
    When (Fahrenheit > 60) & (Fahrenheit < 194) then Do
      Say 'Are you sure you are talking about Helsinki'
      Say 'and not somewhere south?'
      Say 'This High temperature is:' Fahrenheit
    End
    When Fahrenheit >= 194 then Do
      Say 'Say, are we talking about Venus here?'
      Say 'This temperature is:' Celsius
    End
    Otherwise
      Say 'Temperature in Helsinki in Fahrenheit is:' Fahrenheit
  End
End

Return
```

Figure 3. Example FAH2CEL.CMD Part 3, Function Cel

The third part is basically the second part reversed to provide a complete solution to a common question of calculating temperatures.

Chapter 2. OS/2 REXX Specifics

REXX, also called SAA CPI Procedures Language, has been standardized as the procedure language for the following platforms:

- VM
- TSO/E
- OS/400
- OS/2

Since REXX is tightly integrated with the operating system and utilizes the features provided by that environment, there are unique features in each of the REXXs depending on their specific environment. It is also possible to code additional functions to REXX as external function packages to provide system specific functions like object manipulation.

An example of this is the functions provided with OS/2 REXX REXXUTIL DLL. These functions are system specific and cannot be used on other platforms or when writing portable programs. They can, however, provide some powerful features for OS/2. These functions are more fully described in Chapter 4, " REXX Utilities External Function Package (REXXUTIL)" on page 53.

OS/2 is the only REXX platform that has full SAA Common Programming Interface (CPI) Procedures Language level 2 support to date. Level 2 support is described in *SAA CPI REXX Level 2 Reference*. OS/2 REXX commands also perform, in some cases, differently from what VM programmers are used to. This chapter gives a brief overview of OS/2 REXX features and describes some of the major differences between OS/2 REXX and the other platforms.

2.1 Calling from a REXX Procedure

In order to work efficiently with REXX under OS/2 some knowledge of particular OS/2 commands is needed. For example, DETACH and START can give your OS/2 REXX programs powerful multitasking capabilities unknown to most VM programmers. Having this ability to execute procedures and commands in separate sessions can, in many cases, increase the performance of your program.

For instance, an OS/2 REXX program can do screen input in one process, fetch data in another, and do calculations in the third. All this can be done in parallel sessions with REXX using the START or DETACH command.

2.1.1 The REXX Call Instruction

To call to another REXX CMD in OS/2 you must use the CALL instruction. For example, you have two REXX programs, MAIN.CMD and SUB.CMD, and MAIN.CMD calls SUB.CMD using the statement:

```
SUB
```

Now this statement would work in VM but it will give the following error message in OS/2:

```
SYS1803: Chaining was attempted from a REXX batch file.
```

Typing HELPMSG SYS1803 at an OS/2 command prompt will give you the following explanation:

```
SYS1803: Chaining was attempted from a REXX batch file.

EXPLANATION: CMD.EXE does not support chaining from REXX
batch files.
ACTION: Check your REXX batch file for other batch files
names. Precede the batch file name that you want started
with the REXX keyword CALL.
```

Using the statement:

```
Call SUB
```

however, will work in both environments.

Note

For the Call instruction the interpreter looks first for a corresponding label in your procedure. If no label is found, the interpreter then looks for a built-in function or a .CMD file with that name.

You cannot use Call to start OS/2 commands or EXE files. See the following sections on how to do this.

2.1.2 Calling OS/2 .EXE or Command Files

Because OS/2 commands and .EXE files are not OS/2 batch files they cannot be executed using Call. You can execute OS/2 commands or EXE files by one of the following methods:

- Use the command name (with or without quotes)

  For example:

  ```
  DIR
  ```

  or

  ```
  'DIR'
  ```

  This starts the command within the same session and processing will continue once the command has terminated. Some details about using just the OS/2 command are described in 2.1.2.1, "The OS/2 Command Name."

- Use the START command

  For example:

  ```
  START /F 'DIR'
  ```

  is will start a command in a new session and processing will continue immediately after the command is issued. Some details about the START command are described in 2.1.2.2, "The START Command" on page 12.

- Use the DETACH command

  For example:

  ```
  'DETACH DIR > OUT.FIL'
  ```

  This starts and simultaneously detaches an OS/2 program from its command processor. Some details about the DETACH command are described in 2.1.2.3, "The DETACH Command" on page 14.

2.1.2.1 The OS/2 Command Name

Starting OS/2 commands by simply using their names will provide a single tasking environment. Quotes, however, should be used to ensure that the command name has not been used previously as a variable. REXX will allow you to use variables as commands and does not reserve any names for system use. So the following:

```
DIR = 'TREE'
DIR
```

would execute the command TREE, not display a directory listing.

2.1.2.2 The START Command

The Start command is used to start OS/2 commands or DOS commands in a separate session. The complete description of the START command can be found in the *OS/2 Command Reference* or the online version which is located in the Information folder on the default desktop. Some useful parameters for the START command are included here.

/B Parameter

Makes the program the background session. For example:

```
START /B /C SAMPLE.CMD
```

will start the SAMPLE.CMD in an OS/2 window in a background session and after the .CMD is finished the window is closed. Note that if the /FS, /WIN or /PM parameter is specified, the program automatically becomes the foreground session and this parameter is ignored.

/C Parameter

Indicates to start the program indirectly through the command processor, CMD.EXE, and end the session when the command is complete. For example:

```
START /C SAMPLE.CMD
```

1. Will start a new command prompt.
2. Start SAMPLE.CMD.
3. Close the prompt after the .CMD file has finished.

This is useful if you don't want unnecessary command prompts started.

/F Parameter

Makes the program the foreground session. If this parameter is not specified, the program becomes a background session. For example:

```
START /F /C SAMPLE.CMD
```

will start the SAMPLE.CMD in an OS/2 window in a foreground session and after the .CMD is finished the window is closed. Note that if the /FS, /WIN or /PM parameter is specified, the program automatically becomes the foreground session.

/FS Parameter

Requests that an application be started in a full-screen session. This session will be started in the foreground. For example, to start the SAMPLE.CMD in an OS/2 full-screen session:

```
START /FS SAMPLE.CMD
```

/MAX Parameter

Start the command in a maximized window. If you do not give a command name as a parameter START will start a new CMD.EXE session. For example, to start a new OS/2 window in a maximized state in the foreground:

```
START /MAX /F
```

/MIN Parameter

Requests that a Presentation Manager (PM) or any windowed application starts in a minimized (icon) state. For example, to start the command minimized to an icon:

```
START /MIN SAMPLE.CMD
```

This has no effect if the /FS parameter is also used. Also, a PM application may choose not to honor this request.

/PM Parameter

Indicates that the program to be started is a Presentation Manager application. For example:

```
START /PM E.EXE
```

/WIN Parameter

Indicates that this is an OS/2 application that runs within an OS/2 or DOS window. For example:

```
START /WIN SAMPLE.CMD
```

The program to be started can also be given a descriptive name which will show in the Window List and on the Title Bar of the OS/2 or DOS window or full-screen and the icon like in Figure 4 on page 14 for the following command:

```
START "Sample application" /F /C SAMPLE.CMD
```

The order of parameters is also important. Issuing START /FS SAMPLE.CMD will start SAMPLE.CMD in a full-screen session, but START SAMPLE.CMD /FS will pass /FS as a parameter to SAMPLE.CMD and start it in a background window.

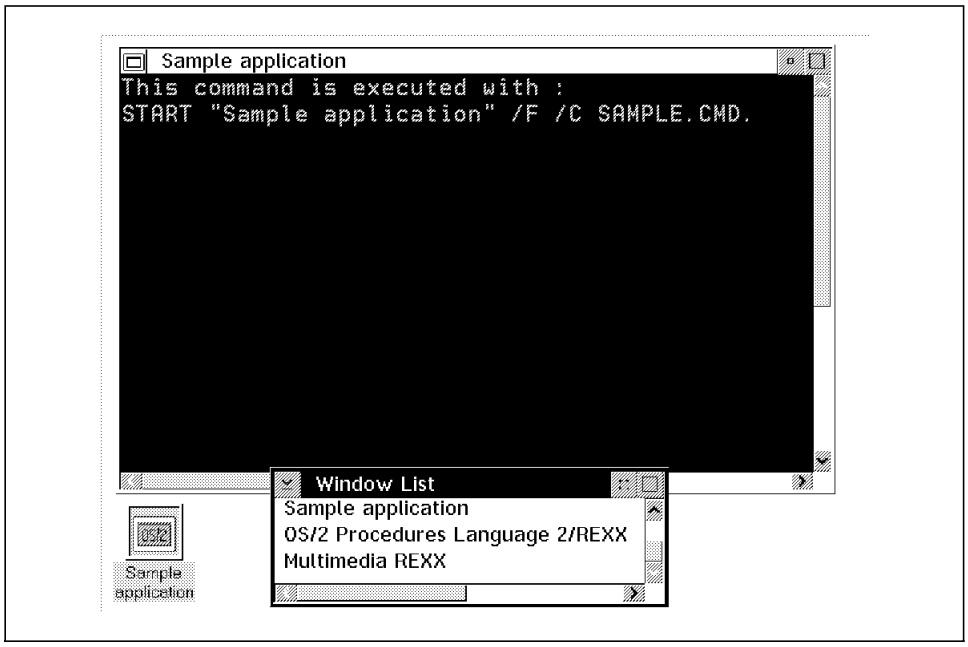


Figure 4. Screen Shot of SAMPLE.CMD Application

2.1.2.3 The DETACH Command

Any program that is started with DETACH must be able to process programs independently outside the control of the command processor. DETACH should not issue any input or output calls to the keyboard, the mouse, or the display. You can detach any program, command, or file that does not require the use of a screen. Examples of these are internal commands and batch (.CMD) files.

The OS/2 operating system detaches CMD.EXE when it runs the internal command or batch file. For example, if you type DETACH DIR, it is changed to the equivalent of DETACH CMD.EXE /C DIR. Detach will start a command in a new session and processing will continue immediately after the command is issued.

2.1.3 Multitasking with START and DETACH

The START and DETACH commands are very useful for providing multitasking capabilities to your program. The following figures provide an example of simple multitasking under OS/2 REXX with DETACH. More examples on multitasking using queues and START are provided in 2.3, "RxQueue" on page 25.

These figures show:

- The data file accessed by the program, Figure 5
- How to read data from the file, Figure 5
- How to do calculations and process a report in one process, Figure 7 on page 17
- How to print the reports one by one in the second process, Figure 8 on page 17
- The output from the program, Figure 9 on page 18

```
A COMPANY
A        1500   20      3
B        2500   30      0
C        1000   40      25
D        450    20      25
B COMPANY
A        1500   15      15
B        2500   10      10
C        1100   20      20
D        450    15      15
```

Figure 5. Data File Used for REXX Multitasking Example

```
/* Do calculations from values given in DataFile               */
/* Print in separate session through MULTIPRT.CMD              */
DataFile = 'SFIG1994.DAT'           /* Name of DataFile        */
Header = 'Sales figures for 1994,'  /* Heading text            */
Detail = 'PRODUCT  VALUE  AMOUNT  DISCOUNT  NET VALUE'
                                    /* Headings                */
Parse Value 0 0 0 0 With A B C D    /* Initialize values to 0  */
LineCount = 1                       /* Initialize LineCount to 1*/
Start = 1                           /* Start = TRUE            */
SubTot = 0                          /* Total for company       */
Do While Lines(DataFile) > 0        /* Read while lines in file */
  Line = Linein(DataFile)           /* Read line               */
  If Pos('COMPANY',Line) > 0 Then Do /* If heading line        */
    If \Start then Call PrintIt    /* Dont print on first line*/
  Out.LineCount = Header Line'*'Detail /* Concatenate headings*/
    Start = 0                       /* No longer start         */
  End
  Else Do                           /* If not heading          */
    Parse Var Line Product Value Amount Discount /* Parse values*/
    /* Do some calculations and string functions               */
    Total = Right(Format((1-Discount/100)*Value*Amount,6,2),15)
    Out.LineCount = Line Total      /* Concatenate Total to Line*/
    SubTot = SubTot + Total         /* Total for company       */
    Select   /* Calculate totals for products                  */
      When Product = 'A' then A = A + Total
      When Product = 'B' then B = B + Total
      When Product = 'C' then C = C + Total
      When Product = 'D' then D = D + Total
      Otherwise nop
    End
  End
  Linecount = LineCount + 1         /* Increment line count    */
End
Call Printit                        /* Print last company
*/ Final = 'Totals for Products:'      /* Summary                  */
Final = Final'* Total Product A:' A
Final = Final'* Total Product B:' B
Final = Final'* Total Product C:' C
Final = Final'* Total Product D: 'D
Final = Final'* Totals Sales:' A+B+C+D /* Final amount          */
'@DETACH MULTIPRT 'Final            /* Print summary lines */
                                           /* in a separate process */
Exit
```

Figure 6. Part of the Program to Do Calculations and Call Print Routine

```
PrintIt:
/* Print reports in separate process                          */
OutLine = ''                          /* Initialize to null string*/
LineCount = LineCount - 1             /* Deduct by 1              */
Do I = 1 to LineCount                 /* Concatenate to one string*/
  OutLine = OutLine'*'Out.I           /* to pass on to printing   */
End
OutLine = OutLine'*                             Total:'SubTot
LineCount = 1                         /* Start next company       */
'@DETACH MULTIPRT 'OutLine            /* Print in separate session*/
/* MULTIPRT.CMD doesn't need screen output or keyboard input    */
/* so we can use DETACH, otherwise would have to use START      */
SubTot = 0                            /* Start next company with 0*/

Return
```

Figure 7. Part of the Program to Print Reports in a Separate Process

```
/* MULTIPRT.CMD                                               */
/* Print strings separated by * to LPT1                       */
Arg Rest                         /* Strig as argument       */
Rest = Strip(Rest,'L','*')       /* Strip leading *         */
Do Forever
  Parse Var Rest ToPrt'*'Rest    /* Parse to separate lines */
  Rc = Lineout('LPT1:',ToPrt)    /* Output to printer     */
  /* For otput to file change LPT1: to fully qualified fname*/
  If Strip(Rest) = '' then leave /* Nothing else to process */
End
```

Figure 8. Separate REXX CMD File to Print Output

```
SALES FIGURES FOR 1994, A COMPANY
PRODUCT  VALUE  AMOUNT  DISCOUNT  NET VALUE
A        1500   20      3         29100.00
B        2500   30      0         75000.00
C        1000   40      25        30000.00
D        450    20      25         6750.00
                          TOTAL:140850.00


SALES FIGURES FOR 1994, B COMPANY
PRODUCT  VALUE  AMOUNT  DISCOUNT  NET VALUE
A        1500   15      15        19125.00
B        2500   10      10        22500.00
C        1100   20      20        17600.00
D        450    15      15         5737.50
                          TOTAL:64962.50


TOTALS FOR PRODUCTS:
 TOTAL PRODUCT A: 48225.00
 TOTAL PRODUCT B: 97500.00
 TOTAL PRODUCT C: 47600.00
 TOTAL PRODUCT D: 12487.50
 TOTALS SALES: 205812.50
```

Figure 9. Printed Output from REXX Multitasking Sample

2.2 File I/O with OS/2 REXX

Most REXX programmers in the VM and TSO environments are used to using EXECIO or IOX for file I/O. However EXECIO is not included in either the SAA definitions or OS/2. Instead, OS/2 uses the following functions:

- Charin
- Charout
- Chars
- Linein
- Lineout
- Lines
- Stream

All of these functions, with the exception of Stream, are also included in REXX on the other platforms and provide a way for making REXX programs more portable. The complete syntax of these functions can be found in Appendix A, "REXX Syntax Diagrams" on page 225.

2.2.1 Charin(name,start,length)

Used for character input from stream.

Charin will increase the read/write position in persistent streams (files) with the number of characters read.

There are several ways to reset the state of the file. The most commonly used is to issue an Rc = Charout(Filename) command, which will close the file. When you reopen the file it will be positioned at the beginning. A start value can also be given to provide the start position within the file. Charin returns all characters that appear in the stream including control characters such as line feed, carriage return, and end of file.

If the name parameter is omitted then characters will be read from the default input stream, STDIN, which is usually the keyboard. Keyboard input requires you to press the Enter key to input the data and continue processing. An example of the use of Charin is as follows:

```
/* If the first line in CONFIG.SYS is                    */
/* IFS=C:\OS2\HPFS.IFS  /CACHE....                 */
Chr = Charin('C:\CONFIG.SYS',,3)
/* would return IFS the first time, =C: the second time...*/
Chr = Charin('C:\CONFIG.SYS',12,4)
/* would always return HPFS                             */
Chr = Charin(,,3)
/* would display Myn if user typed Myname               */
Chr = Charin()
/* would display M if user typed Myname                 */
```

Note

If you wish to get keyboard input without having to press the Enter key then you can use the RexxUtil function SysGetKey. SysGetKey also provides added function like Noecho to keep characters from echoing on screen. SysGetKey is described in detail in Chapter 4, " REXX Utilities External Function Package (REXXUTIL)" on page 53.

2.2.2 Charout(name,string,start)

Used for character output to stream.

Charout will increase the read/write position in persistent streams (files) with the number of characters written.

If name parameter is omitted then characters will be written to the standard output stream, STDOUT, which is usually the display. A start value can be given to move the write position in files. Rc = Charout(name) will close the file.

```
Rc = Charout('C:\OUT.FIL','Text to be written',1)
/* would write the text to file beginning at  */
/* the first position                         */
Rc = Charout(,'I wrote this out')
/* would display the text on screen without   */
/* line feed                                  */
```

2.2.3 Chars(name)

Used for character count in stream.

Will return the remaining number of characters in a stream from the current read position to the end of file.

In VM which has a line-based file system Chars() will return 1 if there are characters left in stream and 0 if there are not. So the best way to make a program portable is to check if Chars() > 0.

```
/* Show the contents of CONFIG.SYS file              */
/* and report progress                               */
InFile = 'C:\CONFIG.SYS'     /* Set intitial values   */
I = 0
Size = Chars(InFile)         /* Total size            */
Do While Chars(InFile) > 0   /* while chars remaining */
I = I + 1                    /* increment I by 1      */
  Character = Charin(InFile) /* read 1 character in   */
  Rc = Charout(,Character)   /* write it to display   */
  If I//100 = 0 then Do      /* show remaining chars  */
    Say                      /* every 100th time      */
    Say 'Characters remaining: 'Chars(InFile)
  End
End
Exit
```

Figure 10. Example of Charin, Charout and Chars

2.2.4 Linein(name,line,count)

Used for line input from stream.

Linein will also increase the read/write position in persistent streams.

You can only give the value 1 in OS/2 for the line parameter, which means start read operation at line number one. The count parameter can be given a value 0 which means that no characters are read and the file is only opened, or 1 which is also the default and reads one line.

```
Line = Linein('C:\CONFIG.SYS')
/* would give first line in CONFG.SYS        */
```

2.2.5 Lineout(name,string,line)

Used for line output to stream.

Lineout will also increase the read/write position in persistent streams.

You can only give the value 1 in OS/2 for the line parameter, which means write at the first character in file. Lineout(name) will close a file.

```
Line = Lineout('C:\OUT.FIL','Write this in the file')
/* would write the line in file               */
```

2.2.6 Lines(name)

Used to check if data remains in stream.

Will return 1 if lines exist and 0 if no more lines exist in stream.

In VM Lines() will return the true number of lines left in stream. So the best way to make a program portable is to check if Lines() > 0.

The following example might return an error in VM, because the logical value of Lines() could be something other than 0 or 1. In OS/2 the example would work fine. To make the program portable use Do While Lines(InFile) > 0 instead of Do While Lines(InFile).

```
/* Read through CONFIG.SYS line by line       */
InFile = 'C:\CONFIG.SYS'
Do While Lines(InFile)
  Line = Linein(InFile)
  Say Line
End
Rc = LineOut(InFile)
Exit
```

Figure 11. Example of Linein and Lines Commands

2.2.7 Stream(name,operation,streamcommand)

Used for querying state of stream and performing functions to stream like opening the stream for read or write operations or setting the read or write position in stream.

The name argument describes the name of stream.

The operation argument can have three values C, D or S. If you give C, which stands for Command, as the second argument then the third argument will be used as the command to be performed against the stream. D and S will give the state of the stream, with D returning more information in ERROR and NOTREADY states. State can return one of the following strings: ERROR,

NOTREADY, READY, UNKNOWN. Figure 12 on page 23 shows Stream command usage.

```
String = Stream('C:\CONFIG.SYS','c','query exists')
/* Gives C:\CONFIG.SYS if file exists and     */
/* gives a null string if it doesn't          */
Rc = Stream('C:\CONFIG.SYS','c','open')
/* Opens C:\CONFIG.SYS for read and write     */
Rc = Stream('C:\CONFIG.SYS','c','open write')
/* Opens C:\CONFIG.SYS for write operations   */
Rc = Stream('C:\CONFIG.SYS','c','open read')
/* Opens C:\CONFIG.SYS for read operations    */
Rc = Stream('C:\CONFIG.SYS','c','close')
/* close C:\CONFIG.SYS                        */
Rc = Stream('C:\CONFIG.SYS','c','seek = 5')
/* seek position to absolute 5th character in file */
Rc = Stream('C:\CONFIG.SYS','c','seek + 5')
/* move position ahead 5  characters               */
Rc = Stream('C:\CONFIG.SYS','c','seek - 5')
/* move position back 5 characters                 */
Rc = Stream('C:\CONFIG.SYS','c','seek < 5')
/* move position back 5 characters from end of file*/
```

Figure 12. Examples of Stream Command Usage

2.2.8 Examples

Figure 13 on page 24 shows a VM example for reading a file into a stem variable and writing the lines to an other file with timestamp as first line and the rest concatenated to one single line. This example does not work for OS/2.

Figure 14 on page 24, on the other hand, shows an example for reading a file into a stem variable and writing the lines to another file with time stamp as first line and the rest concatenated to one single line. This example works both on OS/2 and VM.

```
/*     Arguments InFile , Outfile
        Example: InOut PROFILE EXEC A,OUT FILE A            */

 Arg InFile ',' OutFile
 OutLine = ''
 'EXECIO * DISKR 'InFile' (FINIS STEM TEXT.'  /* Read lines into stem */
 'EXECIO 1 DISKW 'OutFile' (STRING 'DATE() TIME() /* Write timestamp  */
 Do I = 1 to TEXT.0
   OutLine =  OutLine||Text.I             /* Concatenate to one line  */
 End
 'EXECIO 1 DISKW 'OutFile' (VAR OUTLINE'  /* Write line to OutFile    */
 'FINIS 'OutFile                          /* close file               */

Exit
```

Figure 13. VM Specific Example of Reading a File Using EXECIO

```
/*  Arguments InFile , Outfile
     Example   (VM): InOut PROFILE EXEC A,OUT FILE A
     Example (OS/2): InOut C:\CONFIG.SYS,C:\OUT.FIL
 */

 Arg InFile ',' OutFile
 OutLine = ''                             /* Empty variable     */
 Do While Lines(InFile) > 0               /* loop until EOF     */
   Text = Linein(InFile)                  /* get line from file */
   OutLine = OutLine||Text                /* concatenate lines  */
 End
 Rc = Lineout(OutFile,DATE() TIME())      /* Write timestamp    */
 Rc = Lineout(OutFile,OutLine)
 Rc = Lineout(InFile)                     /* Close input file   */
 Rc = Lineout(OutFile)                    /* Close output file  */

Exit
```

Figure 14. OS/2 and VM Independent Example of Reading a File

2.3 RxQueue

Most VM REXX programmers are also familiar with MAKEBUF, DROPBUF and DESBUF for creating, dropping and clearing console and program stacks. OS/2, however, uses the RXQUEUE function instead to create and delete queues.

The usage of PULL, PUSH and QUEUE for placing data in and getting data from the stack and QUEUED for querying number of lines in stack work the same way in all REXX environments.

2.3.1 PUSH

Will place lines to the currently active queue. Data will be placed LIFO (Last In First Out). For example:

```
Push '1'
Push '2'
Push '3'
Do 3
  Pull Data
  Say Data
End
```

Will display:

```
3
2
1
```

2.3.2 QUEUE

Will place lines to the currently active queue. Data will be placed FIFO (First In First Out). For Example:

```
Queue '1'
Queue '2'
Queue '3'
Do 3
  Pull Data
  Say Data
End
```

Will display:

```
1
2
3
```

There are two kinds of queues in OS/2 REXX namely the session queue and private queues. The session queue is automatically provided for each OS/2 session and its name is always SESSION. For instance, each OS/2 command prompt is a session. So queuing lines to the SESSION queue will not make those lines available for other programs started from other command prompts. Also, starting a program with the START or DETATCH commands will start new sessions so the queued lines will not become available to the started programs. In other words, the example in Figure 15, will not display anything if you use START /F GETSQ. It will, however, display 10 lines if started with CALL. This is because the session queue will then be running in the same session as PUTSQ.CMD. To overcome this you must use the private queues. See 2.3.3, “Private Queues Using RXQUEUE” on how to do this.

```
/* PUT2SQ.CMD                                                    */
/* Queue lines to Session queue                                  */
Do I = 1 to 10
  Queue 'This is line number 'I /* Put 10 lines in SESSION QUEUE    */
End
Call GETSQ                /* Run GETSQ.CMD in this session  */
```

Figure 15. Example of Loading and Displaying a Session Queue

```
/* GETSQ.CMD                                                     */
/* Get lines from SESSION queue                                  */
Do Queued()
  Pull Line
  Say Line
End
```

Figure 16. Example of Displaying a Session Queue

2.3.3 Private Queues Using RXQUEUE

Private queues are created and deleted by your program so they are similar to the VM MAKEBUF. However, in OS/2 the program that wishes to use a queue must know the name of the queue and every queue must have a unique name. The parameters for handling queues with RXQUEUE are:

RXQUEUE('create',queuename)

Creates a new queue with the name provided in the optional parameter queuename. If no name is provided then OS/2 will

automatically give the queue a unique name. Also if a queue exists with the name given in the queuename parameter then OS/2 will provide a new name for the queue that is being created. This could give unpredictable results in some cases.

An example of how to check that the created queue name is what is expected is:

```
QN = 'QUE1'
NewQ = RXQUEUE('create',QN)
If NewQ <> QN then
  Say 'Queue with the name 'QN' already exists'
```

RXQUEUE('set',queuename)

Make a queuename the active queue.

RXQUEUE('get')

Get name of currently active queue.

RXQUEUE('delete',queuename)

Delete the queue with the name queuename.

The following four figures show the use of private queues. Figure 17 creates 3 queues and lets OS/2 handle the naming of the queues. It then calls two other procedures in the same command file: one to place data in those queues, Figure 18 on page 28, and another to show the contents of the queues, Figure 19 on page 28. The displayed output is shown in Figure 20 on page 29.

```
/* SHUFFLE.CMD                                              */
Call Create                    /* create the queues          */
Call SetUp                     /* set queues active and queue */
Call Show                      /* show result                */
Exit

Create:
  Do I = 1 to 3
    Q.I = RXQUEUE('create')    /* create an array of 3 queues */
  End
Return
```

Figure 17. Private Queue Part 1, Create Queues

```
Setup:
  Do I = 1 to 10                  /* loop 10 times               */
    Select
      When I//3 = 0 then Do       /* if i//3 remainder = 0 then  */
        Rc = RXQUEUE('set',Q.3)   /* set queue to third queue    */
        Queue 'Number 'I          /* queue data                  */
      End
      When I//2 = 0 then Do       /* if i//2 remainder = 0 then  */
        Rc = RXQUEUE('set',Q.2)   /* set queue to second queue   */
        Queue 'Number 'I          /* queue data                  */
      End
      Otherwise Do                /* otherwise                   */
        Rc = RXQUEUE('set',Q.1)   /* set queue to first queue    */
        Queue 'Number 'I          /* queue data                  */
      End
    End /* Select */
  End /* I = 1 to 10 */
Return
```

Figure 18. Private Queue Part 2, Place Data in Queues

```
Show:
  Do I = 1 to 3
    Rc = RXQUEUE('set',Q.I)       /* set active queue            */
    Say 'Queue number 'I' with name 'Q.I /* show name            */
    Do Queued()                   /* show queued data            */
      Pull Stuff
      Say Stuff
    End
  End
  Do I = 1 to 3
    Say 'Now deleting queue 'Q.I
    Rc = RXQUEUE('delete',Q.I)    /* delete the queues           */
  End
Return
```

Figure 19. Private Queue Part 3, Show Contents

```
Queue number 1 with name S19Q0322109576
NUMBER 1
NUMBER 5
NUMBER 7
Queue number 2 with name S19Q0322109632
NUMBER 2
NUMBER 4
NUMBER 8
NUMBER 10
Queue number 3 with name S19Q0322109688
NUMBER 3
NUMBER 6
NUMBER 9
Now deleting queue S19Q0322109576
Now deleting queue S19Q0322109632
Now deleting queue S19Q0322109688

The queue names will vary every time program
is run.
*/
```

Figure 20. Displayed Output from Private Queue Example

Figure 21 and Figure 22 on page 31 are examples of using queues to transfer data over separate sessions.

```
/* PUT2QUE.CMD                          */
/* Put lines to private queue           */
/* Add REXXUTIL function SysSleep       */
call RxFuncAdd 'SysSleep','RexxUtil','SysSleep'

NQ   = 'QUE1'                           /* put queue name into variable  */
NewQ = RXQUEUE('create',NQ)             /* create new queue              */
If NewQ <> NQ then Do                   /* if queue name already exists  */
  Say 'Queue with the name 'NQ' already exists, exiting program'
  Rc = RXQUEUE('delete',NewQ)           /* show message, delete queue and*/
  Exit                                  /* exit                          */
End
OQ = RXQUEUE('Set',NewQ)                /* establish new queue           */
push date() time()                      /* push date and time            */
Do I = 1 to 10
  Queue 'Line number 'I                 /* queue 10 lines to  queue      */
End
'START /F /C GETFROMQ '                 /* start GETFROMQ.CMD            */
                                        /* /F = foreground  /C = close   */
                                        /* session after termination     */
Do while queued() > 0                   /* display lines in private queue*/
  Call SysSleep 1                       /* wait for 1 second             */
  Say queued()                          /* display num.of lines remaining*/
End

Exit 0
```

Figure 21. Private Queue Across Two Separate Sessions Part 1

```
/* GETFROMQ.CMD                        */
/* get lines from private queue        */
/* Add REXXUTIL function SysSleep      */
call RxFuncAdd 'SysSleep','RexxUtil','SysSleep'
oq = RXQUEUE('Set','QUE1')             /* establish new queue          */
Do Queued()                            /* loop for amount of lines     */
  Call SysSleep 1                      /* sleep for 1 second           */
  Pull Line                            /* pull a line out of queue     */
  Say 'Pulled 'Line' out of QUE1'      /* display line                 */
End
Curq = RXQUEUE('Get')                  /* get name of current queue    */
Call RXQUEUE 'Delete',curq             /* destroy unique queue created */

Exit 0
```

Figure 22. Private Queue Across Two Separate Sessions Part 2

OS/2 REXX doesn't currently have global compound variables so queues can provide a simple way of exchanging data between separate programs. They can also be useful in multitasking when you want to control when a process is finished. Figure 23 is an example of how to place keyboard input into a separate session.

```
/* QGETKEY.CMD                                                  */
/* Get keyboard input from called program                       */
/* pass queuename to KEYS2Q.CMD and communicate through it      */
/* if ESC (x'1B') pressed then leave                            */
NewQueue = RXQUEUE('create')
OldQueue = RXQUEUE('Set',NewQueue) /* set NewQueue active       */

'START /WIN KEYS2Q' NewQueue       /* start windowed session    */

Do Forever                         /* never ending loop         */
  If Queued() > 0 then do          /* if chars queued then act  */
    parse pull Char                /* pull mixed case           */
    If C2X(Char) = '1B' Then Leave /* A way out (ESC)           */
    Rc = Charout(,Char)            /* output char to screen     */
  End
End
Rc = RXQUEUE('delete',NewQueue)    /* delete the queue          */

Exit
```

Figure 23 (Part 1 of 2). Private Queue, Input in Separate Session Part 1

```
/* KEYS2Q.CMD                                                    */
/* Get keyboard input and pass it to calling program            */
/* get queuename as argument and use it to pass data            */
/* if ESC (x'1B') pressed then leave                            */
/* Add RexxUtil function SysGetKey                              */
call RxFuncAdd 'SysGetKey','RexxUtil','SysGetKey'
Arg QueueName                       /* Queue name as argument */
QueueName = Strip(QueueName)        /* strip blanks           */
OldQueue=RXQUEUE('Set', QueueName) /* set QueueName active   */
Do Forever                          /* eternal loop           */
  KeyPressed = SysGetKey()          /* get key from keyboard  */
  Queue KeyPressed                  /* place it in queue      */
  If C2X(KeyPressed) = '1B' Then Leave  /* A way out (ESC)    */
End
'@EXIT'
```

Figure 23 (Part 2 of 2). Private Queue, Input in Separate Session Part 1

Queues can also be useful when getting output from system commands into REXX. Figure 24 on page 33 illustrates the use of REXX to get command output and error output into a REXX program and then manipulating it, for example, to show more information about the error situation via HELPMSG.

```
/* ENV2Q.CMD                                                    */
/* Get environment values from queue, demonstrates the use of */
/* REXX to get STDOUT and STDERR output from system commands  */
/* through RXQUEUE                                            */
/* Add REXXUTIL function SysGetKey                            */
Call RxFuncAdd 'SysGetKey','RexxUtil','SysGetKey'
/* Add REXXUTIL function SysSleep                             */
Call RxFuncAdd 'SysSleep','RexxUtil','SysSleep'
/* Add REXXUTIL function SysCls                               */
Call RxFuncAdd 'SysCls','RexxUtil','SysCls'
Call SET2Q
Call YN
Call DIR2Q
Call YN
Call MDI2Q
Call YN
Call NET2Q
Exit

YN:

Do 2
  Say
End
Say "Do you wish to continue (Y/N)?"
Ans = SysGetKey("noecho")
If Pos(Ans,'Yy') > 0 then nop
Else Exit

Return
```

Figure 24. System Output to REXX Part 1, Main Body

```
SET2Q:

Say "Using REXX to filter out information from the SET command:"
'@SET | RXQUEUE'          /* give SET command and pipe         */
                          /* through RXQUEUE                   */

Do While Queued() > 0     /* loop while lines exist in queue   */
  Pull SetValue           /* pull a value from queue           */
  SetValue = ' 'SetValue  /* add blank to beginning            */
                          /* (simplifies the search)           */
  Select                  /* find out which value it is        */
                          /* and act accordingly               */
    When Pos(' BOOKSHELF=',SetValue) > 0 then Do
      Say "Your BookShelf has the following entries:"
                          /* show only the actual value        */
                          /* without BOOKSHELF= by finding out */
                          /* the position of = and taking only */
                          /* the string after it               */
      Say "   " Substr(SetValue,Pos('=',SetValue)+1)
      Say
    End
    When Pos(' HELP=',SetValue) > 0 then Do
      Say "Your Help has the following entries:"
      Say "   " Substr(SetValue,Pos('=',SetValue)+1)
      Say
    End
    When Pos(' PATH=',SetValue) > 0 then PathValue = SetValue
                          /* Process path after everything is  */
                          /* pulled out of the queue because   */
                          /* will be using PULL for keyboard   */
                          /* later                             */
    Otherwise nop
  End
End
```

Figure 25. System Output to REXX through Queues Part 1

```
Say "Your Path has the following value:"
Say "   " Substr(PathValue,Pos('=',PathValue)+1)
Say
Say "Do you wish to add a directory to the current path"
Say "for this session (Y/N)?"
Key = SysGetKey("noecho")/* get key from keyboard without    */
                         /* showing it                       */
Key = Translate(Key,'YN','yn') /* translate to uppercase     */
If Key = Y Then Do
  Say "Give the fully qualified name for the directory",
      "you want to add:"
  Pull DirName           /* pull directory name from kbd     */
  DirName = Strip(DirName) /* take away leading and trailing */
                           /* blanks                         */
  Dirname = Strip(DirName,,';') /* take away semicolons      */
                           /* if last character in path is   */
                           /* semicolon, don't add one       */
  If LastPos(';',PathValue) = Length(PathValue) then
     NewValue = PathValue||DirName||';'
  Else                     /* else add one                   */
     NewValue = PathValue||';'||DirName||';'
  '@'NewValue              /* execute string as command without*/
                           /* echoing to screen (@)          */
  Say "Your new path is now:"
  Say NewValue             /* say new path                   */
End
Return
```

Figure 26. System Output to REXX through Queues Part 2

```
DIR2Q:

Call SysCls              /* clear screen                     */
I = 0                    /* set initial values               */
Quiet = 0                /* variable quiet will be used in   */
                         /* boolean operations so it has to  */
                         /* have initial value 1 (TRUE) or   */
                         /* 2 (FALSE)                        */
Say "These are the contents of your current directory:"
'@DIR | RXQUEUE'         /* give DIR command without ECHO    */
Do While Queued() > 0
  I = I + 1
  If I//14 = 0 & \Quiet Then Do /* pause every 14th row      */
    Say "Push any key to continue. ESC to terminate."
    Key = SysGetKey("noecho")
    If C2X(Key) = '1B' Then Quiet = 1
  End
  Pull DirEntry
  If \Quiet Then Say DirEntry /* if not quiet mode show entry*/
End

Return
```

Figure 27. System Output to REXX through Queues Part 3

```
MDI2Q:
Call SysCls
Say "Now showing how to get error messages from OS/2 commands:"
Say "Giving a MD (Make Directory) command with an invalid",
    "character (*) in it:"
Call SysSleep 3          /* Sleep for 3 seconds             */
'@MD C:\*DUMMY 2>&1 | RXQUEUE' /* Pipe STDERR info to RXQUUE*/
Do While Queued()
  Pull Stuff
  Say "Sorry, can't do it"
  Say "This is what the system said:"Stuff
  Say "This is what it meant:"
  MessageNum = Strip(Word(Stuff,1),,':') /* message number  */
                       /* is first word without colon     */
  '@HELPMSG' MessageNum  /* display extended help          */
End

Return
```

Figure 28. System Output to REXX through Queues Part 4

```
NET2Q:

Call SysCls
/* show the same with LAN requester NET command           */
Say "Now trying to get error message from NET commands:"
Call SysSleep 3
'@NET START REQ1 2>&1 | RXQUEUE'
Do While Queued() > 0
  Pull Stuff
  Say "No way:" stuff
  MessageNum = Strip(Word(Stuff,1),,':')
  '@HELPMSG' MessageNum
End

Return
```

Figure 29. System Output to REXX through Queues Part 5

2.3.4 LIFO, FIFO and CLEAR

In OS/2 REXX data is placed in queues First In First Out (FIFO) by default. Figure 30 shows how the default order can be changed to Last In First Out (LIFO). The example creates a private queue, reads a file in LIFO, pulls and then it displays the last line. Then it clears all lines in the queue using:

```
'@RXQUEUE 'nq' /CLEAR'
```

Next it places data in the queue FIFO and pulls and displays the first line.

```
/* RXLIFO.CMD                                          */
/* Get first and last line in this command file        */
/* Demonstrate use of /LIFO and /FIFO and /CLEAR       */
Parse Source . . Myname /* Get the name of this file   */

NQ = RXQUEUE('create')  /* Create a new queue          */
OQ = RXQUEUE('set',NQ)  /* Set new queue active        */
'@RXQUEUE 'nq' /LIFO < 'Myname /* Queue all lines LIFO */
Pull Last                      /* Pull last line       */
Say 'Last line in 'Myname' is:'/* Display line         */
Say Last

'@RXQUEUE 'nq' /CLEAR'         /* Clear queue          */
'@RXQUEUE 'nq' /FIFO < 'Myname /* Queue all lines FIFO */
Pull First                     /* Pull first line      */
Say 'First line in 'Myname' is:'/* Display line        */
Say First
'@RXQUEUE 'nq' /CLEAR'         /* Clear queue          */
Rc = RXQUEUE('delete',NQ)      /* Delete queue         */
```

Figure 30. LIFO, FIFO and CLEAR with RXQUEUE

2.4 Printing

Printing from OS/2 REXX can be done basically with the OS/2 command PRINT or COPY. You can also redirect program output to a printer by using > . For example DIR > LPT1: or TYPE C:\CONFIG.SYS > LPT2:. If you wish to print individual lines or characters you can use the REXX functions Lineout or Charout. An example of using Lineout to print lines is in Figure 8 on page 17.

2.4.1 PRINT Command

You can use the OS/2 PRINT command to print files to a specified printer, to cancel an active print job or to cancel all print jobs on a specified print queue. For example: **PRINT /D:LPT2 C:\LISTING\REPORT.STA** will print the file REPORT.STA on the printer attached to LPT2. **PRINT /C** will cancel the file that is currently printing on the default print device. **PRINT /D:LPT2 /T** will cancel all files in the LPT2 print queue and any file currently printing on the same device.

2.4.2 Lineout and Charout

It is possible to direct output to a printer using Lineout or Charout by specifying the LPT port or PRN as the output device. For example: **Rc = Lineout('LPT1:','Text to be printed')** will print the text to the printer connected to LPT1 port. Following is an example of printing on the LPT1 port as two separate print jobs. Lineout('LPT1:') will close the spool. Closing the command file with Exit, Return or when the program has otherwise ended will also close the spool.

```
/* Print 10 lines as first job, close spool */
/* and print next 10 lines                  */
Do I = 1 to 20
  Rc = Lineout('LPT1:','Line number 'I)
  If I = 10 then Rc = Lineout('LPT1:')
End
```

The examples for Lineout apply also to Charout.

Directing trace output from a program to the printer is also useful in many cases. This can be done simply by directing the STDERR output from the program to the printer. Trace uses STDERR so the following works for trace output. **TRACEOUT.CMD 2> LPT1:**. Don't use any blanks between 2 and >.

2.4.3 Printer Objects

OS/2 REXX doesn't currently provide very many ways of interacting with the installed printer objects. It does however enable some amount of interaction. Included are three program listings describing the use of the RexxUtil function SysIni. SysIni is discussed in detail in Chapter 4, " REXX Utilities External Function Package (REXXUTIL)" on page 53.

PRPATH.CMD shown in Figure 31 on page 40 describes how to change the default printer driver installation path. After using the CD-ROM installation

the path for installing new printer drivers points to the original installation path. After diskette or remote (CID) installation the path is A:. PRPATH.CMD allows you to change this to whichever directory you want, for instance to a directory on the network print server. This could prove useful for system administrators.

```
/* Procedure to query or change the printer driver directory */
Parse Upper Arg Path . /* New path as argument                          */
Inifile = 'USER'     /* Information stored in USER inifile (OS2.INI) */
App = 'PM_INSTALL'   /* Application name in INI file                  */
Key = 'PDR_DIR'      /* Key for path information                      */
Key2 = 'MEDIA'       /* Key for media information                     */
Call RxFuncAdd 'SysIni', 'RexxUtil', 'SysIni' /* Add SysIni function */
If Path = "" Then Do /* If no path specified, query path and display */
  Path = SysIni('USER', App, Key)
  If path = 'ERROR:' Then Do /* No information found for PM_INSTALL  */
                             /* PDR_DIR                              */
    Say "The key '"key"' for the application '"App"'"
    Say "was not found in the '"Inifile"' inifile."
    Say "You must give a path if you wish to set one."
  End
  Else                       /* Say printer driver path              */
    Say "The actual printer driver path is:" Path
End
Else Do
  OldPath = SysIni('USER', App, Key) /* Save previous information    */
  If OldPath = 'ERROR:' Then Do      /* Not found                    */
    Say "The key '"key"' for the application '"App"'"
    Say "was not found in the '"Inifile"' inifile."
    Say "The new path will be set to:"Path
  End
  If Left(Path, 2) = 'A:' Then Media = 'DISKETTE'
  Else Media = 'REMOVABLE'   /* A drive=removable others not         */
  If Right(Path, 1) = '\' Then Path = Left(Path, Length(Path) - 1)
                             /* Remove last backslash                */
  Result = SysIni('USER', App, Key, Path) /* Change path             */
  Result = SysIni('USER', App, Key2, Media) /* Change media          */
  Say "The printer driver directory was changed to" Path
End
```

Figure 31. Change Printer Driver Installation Path

PRTIMOUT.CMD shown in Figure 32 on page 41 describes how to set the printer timeout value. Note that this is *not* the timeout value for the port. That value is set on the Output page of any printer object Settings notebook

by double clicking on the desired port. The value set in PRTIMOUT.CMD determines how long the printer error message box indicating that the printer is off-line or out of paper will display before an automatic retry is issued by the system. The system default for this timeout is 180 seconds, or three minutes.

```
/* Set the timeout for the printer offline menu                    */
Parse Upper Arg Timeout . /* Timeout value as argument             */
Inifile = 'SYSTEM'        /* Stored in SYSTEM inifile OS2SYS.INI   */
App = 'PM_SPOOLER_MSGBOX' /* Application name                      */
Key = 'TIMEOUT'           /* Key for application                   */
Call RxFuncAdd 'SysIni', 'RexxUtil', 'SysIni' /* Add SysIni function */
If Timeout = '' Then Do   /* If no timeout as argument             */
  Timeout = SysIni(Inifile, App, Key)
  If Timeout = 'ERROR:' Then Do
    Say "The key '"key"' for the application '"App"'"
    Say "was not found in the '"Inifile"' inifile. The system"
    Say "uses the default value of 180 seconds."
  End
  Else Say "The timeout value for the printer offline menu is",
            Timeout "seconds." /* Show current timeout value       */
End
Else Do
  If Datatype(Timeout, 'W') = 0 Then Do /* Value has to be numeric */
    Say "Parameter" Timeout "is not numeric."
    Exit 1
  End
  Result = SysIni(Inifile, App, Key, Timeout) /* Change value      */
  Say "The timeout value for the printer offline menu has been"
  Say "set to" Timeout "seconds."
End
```

Figure 32. Set Timeout Value for the Printer Offline Menu

PRTPORT.CMD shown in Figure 33 describes how to add printer ports LPT4 to LPT9 to OS2SYS.INI.

```
/* Add LPT4 to LPT9 into OS2SYS.INI                                */
call RxFuncAdd 'SysIni', 'RexxUtil', 'SysIni'
Do I = 4 to 9
  Call SysIni 'SYSTEM', 'PM_SPOOLER_PORT', 'LPT'||i, ';'||'00'x
End
```

Figure 33. Set Printer Ports LPT1 to LPT9

2.5 PMREXX, REXXTRY and RxMessageBox

Some PM features can be included in your REXX program by using PMREXX and RxMessageBox. If you need more PM features see Chapter 10, “Visual REXX Builders” on page 177 which explains some of the visual builders available today.

2.5.1 PMREXX

PMREXX is a windowed Presentation Manager application that enables you to browse the output of your REXX procedures. REXXTRY is a command that lets you interactively run one or more REXX instructions.

By using PMREXX, you add the following features to REXX:

- A window to display the output of a REXX procedure, such as:
  - The SAY instruction output
  - The STDOUT and STDERR outputs from secondary processes started from a REXX procedures file
  - The REXX TRACE output (not to be confused with OS/2 tracing)
- Input window for:
  - The PULL instruction in all of its forms
  - The STDIN data for secondary processes started from a REXX procedures file
- Browsing, scrolling, and clipboard capability for REXX output
- Selection of fonts for the output window
- Simple environment for experimenting with REXX instructions through the use of the REXXTRY.CMD program

Figure 35 on page 44 demonstrates the use of PMREXX to provide a PM front-end to non-PM REXX programs. The command can be started either by typing **RXCALC**, which will start the program in an OS/2 window, or by typing **START PMREXX RXCALC**, which will start the program using PMREXX.

```
/* RXCALC.CMD                                                  */
/* REXX calculator example using PMREXX                        */
/* Add RexxUtil function SysCls                                */
Call RxFuncAdd 'SysCls','RexxUtil','SysCls'
Env = Address()    /* Check environment PMREXX if PMREXX       */
                   /* Else CMD                                 */
Line = ''          /* Initialize some values                   */
V = ''
LFeed = '0d0a'x    /* Linefeed for messagebox                  */
Call Main          /* Call the main process                    */

Exit

Main:

Do Forever         /* Do until exit                            */
  Signal on Syntax /* If syntax error then signal Syntax       */
  If Line = 'EXIT' then Exit /* Leave loop                     */
  Call Menu        /* Show the menu                            */
  Interpret 'V =' Line /* Interpret line to REXX               */
  Call SysCls      /* Clear screen                             */
End

Return
```

Figure 34. PMREXX Example - RXCALC.CMD

```
Menu:              /* Show the menu                        */

Say
Say '  You can use the following functions:'
Say
Say '   + : Add                   The result will be in the variable V'
Say '   - : Subtract             which can also be used in calculation'
Say '   * : Multiply'
Say '  ** : Exponent'
Say '   / : Divide'
Say '  // : Remainder'
Say '   % : Integer divide'
Say
Say '  You can also use parenthesis'
Say '  Type EXIT to exit'
Say Line           /* Show previous input if any          */
Say 'Result=' V    /* Show previous result                */

Pull Line

Return

Syntax:            /* Syntax error                        */
                   /* Build the strings for RxMessageBox  */
Msg = 'Error in the calculation !'LFeed||Line
Title = 'Syntax Error'
Button = 'OK'
Type = 'ERROR'
Msgx=c2x(Msg'#'Title'#'Button'#'Type)
If Env = 'PMREXX' Then /* If PMREXX then just show message*/
  Action=RxMessageBox(Msg,Title,Button,Type)
Else /* Else start RXMSG.CMD in PM session and show msg   */
  '@START /PM CMD.EXE /C RXMSG.CMD 'Msgx
/* Pass values as Hex RXMSG.CMD will translate to chars   */
Line = ''          /* Empty input and result              */
V = ''
Signal Main        /* Signal main process                 */

Return
```

Figure 35. REXX Calculator for PMREXX

Running the program through PMREXX will produce the window shown in Figure 36 on page 45.

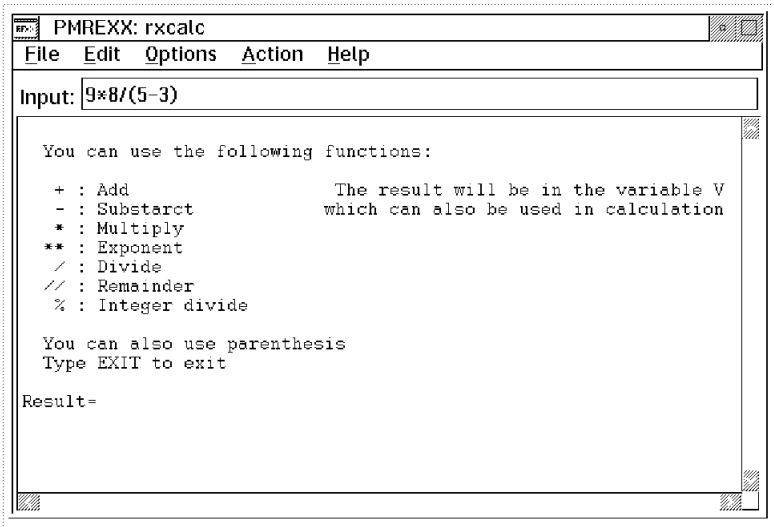


Figure 36. PMREXX Window for RXCALC

```
/* RXMSG.CMD                                                        */
/* Show messages in PM Message box                                  */
Arg Message
Msg = x2c(Message)
Parse Var Msg Msg'#'Title'#'Button'#'Type
OQ = RxQueue('set',Qn)
/* Arguments passed in Hexadecimal format to ensure correct result */
/* Hex values then changed to characters and RxMessageBox           */
/* Called with the character values                                 */
Action=RxMessageBox(Msg,Title,Button,Type)

Exit
```

Figure 37. Using RxMessageBox from Non-PM Programs

If an error is encountered the message in Figure 38 on page 46 is displayed.

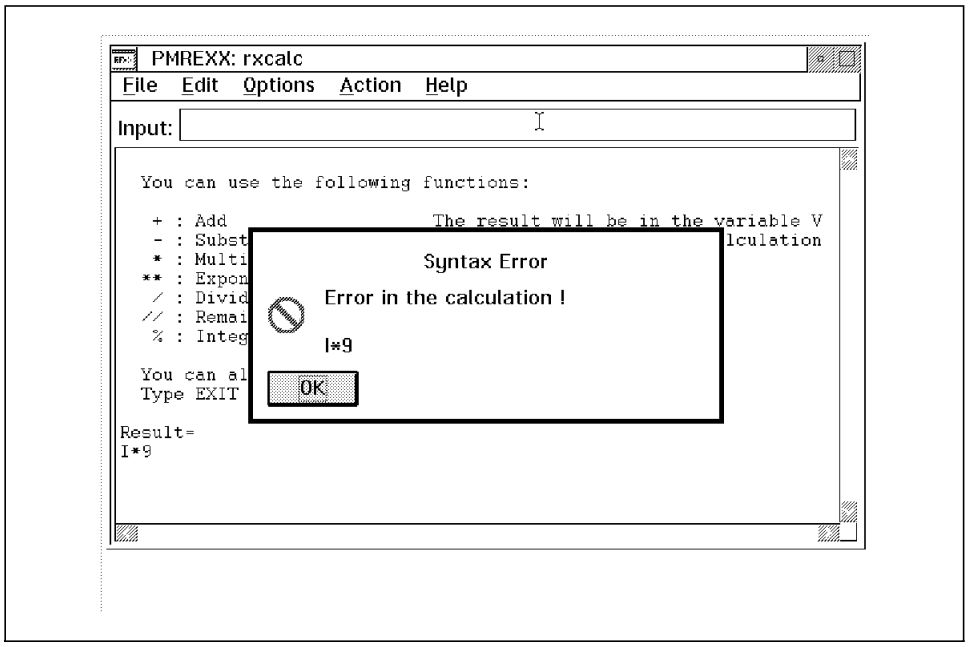


Figure 38. RxMessageBox for RXCALC.CMD

Once the output of the REXX procedure is displayed in PMREXX, you can select the menu-bar choices to take advantage of the following PMREXX browsing features:

File Save, Save As, and Exit the process

Edit Copy, Paste to the input, Clear the output, and Select All lines

Options Restart the process, Interactive Trace, and Set font

Actions Halt procedure, Trace next clause, Redo the last cause, and Set Trace off

Help Help index, General help, Keys help, and Using help

2.5.2 REXXTRY

REXXTRY is a REXX program. As with other REXX programs, REXXTRY can be run in an OS/2 full-screen or window session, or with PMREXX.

You can use REXXTRY to run different REXX instructions and observe the results. REXXTRY is also useful when you want to perform a REXX operation only once, since it is easier than creating, running, and erasing a .CMD file.

Here are some examples of how to use the REXXTRY command:

```
REXXTRY say 1+2
The operation is performed and 3 is displayed.
REXXTRY say 2+3; say 3+4
5 and 7 are displayed.
```

2.5.3 RxMessageBox

RxMessageBox allows you to display a message box from a REXX program running in an OS/2 session (that is, running in PMREXX or called from a Presentation Manager application). You can also display a message box from a non-PM .CMD file by starting a command file through CMD.EXE line which is shown in Figure 34 on page 43 and Figure 37 on page 45. The syntax of the RxMessageBox command is action = RxMessageBox(text, title, button, icon) where:

text Is the text of the message that appears in the message box.

title Is the title of the message box. The default title is ″Error″.

button Is the style of buttons used with the message box. The allowed styles are:

- **OK** A single OK button (the default)
- **OKCANCEL** An OK button and a Cancel button
- **CANCEL** A single Cancel button
- **ENTER** A single Enter button
- **ENTERCANCEL** An Enter button and a Cancel button
- **RETRYCANCEL** A Retry button and a Cancel button
- **ABORTRETRYIGNORE** An Abort button, a Retry button and an Ignore button
- **YESNO** A Yes button and a No button
- **YESNOCANCEL** A Yes button, a No button and a Cancel button

icon Is the style of icon displayed in the message box. The allowed styles are:

- **NONE** No icon is displayed
- **HAND** The hand icon is displayed
- **QUESTION** A question mark icon is displayed
- **EXCLAMATION** An exclamation mark icon is displayed

ASTERISK An asterisk icon is displayed

INFORMATION The information icon is displayed

QUERY The query icon is displayed

WARNING The warning icon is displayed

ERROR The error icon is displayed

action Is the button that was selected on the message box. Possible values are:

1: OK key

2 Cancel key

3 Abort key

4 Retry key

5 Ignore key

6 Yes key

7 No key

8 Enter

For example:

```
                                    /* Give option to quit        */
if RxMessageBox("Shall we continue",, "YesNo", "Query") = 7
  Then Exit                         /* "No" key given, exit       */
```

Chapter 3. External Functions

REXX programs can invoke functions that are physically located in other files. These external functions can be written in REXX, or in compiled languages. For external functions written in REXX, the function is called in the same manner as if it were an internal function.

```
/* Main REXX program                                          */
answer = ADDFUNC(var1,var2)   /* invokes ADDFUNC.CMD          */
```

Figure 39. Calling External REXX Function

When the main program processes the call to ADDFUNC, the main program is searched first for a function called ADDFUNC. If it is not found, then a disk search is performed that looks for a file called ADDFUNC.CMD.

External functions written in compiled languages must be registered with the calling REXX program before they can be invoked. This type of external function is contained in an executable file (.EXE) or a dynamic link library (.DLL). The rest of this chapter focuses on external functions written in compiled languages.

3.1 Usefulness

The fact that REXX programs can access external functions written in compiled languages makes REXX much more powerful. The horizons of what can be accomplished through REXX are broadened. Some of the opportunities provided by being able to access externally compiled functions are:

- Provides interface to other languages, such as C
- Do not have to rewrite in REXX existing functions that are written in other languages
- Can take advantage of the speed of compiled languages
- Provides a way to accomplish tasks that cannot be performed in REXX code
- Access to APIs for various software packages, for example DB2/2, EHLLAPI

3.2 How to Register External Functions

REXX programs require external functions to be registered. Registering a function makes the location of the function known to the REXX program. Once registered, functions are available to all REXX programs running on your system until they are dropped. When they are dropped, all REXX programs running on your system lose access to these functions. Therefore we recommend that you do not drop functions at the end of your programs. Also, for workstations that use the same external functions on a regular basis, it is wise to write a REXX procedure that registers these functions and invoke that procedure in the STARTUP.CMD. This removes the overhead of registering these functions from your applications. The function RxFuncAdd is used to register external functions. Figure 40 is an example of registering an external function. Here is a description of the three parameters that RxFuncAdd requires:

1. The first parameter is the name your REXX program will use to call the function.
2. The second parameter is the name of the file containing the function.
3. The third parameter is the name of the routine in the file that contains the function.

   RxFuncQuery is a REXX function that checks to see if a function is registered. It returns 0 if the function is already registered.

```
/* Register function SQLDBS located in file SQLAR.DLL, routine SQLDBS  */
if RxFuncQuery('SQLDBS') <> 0 then do         /* if not registered     */
rc = RxFuncAdd('SQLDBS' , 'SQLAR' , 'SQLDBS')
  if rc \= 0 then do
    say "Error registering SQLDBS: rc = " rc
    return
  end  /* Do */
end
```

Figure 40. Register an External Function

The function RxFuncDrop is used to drop external functions. The only parameter required is the name of the function to be dropped. Figure 41 on page 51 is an example of dropping a function.

```
/*Drop function named SQLDBS           */
Call RxFuncDrop 'SQLDBS'
```

Figure 41. Drop an External Function

3.3 Example - Accessing User Profile Management Services

REXX does not have a direct interface to User Profile Management Services APIs. However, by using external functions, a REXX program can take advantage of a C program that has accessed a UPM API. The following example, which is on the diskette, is of a REXX program invoking a C function in a DLL. The C function queries UPM to determine the local user ID name, and returns the name to the calling REXX program. Figure 42 contains the REXX program. The C coding required to make a C function available to REXX is discussed in detail in Chapter 6, " REXX and C" on page 93. Note that in order to run this example on your machine, the UPM DLL must be installed (UPM.DLL). This is because the C function QryUserID is using functions in the UPM DLL.

```
/* Query local userid         */
/* function name is QryUserID */
/* located in QRYRXUSR.DLL    */
/* routine is QryUserID       */
Call RxFuncAdd 'QryUserID', 'qryrxusr', 'QryUserID'

Say QryUserID()
```

Figure 42. UPMUSRID.CMD

3.4 Some Established External Function Packages

There are a number of external function packages produced commercially that are designed to extend the REXX language in specific ways. The most prevalent is the REXXUTILS package, which is a DLL containing functions for manipulating Workplace Shell classes and objects, manipulating OS/2 files and directories, and performing text screen input and output. REXXUTILS is provided with OS/2. See Chapter 4, " REXX Utilities External Function Package (REXXUTIL)" on page 53 for a detailed look at the REXXUTILS package.

The EHLLAPI function in the SAAHLAPI DLL gives REXX programs the ability to invoke EHLLAPI commands. SAAHLAPI.DLL is provided by CM/2.

SQLDBS and SQLEXEC are functions that give REXX programs access to DB2/2 APIs. These functions are provided by DB2/2.

Chapter 4. REXX Utilities External Function Package (REXXUTIL)

REXXUTIL is a Dynamic Link Library (DLL) which provides OS/2 REXX specific functions for manipulating Workplace Shell classes and objects, manipulating OS/2 files and directories, performing text screen input and output and performing OS/2 system commands. The REXXUTILs package is shipped as a part of OS/2 2.1.

This chapter will focus on some of the most useful features in REXXUTILs although some of the utilities are used in examples throughout this book.

The complete description of the commands listed in this chapter can be found in the *OS/2 Command Reference* or the online version which is located in the Information folder on the default desktop.

The functions related to manipulating the Workplace Shell objects are described in detail in Chapter 5, "The Workplace Shell and REXX" on page 69. External function packages are described in detail in Chapter 3, "External Functions" on page 49.

To use a REXXUTIL function in a REXX program, you must first register the function using the function RxFuncAdd. You have a choice of registering one function using RxFuncAdd or you can register all functions by first registering SysLoadFuncs via RxFuncAdd and then use SysLoadFuncs to register all functions in REXXUTIL.DLL. The following is an example of registering one function (SysGetKey):

```
Call RxFuncAdd 'SysGetKey', 'RexxUtil', 'SysGetKey'
```

To register all REXXUTIL functions:

```
Call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs
```

Once the REXXUTIL functions are loaded by SysLoadFuncs they are usable by all OS/2 sessions.

The installation example consisting of three .CMD files: 4199.CMD, CONFUPD.CMD and MAKEFOLD.CMD, give a practical example of the usage of the functions described in this chapter. These .CMD files are used to install the sample programs from the diskette to a hard disk. 4199.CMD begins by registering all the REXXUTIL functions as follows:

```
Call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
Call SysLoadFuncs
```

4.1 Drives, Directories and Files

REXXUTIL has some very useful functions for querying, searching and manipulating files and directories as well as listing and querying drives on your system:

- SysDriveMap
- SysDriveInfo
- SysFileDelete
- SysFileSearch
- SysFileTree
- SysMkDir
- SysRmDir
- SysSearchPath

4.1.1 SysDriveMap

SysDriveMap is used to get information about drives available to your system. SysDriveMap takes two arguments: the drive to start with, defaulting to C: and the report option which can be given a value of:

USED Reports drives which are accessible or in use. This is the default option and includes all local and remote (LAN-attached) drives.

FREE Reports drives which are free or not in use.

LOCAL Reports only those drives which reside on your workstation.

REMOTE Reports only those drives which are remote drives such as redirected LAN resources.

DETACHED Reports drives which are detached LAN resources like a LAN drive that is assigned to your workstation but detached after a timeout.

For example the following code will list all your drive information beginning at the drive given as argument:

```
Arg First
If First='' then First='C:'
Say 'USED:    'SysDriveMap(First,'USED')
Say 'FREE:    'SysDriveMap(First,'FREE')
Say 'LOCAL:   'SysDriveMap(First,'LOCAL')
Say 'REMOTE:  'SysDriveMap(First,'REMOTE')
Say 'DETACHED:'SysDriveMap(First,'DETACHED')
```

If your workstation has local drives C: and D: and LAN-attached drives E: and F: the previous example will display the following information given an argument C:.

```
USED:    C: D: E: F:
FREE:    G: H: I: J: K: L: M: N: O: P: Q: R: S: T: U: V: W: X: Y: Z:
LOCAL:   C: D:
REMOTE:  E: F:
DETACHED:
```

The following code is used in 4199.CMD to check if the drive that the user gives as input is actually usable by the system:

```
Drives:
/* Check if wanted drive in use                                    */

Arg Svalue     /* Search value as argument for example F:    */
Valid = 0

Drives = SysDriveMap('C:', 'USED') /* All used drives              */
If (Pos(Svalue,Drives) = 0) | (Strip(Svalue) = '') then Do Until Valid
  Say 'You have the following Drives:' /* No drive or invalid drive*/
  Say 'Select the number of the drive you wish to install on:'
  Do I = 1 to Words(Drives)       /* Give selection list of        */
    Say I'  ' Word(Drives,I)      /* all accesible drives          */
  End
  Pull Number
  If (Datatype(Number,'W') = 1) & (0 < Number) & (Number <= Words(Drives))
    then Valid = 1
  Else Do
    Say 'You typed in an invalid number 'Number'!'
    Say 'Please choose a number between 1 and 'Words(Drives)
  End
End
If Valid then Rv = Word(Drives,Number)
Else Rv = Svalue

Return Rv
```

4.1.2 SysDriveInfo

SysDriveInfo can be used to give information on total and used space on any disk drive usable by your system. It also gives you the label information. The only parameter given is the drive of interest. Information will be returned in the form of Drive: Free Total Label. If the drive is not accessible a null string is returned. The following code example lists all drives in your system and gives information about space usage and labels.

```
Call Dinfo 'LOCAL'
Call Dinfo 'REMOTE'
Exit

Dinfo:
Arg Where
Drives = SysDriveMap('C:',Where)
Say Where 'DRIVES'
Say Left('DISK',4) Right('FREE_SPACE',12) Right('TOTAL_SIZE',12),
    Right('USED',12) Left('LABEL',20)
Do I = 1 to Words(Drives)
  Parse Value SysDriveInfo(Word(Drives,I)) With Disk Free Total Label
  Say Left(Disk,4) Right(Free,12) Right(Total,12),
      Right(Total-Free,12) Left(Label,20)
End
Return
```

The previous example will give output in the following form:

```
LOCAL DRIVES
DISK   FREE_SPACE   TOTAL_SIZE         USED LABEL
C:       25556992    104634368     79077376    OS2
D:      156990976    210747392     53756416
REMOTE DRIVES
DISK   FREE_SPACE   TOTAL_SIZE         USED LABEL
E:       28792832     52412416     23619584
F:      160555008    318750720    158195712
```

The following is used in 4199.CMD to check if the disk the user wants to install has enough free space.

```
Needed = '252000'
Parse Value SysDriveInfo(Rv) With . Free Size Label
If Free < Needed then Do
  Say 'You do not have enough disk space on disk 'Rv
  Say 'You need 'Needed' bytes, You have 'Free' bytes'
  Say 'You need to free 'Needed - Free' bytes'
  Say 'Exiting installation..'
  Exit
End
```

4.1.3 SysFileDelete

Deletes a file given as a parameter. For example:

```
Rc = SysFileDelete(TempFile) /* Delete temporary file          */
```

4.1.4 SysFileTree

SysFileTree is used to search for files and/or directories. It can also be used to change file attributes of the files that match the search specification. The following example from 4199.CMD uses SysFileTree to find out if the user-entered directory exists. The first parameter is the search specification, for example F:\SAMPLES, and the second parameter, in this case File, is the name of the stem which will hold the output. File.0 will have the number of matching values so if it is 0 then directory was not found. Parameter OD means (O) return only fully qualified name of directory and no additional information. (D) look only for directories.

```
Paths:

Arg Svalue                    /* Path to search for as argument    */
                              /* For example F:\SAMPLES            */
Call SysFileTree Svalue, 'File', 'OD'
If File.0 = 0 Then Do
  Say 'Directory' Svalue 'does not exist'
  Exit
End
Else Return Svalue            /* Directory found                   */
```

4.1.5 SysFileSearch

SysFileSearch can be used to search for text strings in a given file. It takes four arguments: Target, File, Stem and Options. Target contains the text to be searched for, File contains the file to search in and Stem contains the stem that will hold the lines containing the target lines. Stem.0 will contain the number of lines found. If no lines are found then stem.0 will contain 0. The options can contain either C for case sensitive search, N for reporting line numbers of lines matching the search argument or both of these. If no options are given then the search will be case insensitive without line numbers.

The following code from CONFUPD.CMD searches for a given directory in the PATH= statement in CONFIG.SYS.

```
/**/
Parse Source . . Myname        /* Find full path of this .CMD file */
Arg Add2Path                   /* Get path as argument             */
BootDrive=Filespec('Drive',Value('SYSTEM_INI',,'OS2ENVIRONMENT'))
                               /* Find out bootdrive               */
SysFile = BootDrive||'\CONFIG.SYS' /* Full path of CONFIG.SYS      */
/* Get all statements with PATH= from CONFIG.SYS                   */
Rc = SysFileSearch('PATH=',SysFile,'PVALS.','N')
If Rc <> 0 then Do             /* Error opening CONFIG.SYS         */
  Say 'Could not open 'SysFile
  Say 'Error number from SysFileSearch = 'Rc
  Exit
End
Found = 1                      /* Set found to TRUE                */
Do I = 1 to PVals.0            /* Step through found lines         */
  If Pos(' PATH=',PVals.I) > 0 then Do
    /* PATH statement (notice blank in front to distinguish from   */
    /* from DPATH, LIBPATH etc.                                    */
    If Word(PVals.I,2) = 'REM' then Iterate I /* Remark, no action */
    If Pos(';'Add2Path';',PVals.I) <> 0 then nop
                               /* Already exists in PATH           */
    Else Do
      NewPath = Strip(Pvals.I,'T',';')||';'Add2Path';'
      /* Strip trailing ; and concatenate new path with trailing ; */
      Parse Var NewPath Lineno NewPath /* Separate line number and */
      Found = 0                        /* actual string           */
    End
  End
End
If Found Then Exit             /* Already in path, no need to add  */
Else Call Update               /* Update CONFIG.SYS with new path  */

Exit
```

4.1.6 SysMkDir

SysMkDir can be used instead of the OS/2 command MD to create a directory. SysMkDir will give a return code 0 if the operation is successful. Without using SysMkDir your program could only check for a successful operation by redirecting the STDERR output to RXQUEUE and then pulling the error output from the queue. Following is an example from 4199.CMD of using SysMkDir to create a directory.

```
CreateDir:

Rc = SysMkDir(Svalue)          /* Create directory                */
If Rc = 0 then Rv = Svalue
Else Do
  Say 'Could not create directory' Svalue
  Exit
End

Return Rv
```

4.1.7 SysSearchPath

SysSearchPath can be used to search for a specific file in all directories listed in PATH, DPATH, LIBPATH etc. For example:

```
View = SysSearchPath('PATH','VIEW.EXE') /* Search all directories  */
                                        /* in path for VIEW.EXE    */
```

Will return C:\OS2\VIEW.EXE if VIEW.EXE is located in C:\OS2 and that directory exists in the PATH statement.

4.2 Workplace Shell Objects

REXXUTIL has some very useful functions for creating and destroying objects and altering the properties of objects. These functions include:

- SysCreateObject
- SysDeregisterObjectClass
- SysDestroyObject

- SysGetEa
- SysIni
- SysPutEa
- SysQueryClassList
- SysRegisterObjectClass
- SysSetIcon
- SysSetObjectData

This section will discuss some of these functions as they are used in MAKEFOLD.CMD. Chapter 5, “The Workplace Shell and REXX” on page 69 describes the usage of these functions in interacting with the Workplace Shell in more detail.

4.2.1 SysCreateObject

MAKEFOLD.CMD is used as a part of the installation program to create a folder on the desktop and then fill that folder with the .CMD, .EXE and .INF files from the samples disk with different icons for each type of object.

The first part of the program sets up some variables used later on in the program. The Directry variable is passed to the program as argument and contains the physical path of the objects. Fid contains the object id for the folder and is used both in creating the folder and populating the folder with program objects. Variable View will contain the full path of VIEW.EXE. The rest of the variables contain the Icon and Bitmap files. The setup variables are as follows:

```
bmap = Directry||'\4199.BMP'          /* Background bitmap          */
SampIcon = Directry||'\4199SAMP.ICO' /* Folder icon                */
CmdIcon = Directry||'\4199RX.ICO'     /* .CMD file icon             */
VisIcon = Directry||'\4199VIS.ICO'    /* Icon for visual builders   */
RedIcon = Directry||'\4199RED.ICO'    /* .INF icon                  */
Fid  = '4199FOLD'                     /* Folder id                  */
View = SysSearchPath('PATH','VIEW.EXE') /* Search all directories  */
                                        /* in path for VIEW.EXE    */
```

The second part creates the folder on the desktop for the sample programs and .INF files.

The first parameter holds the object class for the folder. OS/2 has an object class WPFolder for folder objects, which has preset object properties such as

button appearance, object open behavior, font information etc. so we use that in creating the folder.

The second parameter is the name of the folder. This is not the object ID, but the name which will appear in the title bar of the object, the icon of the object, the Window List etc.

The third parameter describes the location where the folder will be created. There are many locations on the standard desktop to choose from like <WP_INFO>, <WP_GAMES> etc. but as we want to install the folder to the desktop we choose <WP_DESKTOP>.

In the fourth parameter we give the folder an ID which will allow us to reference the object later when installing objects to the folder. Following is the SysCreateObject command for creating the folder:

```
Rc  = SysCreateObject('WPFolder',,          /* Object class         */
                      'GG24-4199 Samples',, /* Name of folder       */
                      '<WP_DESKTOP>',,      /* Location of folder   */
                      'OBJECTID=<'Fid'>')   /* Object id for folder */
                                            /* 4199FOLD             */
```

4.2.2 SysSetObjectData

SysCreateObject allows you to set additional properties through the setup string which is described in Appendix C, “OS/2 Workplace Shell Setup Strings and Color Definitions” on page 269. These properties can also be set by SysSetObjectData. Following is a piece of code which will set the icon for the folder (ICONFILE='SampIcon) open the folder (OPEN=DEFAULT), set the background bitmap (BACKGROUND='bmap) and set the font associated with the icons in the folder to Courier Bold 14 (ICONFONT=14.Courier Bold).

```
/* Set object data for folder                                        */
Rc = SysSetObjectData('<'Fid'>',,               /* Object id        */
                      'ICONFILE='SampIcon,      /* Iconfile         */
                      ';OPEN=DEFAULT'||,        /* Open folder      */
                      ';BACKGROUND='bmap||,     /* Background bitmap*/
                      ';ICONFONT=14.Courier Bold'x2c(0)) /* Font    */
```

The next step is to populate the folder with various program objects. There is an object class called WPProgram in OS/2 which can be used in creating program objects. You can give additional parameters to program objects like PROGTYPE and EXENAME. Progtype describes the type of program. PM means a PM program and WINDOWABLEVIO means windowable program. Both of these can be associated with .CMD files. EXENAME holds the full path of the program, for example C:\REXX\GEA.CMD. The following is an example from MAKEFOLD.CMD to populate the samples folder with REXX .CMD files.

```
'@DIR /F /B 'Directry'\*.CMD | RXQUEUE' /* List .CMD files            */
Do Queued()                             /* Process queued filenames   */
  Pull Name                             /* Pull name of file          */
  Parse Var Name Name'.'Ext             /* Don't use extention in name*/
  Rc = SysCreateObject('WPProgram',,    /* Create program object      */
                        Name,,          /* Name of object             */
                       '<'Fid'>',,      /* Location (folder)of object */
                       'PROGTYPE=PM',   /* PM application             */
                       ';ICONFILE='CmdIcon,    /* Iconfile           */
                       ';EXENAME='Directry'\'Name'.'Ext,,
                                        /* Full path of command file  */
                       'f')             /* Fail if already exists     */
End
```

Note

If you want to extend your SysCreateObject call to several lines you can use double commas like in the previous example. In the setup string you should also use double vertical bars to ensure that the strings are concatenated without extra blanks like in the following code from MAKEFOLD.CMD to populate the samples folder with the .EXE files from the samples diskette.

```
'@DIR /F /B 'Directry'\*.EXE | RXQUEUE'
Do Queued()
  Pull Name
  Parse Var Name Name'.'Ext
  Rc = SysCreateObject('WPProgram',,
                        Name,,
                       '<'Fid'>',,
                       'PROGTYPE=PM',
                       ';ICONFILE='VisIcon||,
                       ';EXENAME='Directry'\'Name'.'Ext)
End
```

In the SysCreateObject setup string it is also possible to give parameters to the program object with PARAMETERS=. The following code creates an object for VIEW.EXE with an .INF file as a parameter.

```
'@DIR /F /B 'Directry'\*.INF | RXQUEUE'
Do Queued()
  Pull Name
  Rc = SysCreateObject('WPProgram',,
                        Name,,
                       '<'Fid'>',,
                       'PROGTYPE=PM',
                       ';ICONFILE='RedIcon||,
                       ';EXENAME='View||,
                       ';PARAMETERS='Directry'\'Name,,
                       'r')
/* VIEW.EXE with .INF as parameter                                  */
End

EXIT
```

4.3 Miscalleneus Functions

Other useful REXXUTIL functions for screen and keyboard I/O and other OS/2 specific functions include:

- SysCls
- SysCurPos
- SysCurState

- SysGetKey
- SysSleep
- SysTextScreenRead
- SysTextScreenSize

These functions will be explained here using a directory list utility, FLIST.CMD, that allows you to list a directory, scroll backwards and forwards and edit a file selected with cursor.

4.3.1 SysCls

SysCls clears the screen:

```
Call SysCls
```

4.3.2 SysCurPos

SysCurPos positions the cursor to a specified location on a text screen. It takes two arguments: Row and Column. For example:

```
Row = 1
Rc = SysCurPos(Row,Length(File.1))          /* Position cursor to end  */
                                            /* of first entry          */
```

4.3.3 SysCurState

SysCurState allows you to change the cursor state. It can give an argument ON which will display the cursor, or OFF which will hide it. For example:

```
Call SysCurState 'OFF'                      /* Hide cursor             */
```

4.3.4 SysGetKey

SysGetKey is one of the most useful REXXUTIL functions. It allows you to intercept keystrokes from the keyboard buffer or wait for a keystroke. It can also suppress display of the character given the argument NOECHO.

Extended keystrokes place two values in the keyboard buffer the first containing either decimal 0 or decimal 224. So in order to query an extended key you must issue two SysGetKey calls. Following is an example from FLIST.CMD to query some keys used in the directory listing:

```
Getkey: Procedure
  Key = c2d(SysGetKey('noecho'))            /* Get keystroke          */
  If Key=0 | Key=224                        /* Extended keystroke     */
    Then Ext_Key = c2d(SysGetKey('noecho'))
  Select
    When Key = 27  then key = 'ESC'
    When Key = 0 | Key = 224 Then Select    /* Second value tells key  */
      When Ext_Key = 59  Then Key = 'F01'
      When Ext_Key = 60  Then Key = 'F02'
      When Ext_Key = 61  Then Key = 'F03'
      When Ext_Key = 62  Then Key = 'F04'
      When Ext_Key = 80  Then Key = 'AR_DN'
      When Ext_Key = 72  Then Key = 'AR_UP'
      When Ext_Key = 79  Then Key = 'END'
      When Ext_Key = 71  Then Key = 'HOME'
      Otherwise nop
    End /* select */
    Otherwise nop
  End /* select */
Return Key
```

4.3.5 SysSleep

SysSleep allows the system to wait a specified number of full seconds which are given as an argument. For example:

```
Call SysSleep 3                           /* Wait 3 seconds         */
```

4.3.6 SysTextScreenRead

Reads a specified number of characters from a specified location of the screen. Can be given three arguments: row, column and length. For example:

```
/* Read in the entire screen                  */
screen = SysTextScreenRead( 0, 0 )
/* Reads in one line starting from second row */
line = SysTextScreenRead( 2, 0, 80 )
```

4.3.7 SysTextScreenSize

Gives the size of a text screen for example an OS/2 window. Returns the number of rows and columns. For example:

```
Parse Value SysTextScreenSize() With Rows Cols
```

Chapter 5. The Workplace Shell and REXX

The Workplace Shell (WPS) is a sophisticated user interface to OS/2 that utilizes the object-oriented concept. It uses a desktop, objects, and icons to represent the makeup of your system. The desktop is the background of the screen that contains the visual representation of your system. Objects refer to things such as application programs, data files, directories, and devices. Icons are the visual representation of objects on the desktop. The Workplace Shell can be customized to fit particular needs. The REXXUTIL external function package contains functions that are very useful in Workplace Shell customization. This chapter takes a look at these REXX functions, and different ways that they can be used to manipulate the Workplace Shell.

5.1 Objects and Object Classes

Since the Workplace Shell is an object-oriented system, it is important to have some knowledge of objects and object classes. All objects are members of an object class. Object classes define specific properties that objects in the class inherit. The structure is such that object classes can have subclasses. The concept is similar to classification of animals, so this analogy is often used to describe objects and object classes.

We can say that Mammals is an object class. All objects that are in the Mammal class have certain properties. For example, they are warm blooded. Subclasses of Mammals could be: Dog object class, and Man object class. The Dog class has certain properties. For example, they have four legs, and they bark. The Man class has certain properties. For example they have two legs.

Figure 43. The Mammal Class Hierarchy

Figure 43 is a representation of the object class hierarchy we just described. Object classes are categorizations based on properties that are used to define types of objects.

Objects are the actual instances of the object class. For example, the golden retriever down the street is an object of the Dog class. We know it has four legs and barks. We also know it is warm blooded.

Now let us look at the object classes that have been defined for the Workplace Shell. Figure 44 is a representation of the Workplace Shell class hierarchy. These are the default object classes. Additional object classes can be defined to the Workplace Shell using the REXXUTIL function SysRegisterObjectClass. Object classes are actually DLLs that define the properties of the class. Note that all object classes are subclasses of object class WPObject. Note that there are three major object classes below WPObject:

- WPFileSystem
- WPAbstract
- WPTransient

```
WPObject
  WPFileSystem           WPAbstract              WPTransient
    WPDataFile             WPClock                 WPCnrView
      WPBitmap             WPCountry                 WPDiskCV
      WPIcon               WPDisk                    WPFolderCV
      WPPointer            WPKeyboard              WPFilter
      WPProgramFile        WPMouse                   WPFinder
        WPCommandFile      WPPalette               WPMinWindow
      WPMet                  WPSchemePalette       WPJob
      WPPif                  WPColorPalette        WPPort
    WPFolder                 WPFontPalette         WPPrinterDriver
      WPDesktop            WPProgram               WPQueueDriver
      WPStartup            WPPrinter
      WPDrives               WPRPrinter
      WPMinWinViewer       WPShadow
      WPFindFolder           WPNetLink
      WPNetgrp             WPShredder
      WPNetwork            WPSound
      WPServer             WPSpecialNeeds
      WPSharedDir          WPSpool
      WPTemplates          WPSystem
      WPRootFolder         WPPower
```

Figure 44. The Workplace Shell Object Class Hierarchy

5.1.1 WPFileSystem

Objects that are in the WPFileSystem object class structure typically represent files or directories that are located on the hard disk. For example, objects of the WPFolder class represent directories in the file system. Objects of the WPDataFile class represent files. For example, the file OS2LOGO.BMP is an instance of the object class WPBitmap, which is a subclass of the WPDataFile object class. WPFileSystem objects store specific information about their properties in the extended attributes of the object file. An example of the type of data stored in the extended attributes is the icon that represents a file. The extended attributes of a file are accessible by REXX using the REXXUTIL functions SysGetEA, SysPutEA and SysSetIcon.

5.1.2 WPAbstract

WPAbstract objects do not necessarily represent files on your hard disk. They may reference, or point to, files however. We will see this in more detail later. WPAbstract objects can represent things like device drivers and system settings. They can contain information about the configuration and customization of your workstation. For example, the mouse driver is an object of the class WPMouse. They can also represent executable programs. They may represent references to applications installed on your system. Since WPAbstract objects do not necessarily have a file associated with them, specific information about their properties are stored in the system initialization files, OS2.INI, and OS2SYS.INI.

OS2.INI, called the user INI, and OS2SYS.INI, called the system INI, are used by OS/2 on startup of a workstation to determine the configuration of the workstation. The OS2.INI file contains information that the user may want to change. Things like colors, country settings, and mouse settings are stored here. The REXXUTIL function SysIni provides a way to read and write OS2.INI data. In addition, the SysCreateObject, SysSetObjectData, and SysDestroyObject REXXUTIL functions can manipulate WPAbstract objects by updating the OS2.INI.

5.1.3 WPTransient

Objects of the class WPTransient may contain specific data about their properties, but that data does not need to be saved by the Workplace Shell. As such, there are no REXXUTIL functions to access object data for WPTransient objects. An example of a WPTransient object is a print job.

5.2 Creating Objects

SysCreateObject is a REXXUTIL function that can be very effective in customizing the Workplace Shell. Typically the classes of objects that are manipulated with SysCreateObject are WPFolder, WPProgram, and WPShadow of the WPAbstract class. Here is an example to show you how SysCreateObject can be used to customize a desktop. Remember that to use external functions, they must be registered using RxFuncAdd.

The PC Company of IBM has asked us to write a REXX program that will create the following:

- PC Company folder on the desktop
- PC application in the PC Company folder
- Shadow of an OS/2 command prompt on the desktop for easy access

Using REXX to create new objects is easy if you use a fill in the blank approach. Whether creating a folder object, a program object, or one of many other objects, certain blanks must be filled in, and then it is just a matter of typing. The hard part is figuring out what needs to go in the blanks. This will become much clearer as we go through a few examples.

5.2.1 Creating a Folder Object

Following are the blanks that must be filled in for the SysCreateObject function:

- Classname
- Title
- Location
- Setup
- Duplicateflag

As requested by the PC Company, we must create the PC Company folder on the desktop. Let's walk through this first example, remembering the blanks to fill in. If necessary, refer to Appendix C, "OS/2 Workplace Shell Setup Strings and Color Definitions" on page 269 for the setup string parameters in the WPFolder class.

1. The first blank is classname. Since we are trying to create a folder, fill in this blank with WPFolder.
2. The second blank is the title, which has already been given: PC Company^Folder (the caret separates these on two lines).

3. The third blank, location, also has been given to us: they requested it to be on the desktop. Therefore, put the object ID of the Desktop, <WP_DESKTOP>, into the location blank. An object ID is any string that begins with < and ends with >. The object ID, as we shall see later in this chapter, is a unique identifier for an object. It is needed to locate and to make changes to objects.
4. The fourth blank, setup (which is optional), is usually simple when creating a folder. Give the folder the object ID <PC_COMPANY_FOLDER>. Note that the underscores could have been spaces, but underscores are usually used for consistency with the existing object IDs.
5. The fifth and final blank, duplicateflag (also optional), is used to determine what to do if the object already exists. Here are the options for duplicateflag. The default is FAIL.

- REPLACE - Delete the old object and create a new object
- FAIL - Will not do anything if object already exists
- UPDATE - Update an object
- DELETE - Delete an object

That's all it takes to create a new folder on your desktop. Figure 45 shows the REXX code.

```
/* Fill in the  blanks  as parameters to pass to SysCreateObject    */

classname='WPFolder'
title=    'PC Company^Folder'
location= '<WP_DESKTOP>'
setup=    'OBJECTID=<PC_COMPANY_FOLDER>;'

result = SysCreateObject(classname, title, location, setup, 'f')
```

Figure 45. Create Folder Object

5.2.2 Creating a Program Object

Now that the PC Company folder is on the desktop, the next step is to create the PC application inside that folder. Refer to Appendix C, "OS/2 Workplace Shell Setup Strings and Color Definitions" on page 269, which shows the setup string parameters for the WPProgram class. WPProgram objects are references, or pointers, to programs residing on disk. We need to decide how to fill in the blanks to create this program object, just as we did when creating the folder.

1. The classname needs to be WPProgram since we will be creating a program object.
2. The title is easy: use PC Application.
3. Since we want to put the program object inside the folder that we created, use that folder's object ID, <PC_COMPANY_FOLDER>.
4. The setup field for a program object has a few more options than for a folder object. First, let's give the program object the object ID <PC_APPLICATION>.
5. Next, tell the program object the path and name of the program that we want it to run. Instead of writing a real C&C application, use the OS/2 command processor. For the EXENAME field, use C:\OS2\CMD.EXE (assuming that OS/2 is installed on the C: drive).
6. The last field, PROGTYPE, tells what kind of program it is. Set this up as an OS/2 window; use WINDOWABLEVIO as the value.

That is all we need to do to create this program object. Figure 46 shows the REXX code for creating this program object.

```
/* Fill in the  blanks  as parameters to pass to SysCreateObject.    */

classname='WPProgram'
title=    'PC^Application'
location= '<PC_COMPANY_FOLDER>'
setup=    'OBJECTID=<PC_APPLICATION>;'||,
          'EXENAME=C:\OS2\CMD.EXE;'||,
          'PROGTYPE=WINDOWABLEVIO;'

result = SysCreateObject(classname, title, location, setup, 'f')
```

Figure 46. Create Program Object

5.2.3 Creating a Shadow Object

To finish the request from PC Company, one more task must be completed, and that is to create a shadow of the OS/2 window command prompt and place it on the desktop for easy access. The SysCreateObject function will be used again, but with WPShadow as the classname.

WPShadow is the object class name for shadow objects. Shadow objects are pointers to files or other objects. They contain the location of the object that they are shadowing. Manipulation of the shadow object is reflected in the object that it is pointing to. The only actions you can take on a shadow object that do not affect the original object are move, copy, and delete. Shadow objects are very useful when working with data files. We will see that later in this chapter. A shadow is probably the easiest object to create. Let's go through the program and fill in the blanks.

1. The classname needs to be WPShadow since we are creating a shadow.
2. The title can be anything, so call it OS/2 Window^Shadow. Again, the caret symbol separates the first line from the second.
3. .The location field shows where to place the object, so use <WP_DESKTOP> since we want the shadow to be placed on the desktop.
4. The SHADOWID field is asking what object we want to have shadowed. In this case, we are creating a shadow of the OS/2 window command prompt, for which the object ID is <WP_OS2WIN>.

Figure 47 shows the REXX code to create a shadow object.

```
/* Fill in the  blanks  as parameters to pass to SysCreateObject */

classname='WPShadow'
title=    'OS/2 Window^Shadow'
location= '<WP_DESKTOP>'
setup=    'SHADOWID=<WP_OS2WIN>'
result = SysCreateObject(classname, title, location, setup, 'f')
```

Figure 47. Create a Shadow Object

5.2.4 Creating a Program Object in the Startup Folder

Programs that are in the Startup folder are executed whenever the system is initialized. This is useful if you wish to have a system come up and immediately start an application, for example. For a workstation that will be running REXX programs, it is convenient to register external functions during the system startup. Remember, once external functions are registered, they are available to all REXX programs running on the system as long as they are not dropped. Program WPSREG.CMD, shown in Figure 48, creates a program object on the desktop. The program that it references registers the REXXUTIL function package. It then creates a shadow of that program object in the Startup folder.

The program REGFUNC.CMD and shown in Figure 49 registers the REXXUTIL external function package.

```
/* create program object for regfunc.CMD          */
classname = 'WPProgram'
title='Register REXXUTIL'
location='<WP_DESKTOP>'
setup='OBJECTID=<REGFUNC_CMD>;'||,
         'PROGTYPE=WINDOWABLEVIO;'||,
         'EXENAME=E:\REXXUTIL\REGFUNC.CMD;'

result = SysCreateObject(classname,title,location,setup, 'U')

/* create shadow of that program object           */
classname = 'WPShadow'
title= 'Register REXXUTIL'
location='<WP_START>'
setup='SHADOWID=<REGFUNC_CMD>;OBJECTID=<SHAD_REGFUNC_CMD>'

result = SysCreateObject(classname,title,location,setup, 'U')
return
```

Figure 48. Create Program Object in Startup Folder - WPSREG.CMD

```
/* register REXXUTIL functions        */
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs
return
```

Figure 49. REGFUNC.CMD

5.3 Creating Drag and Drop REXX Programs

Dragging a data file with the mouse and dropping it onto a program reference object is a feature of the Workplace Shell. For example, the Shredder object utilizes this feature. To discard data, objects can be dragged to and dropped on the Shredder object. When the mouse button is released, the program referenced by the shredder object attempts to delete files associated with the object. In this section we will show you how to create a WPProgram object referring to a REXX program so that it can handle drag and drop.

REXX program objects are created the same way other program reference objects are created. The EXENAME parameter of the setup string is set to the path and filename of the REXX program. To allow the drag and drop feature to occur, we need to associate files to the program object. Associating a file to a program in this way means that the program will accept the file name as an input parameter when the file′s object is dropped on the program′s object. This means that the program must be written to accept a file name as the incoming parameter.

To associate a file or group of files to a program object we can use the ASSOCTYPE or ASSOCFILTER parameters in the setup string. They are discussed in more detail in Appendix C, “OS/2 Workplace Shell Setup Strings and Color Definitions” on page 269. For example, ASSOCTYPE = ″Plain Text″ means that the program can accept any ASCII text file as input. Wildcards can be used to specify groups of file types. For example, ASSOCTYPE = *.CMD means that only files with the extension .CMD are associated with this program object. Figure 50 on page 78 shows the REXX code for creating a REXX program object that plain text files can associate to. Plain text files can be dropped on the GEA.CMD program reference object. GEA.CMD reads the extended attributes of the input file name and loads the icon data into the icon editor. Any changes made to the icon are then loaded back into the extended attributes. The details of this progam will be discussed in :hdref=wpsea.. Since the code is on the diskette, you can try out the drag and drop feature with this program. Note that you will not be able to drag and drop objects that do not reference plain text files. For example, the Templates folder cannot be dropped on the GEA.CMD program object since it does not represent a plain text file.

```
classname = 'WPProgram'
title='Update Icon'
location='<WP_DESKTOP>'
setup='OBJECTID=<GEA_CMD>;'||,
         'PROGTYPE=WINDOWABLEVIO;'||,
         'EXENAME=E:\REXXUTIL\GEA.CMD;'||,
         'ASSOCTYPE=Plain Text;'

result = SysCreateObject(classname,title,location,setup, 'U')
```

Figure 50. Associate Files to Program Object - WPSDRAG.CMD

5.4 Creating (Shadow) Objects Associated With Data Files

There are a number of ways to represent data files in the Workplace Shell.

To create a shadow object that references a data file you need to supply the physical location of the data file. The shadow object will exist on the desktop, but the data file it refers to can exist anywhere on the system. This allows you to physically store data files in subdirectories where it makes sense logically, yet still be able to represent data files on the desktop. The shadow object can now be used to manipulate that data file, as if the data file was actually on the desktop. Figure 51 shows the REXX code to create a shadow of a data file.

```
classname = 'WPShadow'
title= 'TESTFILE'
location='<WP_DESKTOP>'
setup='SHADOWID=E:\REXXUTIL\TESTFILE.DAT;OBJECTID=<SHAD_TESTFILE>'

result = SysCreateObject(classname,title,location,setup, 'U')
```

Figure 51. Create Shadow of a Data File

5.5 Modifying Workplace Shell Objects

Once you have created objects, you will almost always come across situations where you need to change the data associated with those objects in some way. There also may be system or application objects that you did not create that you need to modify in some way. The REXXUTIL functions SysCreateObject, SysSetObjectData, SysIni, and SysDestroyObject are very useful in customizing Workplace Shell objects. We have found that there are instances where a number of these functions could be used to accomplish the same thing. For example, both SysSetObjectData and SysCreateObject (with duplicateflag set to UPDATE) can set an icon to a object, among others. Since we are dealing with WPAbstract objects here, and their data is stored in the INI files, these four functions can all be used to access and modify INI data. Before we get into a discussion about modifying Workplace Shell objects, we need to talk about object IDs, RC files, and INI files.

5.5.1 Object IDs

An object ID uniquely identifies an object to the Workplace Shell. It is most often used with REXX, since you can only modify an object using its object ID. One exception to this is a file system object. If the object is a folder, which is a subdirectory in the file system, it can be identified using its fully qualified path name instead of an object ID. Each base operating system object on an OS/2 2.1 desktop has an object ID. Objects that belong to other applications may or may not have object IDs, depending on the installation program of that particular application. There are several ways to identify the object ID for a particular object. One method is to look into the \OS2\INI.RC file to see the base OS/2 object IDs and other information. This will not contain information for object IDs added to the system after the initial system installation. The \OS2\OS2.INI file contains information for all objects in the Workplace Shell. This includes objects added to the Workplace Shell by application installations, as well as objects that you have created. SysCreateObject writes information to the \OS2\OS2.INI.

5.5.2 RC Files

The \OS2\INI.RC file contains information on all base OS/2 object IDs. This RC file is used to initially create the \OS2\OS2.INI file, which is used to set up the Workplace Shell. Although there are many lines in this file, we are interested in the ones that begin with PM_InstallObject. Each line defines how to create an individual object that is installed with the base operating system. For example, the following line says to install an object with the title System Clock, with the class of WPClock, in the folder that has an object ID

of <WP_CONFIG> (the System Setup folder), and assign this object the object ID of <WP_CLOCK>:

```
"PM_InstallObject" "SystemClock;WPClock:<WP_CONFIG>""OBJECTID=<WP_CLOCK>"
```

By understanding how this one object is created, you should be able to review the PM_InstallObject lines to understand how all the objects get installed. Additionally, you can identify the object ID for any given object. Since this file contains readable text, REXX programs can write to and read from the RC file just like any other text file.

5.5.3 User INI File

There is another way to find information about object IDs that will work for all base operating system objects as well as any object ID that has been added since the initial installation. The user INI, \OS2\OS2.INI, a binary file that cannot be easily read, contains much information about the system configuration, including object IDs. The REXXUTIL function SysIni allows you to read and write INI data. Figure 52 on page 81 shows a sample program that displays all object IDs on a system by reading the \OS2\OS2.INI file. It uses SysIni to read this information. Refer to 5.7, “SysIni” on page 84 for a discussion on writing to INI files and reading from the INI files using SysIni. This method of finding object IDs is important. It is the only one that enables you to see the object IDs for objects other than the operating system objects, including the ones you have created. You should now understand how to find object IDs for all base operating system objects. You also have a program that can be run on any OS/2 2.1 system to identify the object IDs on that system. It is included in the MLAMBDLL.INF file on the diskette.

```
/* List ObjectIds                                                */
App='PM_Workplace:Location'
call SysIni 'USER', App, 'All:', 'Keys'
if Result \= 'ERROR:' then do
   Call SysCls
   Say '';Say '';Say 'Listing ObjectId information';Say '';
   parse value SysTextScreenSize() with row col
   j=row-10
   Do i=1 to Keys.0
      If trunc(i/j)==i/j Then Do
         Say '';Say 'Press any key to show next screen...'
         key=SysGetKey()
         Call SysCls
         Say '';Say '';Say 'Listing ObjectId information';Say '';
      End
      Say Keys.i
   End
End
Else Say 'Error querying for' App
Return
```

Figure 52. REXX Program to Display All Installed Object IDs

5.5.4 SysSetObjectData

The SysSetObjectData function is a powerful tool that can modify the settings of an existing object. The settings that you can modify can be placed in the setup string of a SysSetObjectData call. The following four examples illustrate the power of this function.

For the first example, open any object on the desktop by specifying

```
OPEN= DEFAULT
```

and an object ID (or a fully qualified path name). Figure 53 on page 82 shows sample code to open the **Start Here** object. It would be easy to modify the code so that it can accept a string as a parameter passed to it. By doing this, you can have a simple program that can open any object from the command line.

```
/*  REXX code to open an object          */

object=   '<WP_STHR>'
setup=    'OPEN=DEFAULT;'

result = SysSetObjectData(object, setup)
```

Figure 53. SysSetObjectData to Open an Object

A second example uses REXX to help system performance. Try opening the **Templates** folder. If it is set to the default **Non-grid**, it will take some time for it to open, even on the fastest systems. Now, open the settings and change the **Icon view** to **Flowed**. See how long it takes to open the folder. The difference should be visible (above and beyond the expected performance increase of opening a folder immediately after closing, since the data is likely to still be in the cache). Figure 54 shows the REXX code to change the icon view setting to flowed.

```
/* FLOWOBJ.CMD   REXX program to set Icon View to Flowed   */

result = SysSetObjectData('<WP_TEMPS>', 'ICONVIEW=FLOWED')
```

Figure 54. SysSetObjectData to Change Icon View Setting

The third example of the SysSetObjectData function provides some security for the WPS. Security refers to protecting end users from making mistakes rather than protecting against theft, vandalism, and so on. Many users have requested this type of protection. One way to provide it is through REXX. For example, it can be very dangerous to test drag-and-drop techniques when using the **Shredder** as the target. However, a new user may not be aware of this. One solution is to protect a specific item from being deleted. Figure 55 shows a program that does exactly that; it marks the OS/2 **Command Reference** object as undeletable.

```
/* NoDelObj.CMD   REXX program to set NODELETE=YES */

object=   '<WP_CMDREF>'
setup=    'NODELETE=YES;'

result = SysSetObjectData(object, setup)
```

Figure 55. SysSetObjectData to Make an Object Undeletable

Hiding objects is another form of protection. Some OS/2 systems are designed for end users who run only two or three applications. Some administrators have requested a way to protect the user from accessing certain objects on the desktop that they have no need to use. One way to keep the user from accessing an object is to delete it. However, this is undesirable because the administrator may later want to customize the desktop further and if the object was deleted he would be unable to do that. A better solution is to make the object invisible by hiding it. This is easy to do with REXX and the SysSetObjectData function. When the administrator wants to modify the system, it is also easy to make an object visible. Figure 56 shows an example that can be used to hide the **Shredder** object. To reverse this procedure, change NOTVISIBLE to NO and run the program again.

```
/* HIDEOBJ.CMD   REXX program to hide an object */

object=   '<WP_SHRED>'
setup=    'NOTVISIBLE=YES;'

result = SysSetObjectData(object, setup)
```

Figure 56. SysSetObjectData to Hide an Object

Icons are pictures used to represent objects on the desktop. Icon data can be stored in the extended attributes of a WPFileSystem object. This method of storing icon data will be addressed in more detail in 5.8, "Extended Attributes" on page 89. Objects can also be associated with icons. Icon files have the file extension .ICO. They contain binary data that represents a picture of some form. SysSetObjectData can be used to associate an object to an icon file. This is especially useful for WPAbstract objects. Figure 57 shows an example of associating an object to an icon file.

```
/* Associate Workframe/2 Icon to Sample Folder Object  */

rc = SysSetObjectData('<DB2/2_SAMPLE_FOLDER>','ICONFILE=E:\IBMWF\WF.ICO;')
```

Figure 57. SysSetObjectData to Associate Object with Icon

5.6 Moving Objects

Once an object has been created, it may be necessary at some point to move the object from one folder to another. There is no simple command for moving the location of an object. Instead, the process for moving an object′s location consists of three steps:

1. Save the current settings.
2. Destroy the object.
3. Create the object in the new location.

Saving the current settings refers to the settings that are in the user INI for the object. These settings were most likely placed in the user INI by the SysCreateObject function and possibly updated by with the SysSetObjectData function, using the setup string parameters. You will need this data when you recreate the object in the new location. The question is: How do you get the object′s settings data out of the user INI? There are a couple of ways. The first is to use the SysIni function to retrieve the data. This requires knowledge of the keys applicable to the object, and manipulation of hex data to get the information in readable form. Another method is to use a Workplace Shell management product, such as DeskMan/2**. DeskMan/2 has a backup feature that will allow you to recreate any object on the desktop. To do this, DeskMan/2 generates REXX code that retrieves the object′s settings data from the user INI and builds a SysCreateObject function call. This REXX code is available to the DeskMan/2 user. Since DeskMan/2 develops the REXX code to recreate the object in the same location, all you need to do is alter the REXX code to reflect the new location.

Before invoking the code to recreate the object in the new location, the current object should be destroyed. This can be accomplished by using the SysDestroyObject function. This function requires the object ID of the object that is to be destroyed. For the syntax of SysDestroyObject, refer to Appendix A, “REXX Syntax Diagrams” on page 225.

5.7 SysIni

The \OS2\OS2.INI file, or user INI, contains many OS/2 customization and configuration parameters. An associated file \OS2\OS2SYS.INI, or system INI, stores information about printers and other hardware devices. Both are binary files and cannot be edited with a normal text editor. However, both files are compiled from text versions of the files \OS2\INI.RC and \OS2\INISYS.RC respectively. Before beginning to work with these INI files,

you should understand their significance. These two files and the desktop subdirectory contain most of the information about the entire Workplace Shell desktop. The REXXUTIL function SysIni can be used to both read information from the INI files and write information to the INI files.

Note

Be sure that you have backups of the two INI files before beginning to work with them.

5.7.1 Using SysIni to Change System Settings

How can we use the SysIni function to help complete customization of the desktop? The SysIni function is easy to use; the difficult part is knowing what values to supply to pass as parameters to the function. For example, one of the first things most users change is the background color of their desktops to something other than the default gray. Once the parameters are known, this is easily accomplished, as shown in Figure 58 on page 86. Note that some changes made to the INI file will be visible only after a reboot and reinitialization of the Workplace Shell. For example, the background color change requires a reboot to take effect. Also, the three numbers passed as the value represent the RGB values for the desired color. The easiest way to determine these numbers is to change the desktop color of the background manually, following these steps:

1. On the desktop, use the right mouse button to bring up the System Menu.
2. Select **System setup**. This opens the **System Setup** folder.
3. Double click on the **Color Palette** icon.
4. Select **Edit color...**
5. Select **Values>>**
6. The RGB values are displayed.

Appendix C, “OS/2 Workplace Shell Setup Strings and Color Definitions” on page 269 contains the RGB values for the 16 default solid colors of OS/2 2.1 if you wish to use those.

```
/* Note: SysIni calls do not take effect until reboot         */

/* The three numbers for the value represent the RGB values   */

application= 'PM_Colors'
keyname=     'Background'
value=       '002 217 217'

call SysIni 'USER', application, keyname, value
```

Figure 58. SysIni to Change Desktop Background Color

The SysIni function also enables you to modify objects that are located in the System Setup folder within the OS/2 System folder. The system object is one of the most important objects because it has a Settings Notebook that controls the way OS/2 runs, including confirmations, default controls for the windows, print screen, and so on. Examples of objects that can be changed with the SysIni function follow.

First, some people like to disable the PrintScreen function on the keyboard, since they don't normally use it and they sometimes press it by mistake. This wastes both processor time and the printer's toner, so it is preferable to turn it on only when needed. Another performance aid is to turn off the animation feature. Animation makes the windows look nice when opening and closing, but it takes longer. Finally, some prefer to have the windows hidden and the Hidden button displayed (the small square). These are not the system defaults. Figure 59 on page 87 shows how to change the INI values with REXX code.

```
/* INISTUFF.CMD   REXX code to change some INI stuff      */

application= 'PM_ControlPanel'
keyname=     'PrintScreen'
value=       '0'||'00'x
call SysIni 'USER', application, keyname, value

application= 'PM_ControlPanel'
keyname=     'Animation'
value=       '00000000'x
call SysIni 'USER', application, keyname, value

application= 'PM_ControlPanel'
keyname=     'MinButtonType'
value=       '1'||'00'x
call SysIni 'USER', application, keyname, value

application= 'PM_ControlPanel'
keyname=     'HiddenMinWindows'
value=       '1'||'00'x
call SysIni 'USER', application, keyname, value
```

Figure 59. SysIni to Change System Settings

The largest stumbling block is learning which parameters change which settings. One way to overcome this is to use an INI editor to see the contents of a file. There are many variations available from sources such as bulletin boards. EDTINI and INIMAINT are two of the most popular. Once you have an editor, it is easy to browse through the file to see the different application names, application keys, and values. It may still require some trial and error to identify values associated with settings.

5.7.2 Using SysIni to Read INI Data

Since the INI files contain system information, it is helpful to be able to read that information at times. Each INI file is broken up into a number of sections called applications. Each application is then further broken up into a number of keys. Pieces of data, called key values, are then associated with each of the key names. For example the system colors are stored in an application called PM_Colors. Within PM_Colors there are a number of keys relating to parts of the system we are able to change the color of. If we were to look at the key name IconText we might find the key value ″0 0 0″ which tells OS/2 to make the color of icon text black.

SysIni can be used to retrieve a single key value, all keys for an application, and the names of all applications. We saw in 5.5, “Modifying Workplace Shell Objects” on page 79 that the SysIni function can be used to read all existing object IDs from the user INI file. It does this by reading all key values in the application called PM_Workplace:Location. For the syntax of the SysIni function, see Appendix A, “REXX Syntax Diagrams” on page 225. Figure 60 contains a sample usage of the SysIni function to read the user INI.

```
/* List ObjectIds                                              */
App='PM_Workplace:Location'
call SysIni 'USER', App, 'All:', 'Keys'          1
if Result \= 'ERROR:' then do
   Call SysCls
   Say '';Say '';Say 'Listing ObjectId information';Say '';
   parse value SysTextScreenSize() with row col
   j=row-10
   Do i=1 to Keys.0                                    2
      If trunc(i/j)==i/j Then Do
         Say '';Say 'Press any key to show next screen...'
         key=SysGetKey()
         Call SysCls
         Say '';Say '';Say 'Listing ObjectId information';Say '';
      End
      Say Keys.i                                          3
   End
End
Else Say 'Error querying for' App
Return
```

Figure 60. REXX Program to Display All Installed Object IDs

Notes:

1 'USER' signifies that the user INI file, OS2.INI, is to be read. The application to be read is PM_Workplace:Location. 'All:' signifies that all the keys in the application are to be read. Keys is a stem variable that will be loaded with the INI data.

2 The first element of the stem variable is loaded with the number of key values returned. In this case, it is Keys.0.

3 The remaining stem variable elements contain the key data.

5.8 Extended Attributes

Files contain information that can be read and written using standard REXX I/O functions like Linein, Lineout, Charin, and Charout. If it is a plain text file this data can be viewed and edited. Files can also contain additional information in the extended attributes portion of a file. The extended attributes portion has a structure, and data can be written to, and read from, the extended attributes using the REXXUTIL functions SysGetEA and SysPutEA. The extended attributes headers used by the Workplace Shell are:

- .CLASSINFO
- .ICON
- .LONGNAME
- .TYPE

For an example of the type of data that can be stored in the extended attributes, look at the icon view of the file GEA.CMD that is on the diskette. To look at an icon view of the file, double click on the **Drives** icon on the desktop. Traverse through the drives and directories to where GEA.CMD is stored. The icon view shows an icon similar to the **Start Here** icon. This icon data is stored in the GEA.CMD extended attributes as binary data.

GEA.CMD is a program that can be invoked via drag and drop. It uses SysGetEA to retrieve icon information from the extended attributes of any text file that is dropped on it. It will then start an icon editor session using the icon data. You can then make changes to the icon. When you leave the icon editor, GEA.CMD will load the new icon data back into the file's extended attributes using SysSetIcon. SysSetIcon is a REXXUTIL function that loads icon data into extended attributes. It is an alternative to SysPutEA when loading icon data into the extended attributes of a file. In order to see any changes reflected in the icon, you can go into the settings on the text file to force the Workplace Shell to look at the extended attributes. Figure 61 on page 90 displays the GEA.CMD program.

```
/*    GEA.CMD  - view, change extended attribute icon data.    */
/*                    Uses SysGetEA, SysSetIcon, Icon Editor   */
/*                    Input argument:                          */
/*                                  filename                   */
Arg Fn
/* register REXXUTIL functions                                 */
call RxFuncAdd 'SysLoadFuncs', 'RexxUtil', 'SysLoadFuncs'
call SysLoadFuncs

TmpName = 'TEMP$.ICO'
if (Stream(TmpName,'c','query exists') <> '') then /* temp file exists */
  'ERASE ' TmpName

/* display typeinfo, classinfo, and longname data              */
if SysGetEA(Fn, ".type", "TYPEINFO") = 0 then do  1
  parse var typeinfo 11 type .
  say 'Type of file:'Type
end
if SysGetEA(Fn, ".classinfo", "CLASSINFO") = 0 then do
  parse var classinfo 13 class 23
  say 'Class of file:'Class
end
if SysGetEA(Fn, ".longname", "NAME") = 0 then do
  parse var Name 5 Name
  say 'Name of object:'Name
end

/* load icon data to temp file                                 */
if SysGetEA(Fn, ".icon", "ICON") = 0 then do
  Parse Var ICON AInfo 5 ICON
  Do I = 1 to Length(ICON)
    Rc = Charout(TmpName,Substr(Icon,I,1))        2
  End
  If Length(Icon) \= 0 then Rc = Charout(TmpName)
end
say
```

Figure 61 (Part 1 of 2). GEA.CMD

```
say 'Icon data for file ' fn ' will now be loaded into the icon editor.'
say 'You can make changes to the icon.'
say 'If this is a brand new icon, you will be prompted to save '
say  'the data to a file. Save it to ' TmpName '.'
say 'When you are ready, press the Enter key'
rc=SysGetKey()

/* invoke icon editor                                          */
'@ICONEDIT 'TmpName                                  3
/* write changes to icon to extended attributes               */
Rc = SysSetIcon(fn, TmpName)                        4
'@DEL 'TmpName
return
```

Figure 61 (Part 2 of 2). GEA.CMD

Notes:

1 The first parameter is the file name. type is the name of the extended attribute. TYPEINFO is the variable name that the extended attribute value will be loaded into.

2 Write the icon binary data to a temporary file.

3 Start the Icon Editor.

4 Write icon binary data to extended attributes.

Chapter 6. REXX and C

REXX and the C language are very closely related. In fact, the REXX interpreter is a C Dynamic Link Library (DLL). The REXXUTIL functions are also written in C. As a result, there is a specific protocol that allows REXX programs to call C functions, and C programs to call REXX functions. The ability of REXX programs to call C functions greatly expands the boundaries of what REXX programs can accomplish. Conversely it may be better for C programs to call REXX programs in certain situations to take advantage of the powerful string parsing and Workplace Shell functions available in REXX. For a discussion on how to call C functions from REXX programs, refer to Chapter 3, “External Functions” on page 49. This chapter will discuss in detail how to write C functions that are accessible by REXX, as well as an in-depth look at how to call REXX functions from C programs. An excellent source of information on this topic is the OS/2 2.1 Developer′s Toolkit online REXX Reference.

6.1 Creating C Functions for REXX

When you use RxFuncAdd to register an external function, you are telling the REXX interpreter where the external function is located. For example, in Figure 62 the function SysCls is located in the DLL called REXXUTIL.DLL.

```
Call RxFuncAdd 'SysCls' 'Rexxutil' 'SysCls'
```

Figure 62. Register SysCls

Dynamic link libraries consist of one or more C functions that are available for use by running programs. In order for REXX to use these functions, they must contain statements that use a certain protocol defined for interaction between REXX and C. Since the only data type in REXX is the string, this protocol is based on passing string data between REXX and C. In this section we will discuss how to write DLLs that are accessible by REXX programs. In order to create these DLLs you need a C compiler, and the REXXSAA.H header file which comes with the OS/2 2.1 Developer′s Toolkit.

6.1.1 RXSTRING

RXSTRING is a C data type defined specifically to handle REXX strings. This is the basis for handling parameters passed from REXX to C, and for values returned from C functions to REXX. The C program receives arguments in RXSTRING data types. Depending on the function, the arguments may be converted to another C data type to perform some operation. The return value is converted to data type RXSTRING before being passed back to the REXX program. Figure 63 is the layout of the RXSTRING data type. PRXSTRING is a type defined to be a pointer to RXSTRING.

```
typedef struct  {
      ULONG                   strlength; /* length of string    */
      PCH                       strptr;  /* pointer to string   */
}  RXSTRING;

typedef RXSTRING *PRXSTRING;  /* pointer to an RXSTRING         */
```

Figure 63. RXSTRING

6.1.2 Writing the C Function

There are specific steps that must be taken to make a C function REXX accessible. They are:

1. Include REXX functions with the statement: #define INCL_RXFUNC.
2. Include REXX header file with the statement: #include <rexxsaa.h>.
3. Declare function as a REXX interface with REXXFunctionHandler.
4. Convert RXSTRING parameters to C data types if necessary.
5. Load return string into RXSTRING.

To get started, we will look at a simple C function called QryUserID that was written to provide REXX with access to a User Profile Management Services API. APIs for User Profile Management Services are not directly available to REXX programs. They are available to C. QryUserID allows a REXX program to obtain the user ID of the local logon in UPM. Figure 64 on page 95 shows the C source of QRYRXUSR.C, which contains the QryUserID function code. QryUserID takes no input arguments, and returns a string that contains a user ID.

```
/* QRYRXUSR.C   Query local user ID from Rexx             */

#include <stdio.h>
#include <stdlib.h>
#include <string.h>

#define INCL_32
#include <upm.h>

#define INCL_RXFUNC      1
#include "rexxsaa.h"     2

/* macro to build RXSTRING that is returned from C function*/
#define BUILDRXSTRING(t, s) { \
  strcpy((t)->strptr,(s));\
  (t)->strlength = strlen((s)); \
}
RexxFunctionHandler QryUserID;      3
/*--------------------------------------------------------------------*/
/* Return the logged on local user ID                                 */
/*--------------------------------------------------------------------*/
ULONG QryUserID(                          4
PUCHAR          Name,                  /* Function name             */
ULONG           argc,                  /* Number of arguments       */
RXSTRING        argv[],                /* List of argument strings  */
PSZ             Queuename,             /* Curre queue name          */
PRXSTRING       Retstr)                /* Returned result string    */
 {
  LONG          i;
  UCHAR         str[80];
  UCHAR         usr[UPM_UIDLEN + 1];
  int           rc;
  USHORT        utp;
```

Figure 64 (Part 1 of 2). QryUserID

```
  if (argc>0)                   /* Accept no args            */
    return(40);

  rc = UPMELOCU(usr, &utp);     /* call UPM API              */

  if (rc == UPM_OK)
    strcpy(str, usr);           /* place user ID into var str*/
  else
    strcpy(str,"");
                                /* place user ID in RXSTRING */
  BUILDRXSTRING(Retstr,str);       5

  return(0);
}
```

Figure 64 (Part 2 of 2). QryUserID

Notes:

1 Include REXX functions contained in REXXSAA.H.

2 Include REXXSAA.H.

3 Identify that this function is REXX accessible.

4 Function parameter list must contain these arguments. The actual arguments passed from the REXX program are contained in the array argv[], which is of type RXSTRING.

5 BUILDRXSTRING is a macro defined earlier that takes a string and puts it into the variable of type RXSTRING that will be returned to the calling REXX program.

Figure 65 on page 97 contains a REXX program called GetUser that registers and then invokes the GetUPMID function. Note that external functions can also be called as subroutines, just like internal functions. If an external function is called as a subroutine, then the returned value will be located in the REXX variable RESULT.

```
/* Get local user ID                                    */
/* function name GetUPMID, located in QRYRXUSR.DLL   */
Call RxFuncAdd 'GetUPMID', 'extfunc', 'GetUPMID'

user = GetUPMID()
say 'local user is: ' user
return
```

Figure 65. GETUSER.CMD

6.1.3 Parameter Handling

We have already talked about the use of RXSTRING to pass parameters between REXX and C. Since C has many other data types besides character strings, we need to take a look at manipulating the RXSTRING data in C. C strings require a null character at the end of the string. The REXX interpreter will place the null character at the end of parameter strings sent to C. This is a convenience; however, it brings up a cause for concern. If a REXX string being passed to C has other nulls embedded in it, the C string functions will not work correctly. C assumes the null character means end of string. There are a number of macros defined in REXXSAA.H that are helpful in handling parameters. For example, RXVALIDSTRING returns a true value if there is data in the RXSTRING.

RXSTRING.LIB, a library of functions that comes with the OS/2 2.1 Developer's Toolkit, contains useful functions for converting, copying, and comparing RXSTRING variables. For example, the RXTOI function converts an RXSTRING to an integer. The example in Figure 66 on page 98 shows the function SysCurPos, taken from the REXXUTIL function package. SysCurPos can receive two input parameters, which represent the row and column that the screen cursor should be moved to. If no parameters are received, SysCurPos returns the current row and column position of the cursor. Note that the includes and function handler information are not shown in the figure.

```
/* Function:  SysCurPos - positions cursor in OS/2 session          */

#define  NO_UTIL_ERROR    "0"          /* return string             */
#define  INVALID_ROUTINE 40            /* Raise Rexx error          */
#define  VALID_ROUTINE    0            /* Successful completion     */
#define BUILDRXSTRING(t, s) { \
  strcpy((t)->strptr,(s));\
  (t)->strlength = strlen((s)); \
}
ULONG SysCurPos(CHAR *name, ULONG numargs, RXSTRING args[],
    CHAR *queuename, RXSTRING *retstr)
{
  USHORT start_row;                   /* Row at start              */
  USHORT start_col;                   /* Col at start              */
  LONG   new_row;                     /* Row to change to          */
  LONG   new_col;                     /* Col to change to          */

  BUILDRXSTRING(retstr, NO_UTIL_ERROR);       1
  /* check number of arguments                                     */
  if ((numargs != 0 && numargs != 2))  /* wrong number?            */
    return INVALID_ROUTINE;           /* raise an error            */
```

Figure 66 (Part 1 of 2). SysCurPos

```
  VioGetCurPos(&start_row, &start_col, (HVIO) 0); /* get position    */

  /*convert integers to strings and load into RXSTRING return variable*/
  sprintf(retstr->strptr, "%d %d", (int)start_row, (int)start_col); 2
  retstr->strlength = strlen(retstr->strptr);              3

  /* use rexxsaa.h macro to check validity of input arguments        */
  if (numargs != 0) {       /* reset position to given    */ 4
    if (!RXVALIDSTRING(args[0]) ||     /* not real arguments given?  */
        !RXVALIDSTRING(args[1]))
      return INVALID_ROUTINE;          /* raise an error             */
                                       /* convert row to binary      */
    if (!string2long(args[0].strptr, &new_row) || new_row < 0)
      return INVALID_ROUTINE;          /* return error               */
                                       /* convert row to binary      */
    if (!string2long(args[1].strptr, &new_col) || new_col < 0)
      return INVALID_ROUTINE;          /* return error               */

    /* Set the cursor position, using the input values.             */
    VioSetCurPos((USHORT) new_row, (USHORT) new_col, (HVIO) 0);
  }

  return VALID_ROUTINE;                /* no error on call           */
}
```

Figure 66 (Part 2 of 2). SysCurPos

Notes:

1 BUILDRXSTRING macro is placing the string "0" places in the RXSTRING variable retstr, which is used as the return string to the calling REXX program.

2 In the case where SysCurPos is called with no arguments, load the cursor row and column into the RXSTRING variable retstr, which is used as the return string to the calling REXX program.

3 Load the return string length into the RXSTRING.

4 Use RXVALIDSTRING from REXXSAA.H to check validity of input arguments.

6.2 Creating DLLs Callable by REXX Programs

To create a DLL of C functions you need a C compiler. We used the IBM C Set/2 compiler. Figure 67 contains the commands used to compile and link EXTFUNC.C, which contains three C functions that are available to REXX. They are:

1. The LockPC function uses calls to Presentation Manager (PM) to lock the PC until the correct password is given.
2. The Shutdown function uses calls to PM to perform a system shutdown.
3. The GetUPMID function returns the current local UPM user ID.

Make sure to include any libraries that your functions need in the link step. For example, the GetUPMID function needs access to the UPM.LIB, so it was included in the link step. The compile and link steps can be invoked from the OS/2 command line, or as part of a REXX program. The IBM WorkFrame/2 provides an environment for developing, compiling, and linking C code as well. The result of these compile and link commands is a file called EXTFUNC.DLL, which can be registered by REXX programs.

```
icc /Gd- /Ge- /O- /Ic:\muglib /C+ extfunc.c
link386 extfunc /NOI,extfunc.dll,,c:\muglib\upm.lib,extfunc.def
```

Figure 67. Creating DLL Compile and Link Steps - GENEXT.CMD

The EXTFUNC.DEF is a definition file. This file contains an EXPORT statement, which lists the function names in the DLL. In order for programs to access a function in the DLL, the function name must appear in the EXPORT list. This list is used in the link step during the generation of entry points into the functions. Figure 68 is the definition file for EXTFUNC.

```
LIBRARY extfunc INITINSTANCE TERMINSTANCE

EXPORTS LockPC
              GetUPMID
              Shutdown
```

Figure 68. EXTFUNC.DEF

Figure 69 on page 101 is an example REXX program that registers the functions in the EXTFUNC.DLL and then calls each one. The examples shown in this chapter, as well as the EXTFUNC.DLL and EXTFUNC.C, are on the diskette. Your REXX programs must be able to access a DLL in order to use it. The safest way to do that is to include your DLL in the LIBPATH in the CONFIG.SYS.

```
/* Register and call EXTFUNC functions                          */
call RxFuncAdd 'LockPC' , 'Extfunc' , 'LockPC'
call RxFuncAdd 'Getlocal' , 'Extfunc' , 'GetUPMID'
call RxFuncAdd 'ShutPC' , 'Extfunc' , 'Shutdown'

call LockPC
say  'local logon user is ' Getlocal()
call ShutPC

return
```

Figure 69. Using EXTFUNC.DLL Functions - CALLEXT.CMD

6.3 Calling REXX from C (REXXSTART Function)

There may be instances where C programs may need to call REXX. For example, there may be an application written in REXX that a C program needs to invoke. It may be beneficial to invoke the REXX routine, rather than rewrite in C. Since the REXX interpreter is an OS/2 DLL, for C to call REXX is a very straightforward process. The OS/2 Developer's Toolkit contains information on this topic.

The REXXSTART function from REXXSAA.H invokes the REXX interpreter, allowing C programs to call REXX programs. As discussed earlier in this chapter, REXXSAA.H also has some useful macros. For example, MAKERXSTRING loads string data and string length into an RXSTRING variable. Returned values from REXX programs are accessible by the calling C program. They are in RXSTRING format. Figure 70 on page 102 is an example of a C program REXXDB2.C calling the REXX program GETDB.CMD. GETDB.CMD lists the databases available, and prompts the user to select a database. The database name is returned to the calling program. REXXDB2.C invokes GETDB.CMD, and then prints out the database name selected.

```
/*  File Name:          REXXDB2.C                                    */

#define INCL_REXXSAA
#include <rexxsaa.h>                  /* needed for RexxStart()     */
#include <stdio.h>                    /* needed for printf()        */
#include <string.h>                   /* needed for strlen()        */

int main(void);                       /* main entry point           */

int main()
    {
    RXSTRING arg;                      /* argument string for REXX  */
    RXSTRING rexxretval;               /* return value from REXX    */

    UCHAR   *str = "";                  /* string sent to REXX      */

    APIRET   rc;                       /* return code from REXX     */
    SHORT    rexxrc = 0;               /* return code from function */


    MAKERXSTRING(arg, str, strlen(str));/* create input argument */ 1

    rc=RexxStart((LONG)      0,             /* number of arguments   */
                 (PRXSTRING) &arg,          /* array of arguments    */
                 (PSZ)       "GETDB.CMD",/* name of REXX file        */
                 (PRXSTRING) 0,             /* No INSTORE used       */
                 (PSZ)       0,    /* Default Command environment*/ 2
                 (LONG)      RXSUBROUTINE,  /* Code for how invoked  */
                 (PRXSYSEXIT) 0,            /* No EXITs on this call */
                 (PSHORT)    &rexxrc,       /* Rexx program output   */
                 (PRXSTRING) &rexxretval ); /* Rexx program output   */

    Printf("Database name returned is: %s",rexxretval.strptr); 3
    DosFreeMem(rexxretval.strptr);          /* Release storage  */ 4
                                            /* given to us by REXX.  */
    return 0;
    }
```

Figure 70. Calling REXX From C Example - REXXDB2.C

Notes:

1 MAKERXSTRING from REXXSAA.H loads information into an RXSTRING.

2 Command environment is determined by REXX file extension unless it is overwritten here. In this case the command environment is CMD, since the REXX file is GETDB.CMD.

3 Return string from the REXX program is located in the variable listed in the last parameter of the REXXSTART call.

4 Release RXSTRING storage.

Chapter 7. Multimedia REXX

OS/2 REXX can use the Multimedia Presentation Manager/2 (MMPM/2) capabilities that are shipped with OS/2 2.1 by use of Media Control Interface (MCI) textual string commands.

MCI provides a view of the OS/2 multimedia system that is similar to that of a video and audio home entertainment system. Each component in the system is known as a media device, and can be controlled by a set of textual string-oriented commands.

Media devices can be both internal and external hardware devices. Some devices are compound devices and can open files, such as waveaudio or Musical Instrument Digital Interface (MIDI). MCI can, for instance, be used for opening and positioning files and controlling the related Media Player.

Other devices are controllable external devices that can be controlled directly, such as a Compact Disk - Read Only Memory (CD-ROM) device or videodisc. You can for instance open or close the tray in a CD-ROM device or play the Compact Disk (CD). The devices suported by MMPM/2 include:

Videotape A videotape player or recorder.

Videodisc A videodisc player.

CDaudio A CD-ROM device which supports standard compact disc playback.

Waveaudio A device which supports digital audio files (″sound board″).

Sequencer A device which supports MIDI files.

Digitalvideo A device which supports audio/video files, either hardware-assisted or software motion video-only.

Note

Digitalvideo, image devices or clipboard cut-and-paste operations require a PM environment to function. Therefore, they cannot be executed from a command file unless the file is run through either PMREXX, using ′START /PM CMD.EXE cmdfile′ or a program created with one of the visual builders like the example program RXPLAY.EXE on the samples diskette.

7.1 MMPM/2 Installation

Although MMPM/2 is shipped with OS/2 2.1, it is not automatically installed during OS/2 installation. It must be installed using the command MINSTALL from the first MMPM/2 diskette.

7.2 Using MCI from REXX

This chapter intends to demonstrate some important issues in building a MMPM/2 REXX application. The samples diskette also contains a sample application:RXPLAY.EXE, which can be used to explore the use of MMPM/2 REXX commands.

7.2.1 Registering MMPM/2 Functions

The MCI REXX functions are supplied in the MCIAPI.DLL external function package. External functions are described in detail in Chapter 3, “External Functions” on page 49. In order to use the REXX MCI interface commands you must first register them with the following code:

```
rc = RxFuncAdd('mciRxInit','MCIAPI','mciRxInit')
Call mciRxInit
```

Figure 71. Registering REXX MMPM/2

After registration the commands are useable within your system until a shutdown is performed or the function mciRxExit is performed. In addition to mciRxInit and mciRxExit there are two functions you need for building MMPM/2 applications in REXX:

mciRxSendString For sending the MCI command strings.

mciRxGetErrorString For getting error information on a return code from mciRxSendString.

7.2.2 Checking if MMPM/2 is Installed

In order to use MCI, MMPM/2 must be installed on your machine. The following code checks the presence of the OS/2 MMBASE environment variable which is set by MMPM/2 installation and contains the path that MMPM/2 is installed in:

```
MMpath = value( "MMBASE",, "OS2ENVIRONMENT" )
If MMpath = '' Then Do
  Say 'Cannot locate MMPM/2 on the system!'
  Exit
End
Else MMpath = Strip(MMdrive,,';')
```

Figure 72. Checking if MMPM/2 is Installed

7.2.3 Opening a Media Device

To use a media device it must first be opened. This can be done in REXX either by opening a file or opening a media device. If a file is opened then the Media Control Interface can use the extended attributes or file extensions associated with the file to select the controlling device to like in the following example:

```
Rc = mciRxSendString('open c:\vid\a.avi alias player wait','Ret','0','0')
```

Figure 73. Opening a File and a Media Device

Note

The last two parameters in the mciRxSendString command are reserved for future use for notifying window handle and user parameter and must always be set to 0.

An alias name can also be provided for further reference to the device, and it is good practice to do so as in Figure 74. However if an alias name is not provided then open will return a device id for the device on a succesful open.

```
Rc = mciRxSendString('open digitalvideo alias player wait','Ret','0','0')
```

Figure 74. Opening a Media Device

Note

It is possible to open a device as shareable, but since a REXX command file cannot receive notification on when it is gaining or losing access on a MCI device, the shareable option should not be used.

7.2.4 Error Checking

A mciRxSendString call always returns 0 on a successful execution. If the return code is not 0 then the **mciRxGetErrorString** function can be used to determine the cause of the error:

```
Rt = mciRxSendString('open c:\vid\a.avi alias player wait','Ret','0','0')
If Rt <> 0 Then Do
  ErrRc = mciRxGetErrorString(Rt,ErrorString)
  Say 'Unsuccessful execution!'
  Say 'Error code =' Rt
  Say 'Error info =' ErrorString
End
```

Figure 75. Error Checking in MMPM/2 REXX

7.2.5 MCI Commands

After a device is opened there are several commands that can be executed to control the device. The commands are:

ACQUIRE Gains the use of physical recources associated with a device.

CAPABILITY Requests information about a particular capability of a device.

CLOSE Closes a device.

CONNECTOR Enables, disables or queries the status of connectors, for example line in connector, on a device.

INFO Returns product information on a device.

LOAD Loads a file into a previously opened device.

OPEN Opens a device.

PAUSE Pauses a device.

PLAY Begins playing a previously loaded file on a device.

RECORD Starts recording data.

RELEASE Releases resources that were previously acquired.

RESUME Resumes playing from a paused state.

SAVE Saves data for a device.

SEEK Moves to a specified position in a file and stops.

SET Changes settings for a device.

STATUS Queries the status of a device.

STOP Stops a device.

All commands are executed through mciRxSendString either by using:

```
Call mciRxSendString('command object parms','RetVar','0','0')
```

or using:

```
Rc = mciRxSendString('command object parms','RetVar','0','0')
```

Those commands that return string information through the use of the return variable (**RetVar** in the previous example) must always use a **Wait** keyword after the command parameters. The following chapters describe the usage of these commands in more detail. See the online Multimedia REXX Reference which is located in the Multimedia folder on the default desktop for a complete description of the commands.

7.2.6 ACQUIRE

```
Rc = mciRxSendString('ACQUIRE object items WAIT','RetVar','0','0')
```

Gains the use of the physical resources associated with the object. The following optional items modify the basic command:

Exclusive Obtains exclusive use of the resources associated with the object. No other applications can obtain use of the device until the **RELEASE** command is executed. Has no affect unless the device is opened as shareable.

Exclusive instance Obtains exclusive use of only the resources needed by the object. Other applications may be able to use the same device concurrently, depending upon the number and types of resources they require.

Queue Requests use of the resources associated with the object. If another application has exclusive use of the same physical resources, then this command will wait until the resources can be acquired.

7.2.7 CAPABILITY

```
Rc = mciRxSendString('CAPABILITY object items WAIT','RetVar','0','0')
```

Requests information about a particular capability of a device. The WAIT keyword must be specified in order to receive the returned string information. Following is an example of using CAPAPILITY to check if the digitalvideo device can record:

```
Rc = mciRxSendString('open digitalvideo alias player wait','Ret','0','0')
Rc = mciRxSendString('CAPABILITY player can record WAIT','RetV','0','0')
If RetV = 'TRUE' then
  Say 'Device can record.'
Else
  Say 'Device cannot record.'
```

7.2.8 CLOSE object

Closes the device context and frees resources. The object to close should be the same one used when the device context was initially opened. for example:

```
Rc = mciRxSendString('open digitalvideo alias player wait','Ret','0','0')
Rc = mciRxSendString('CAPABILITY player can record WAIT','RetV','0','0')
If RetV = 'TRUE' then
  Say 'Device can record.'
Else Do
  Say 'Device cannot record.'
  Rc = mciRxSendString('close player','Ret','0','0')
End
```

7.2.9 CONNECTOR

```
Rc = mciRxSendString('CONNECTOR object action items WAIT','RetV','0','0')
```

Enables, disables or queries the status of connectors, for example line in connector, on a device. **Object** is any MCI object described in Chapter 7, "Multimedia REXX" on page 105. **Action** is one of the following:

- Enable
- Disable
- Query

Items is one of the following:

- Number <connector number>
- Type <connector type>
- Both of the previous

The following example queries the wave stream capability of an alias:

```
Rc = mciRxSendString('CONNECTOR player query type wave stream WAIT',,
                     'RetV','0','0')
If RetV = 'TRUE' Then
  Say 'Wave stream exists.'
Else
  Say 'Wave stream does not exist.'
```

Figure 76. Usage of Connector to Query Wave Stream Capability

7.2.10 INFO

Returns product information for a particular device. For example:

```
Rc = mciRxSendString('INFO Digitalvido product WAIT','RetV','0','0')
If RetV = 'Software Motion Video' Then
  Say "It's part of OS/2!"
```

7.2.11 Load

Loads a file into a previously opened device. If a device is not already open you can use **Open** to load the file like in the following example:

```
If Alias <> '' then
  Rcc=mciRxSendString('load Player C:\MC.AVI wait', 'RetV','0','0')
Else
  Rcc=mciRxSendString('open C:\MC.AVI alias Player wait','RetV','0','0')
If Rcc <> 0 then Do
  MacRC = mciRxGetErrorString(Rcc, 'ErrStVar')
  Say 'File load failed.'
  Say 'Error code:'Rcc
  Say 'Error message:'ErrStVar
  Return
End
```

Figure 77. Load a File to a Device

7.2.12 PAUSE

Stops playing a file. The difference between PAUSE and STOP is device dependent. On video devices, PAUSE generally continues to display the last frame, whereas STOP causes the display to blank. A device that is paused can frequently begin playing again with less latency than if it were stopped. **RESUME** or **PLAY** can be used to continue playing. Example in Figure 78 demonstrates use of PAUSE by opening a video file,starting to play for 2 seconds and pausing to wait for input:

```
rc = RxFuncAdd('mciRxInit','MCIAPI','mciRxInit')
Call mciRxInit
rc = RxFuncAdd('SysSleep','RexxUtil','SysSleep')

Rc=mciRxSendString('open digitalvideo alias player wait','RV','0','0')
Call mciRxSendString 'load player D:\MM\FISHB15.AVI wait','RV','0','0'
Call mciRxSendString 'play player', 'RetStr', '0', '0'
Call SysSleep 2
Call mciRxSendString 'pause player', 'RetStr', '0', '0'
Say 'Continue Y/N?'
Pull Ans
Ans = Translate(Ans,'Y','y')
If Ans = 'Y' then
  Call mciRxSendString 'resume player', 'RetStr', '0', '0'
Say 'Press enter to terminate.'
Pull
Call mciRxSendString 'close player', 'RetStr', '0', '0'
 Rc = mciRxExit
```

Figure 78. PAUSPLAY.CMD, Play a Video From a REXX .CMD File

7.2.13 PLAY

Plays a file loaded to a device. The starting position can be defined using the **from** parameter. The ending position can be defined using the **to** parameter. For example:

```
Rc = mciRxSendString('play player from 1 to 13000','RetV','0','0')
```

7.2.14 RECORD

Starts recording data. By default, recording does not overwrite existing data but rather inserts data at the current position. On devices (such as audio or video tape) that cannot support inserting data, recording overwrites existing data by default. The following items modify the basic command:

- insert
- overwrite
- from pos
- to pos

Example for recording data:

```
Rc = mciRxSendString('record player insert','RetV','0','0')
```

7.2.15 RELEASE

Releases previously acquired resources. For example:

```
Rc = mciRxSendString('acquire player exclusive wait','RetV','0','0')
Rc = mciRxSendString('play player wait','RetV','0','0')
Rc = mciRxSendString('release player return resource','RetV','0','0')
```

7.2.16 RESUME

Resumes playing or recording from a previously paused state. For example:

```
Call mciRxSendString 'pause player', 'RetStr', '0', '0'
Say 'Continue Y/N?'
Pull Ans
Ans = Translate(Ans,'Y','y')
If Ans = 'Y' then
  Call mciRxSendString 'resume player', 'RetStr', '0', '0'
```

7.2.17 SAVE

Saves data for a device. The destination path and name of the file to be saved can be specified. If a filename was specified during the LOAD command, then you do not need to specify a filename when saving the file; the data will be saved using the current filename. If an untitled file was loaded, then a filename must be specified, or an error will be returned. For example:

```
Call mciRxSendString 'save player', 'RetStr', '0', '0'
```

7.2.18 SEEK

Finds a specified position and stops. Can be given the parameter **end**, **start** or a position. For example:

```
rc = RxFuncAdd('mciRxInit','MCIAPI','mciRxInit')
Call mciRxInit
Rc=mciRxSendString('open digitalvideo alias player wait','RV','0','0')
Call mciRxSendString 'load player D:\MM\FISHB15.AVI wait','RV','0','0'
Do Forever
  Call mciRxSendString 'play player wait', 'RetStr', '0', '0'
  Call mciRxSendString 'seek player to start', 'RetStr', '0', '0'
  Say 'Continue Y/N?'
  Pull Ans
  Ans = Translate(Ans,'Y','y')
  If Ans <> 'Y' then Leave
End
Call mciRxSendString 'close player', 'RetStr', '0', '0'
Rc = mciRxExit
```

Figure 79. Play a Video Continuously From a REXX .CMD File

7.2.19 SET

Establishes the desired settings for a device. For example set the speed format to frames per second:

```
Call mciRxSendString 'set player speed format fps wait','Rv', '0', '0'
```

7.2.20 STATUS

Obtains status information for a device. For example get the length of a previously loaded file:

```
Call mciRxSendString 'status player length wait','Len', '0', '0'
Say 'Length of file is:'Len
```

7.2.21 STOP

Stops a device.

7.3 RXPLAY.EXE

As digitalvideo and image devices require a PM environment, a logical environment for building multimedia applications for those devices with REXX is a visual builder.

Chapter 10, “Visual REXX Builders” on page 177 describes two of the most commonly used visual builders. We used VisPro/REXX 1.1 to build a small application, RXPLAY.EXE to demonstrate the use of REXX MCI commands. RXPLAY.EXE is included in the samples diskette. The source is also included in the 4199VP folder on the samples diskette. To start using RXPLAY.EXE, go to the drive and directory it exists on and type RXPLAY. You will get a window like in Figure 80 on page 116. The window contains a menubar with **File**, **Device open**, **Device close** and **Help** items. There is also a **REXX syntax** list box, which at startup shows the syntax for registering the REXX MCI functions. All actions that are performed with the player will provide the REXX syntax for that particular action in the REXX syntax list box. There are also five pushbuttons:

- **< <** for rewinding the player.
- **Pause** for pausing the player.
- **Play** for playing the player.
- **Help** to display the MMPM/2 REXX online reference.
- **Copy to** for copying the code from the REXX syntax list box to a file.

Included is also a slider to set the position of the file in the player and vertical and horizontal slider to position the media player on screen. On the bottom there are three drop down list boxes:

- **Commands** for performing commands to the player.
- **Status** for displaying status information for the player.
- **Capability** for displaying capability information for the player.

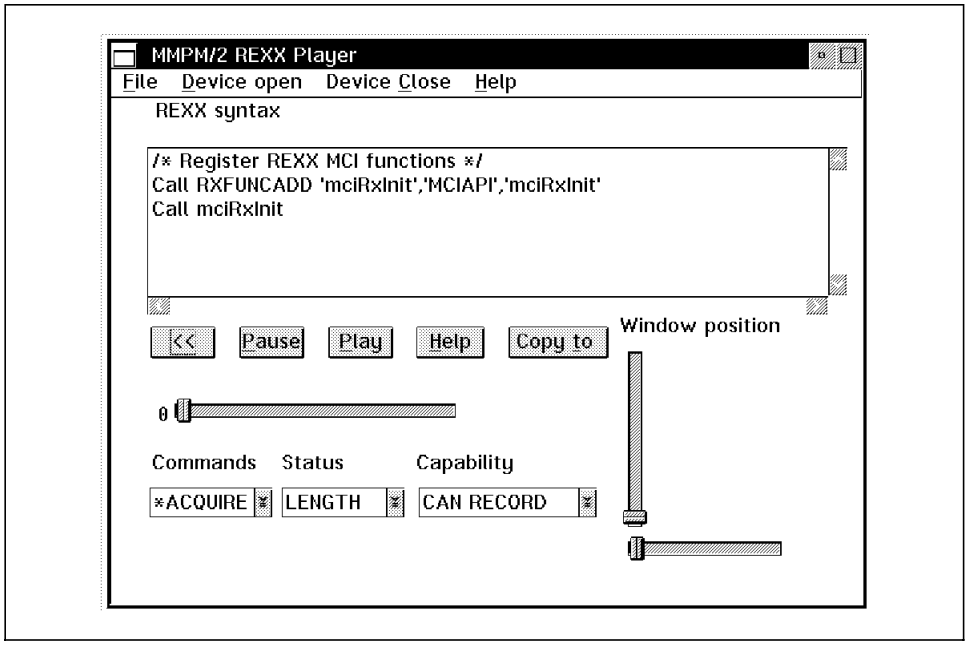


Figure 80. RXPLAY.EXE MMPM/2 REXX Player

To start testing the REXX multimedia capabilities first open a device by selecting **Device open** and then select a device from the list of devices like in Figure 81. The program now tries to open the selected device. If the device cannot be opened a message box will appear giving information why the device could not be opened and the corresponding REXX code will appear in the REXX syntax window.

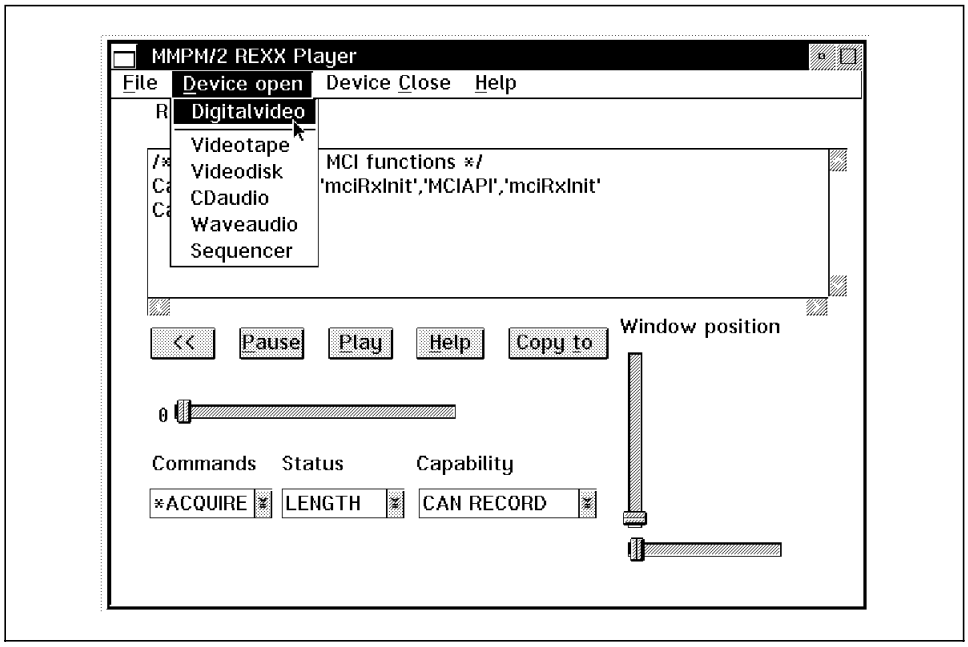


Figure 81. MMPM/2 REXX Player Device Open

Then select **File** and **Open** to open a file. A file can also be opened without first opening a device, which would give a different REXX code. You now get a standard file open dialog with the directory set to MMOS2MOVIES directory on your MMPM/2 installation drive like in Figure 82. After selecting a file, wait until a figure describing the length of the file being played appears on the right side of the slider.

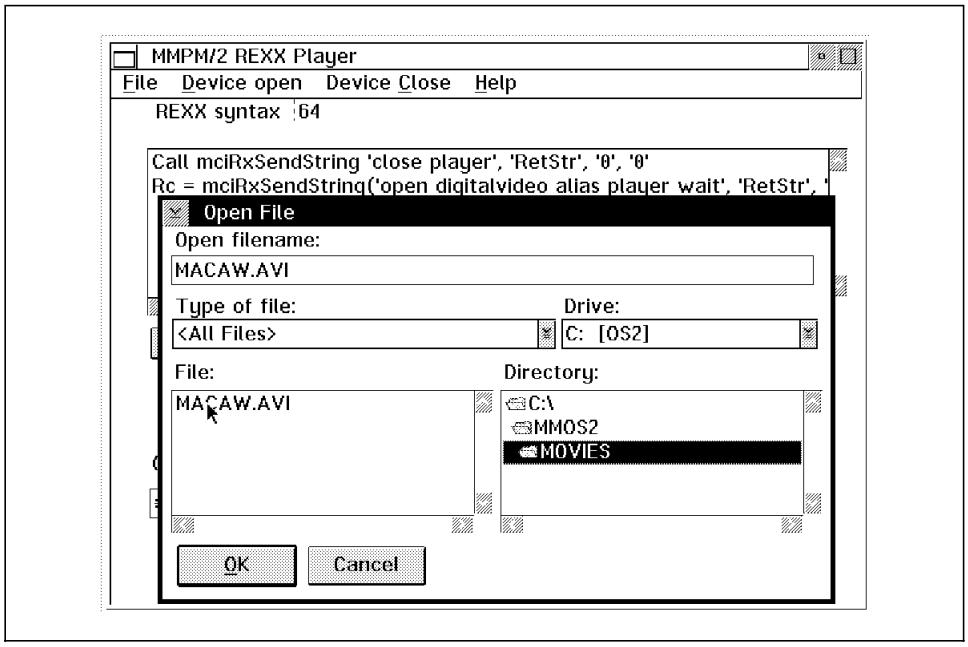


Figure 82. MMPM/2 REXX Player File Open

Now you can start demonstrating with the pushbuttons, sliders and drop down boxes. For instance if you have selected an .AVI file and press the **Play** pushbutton the media player is started. The **Window position** sliders can be used to position the media player on the screen like in Figure 83.

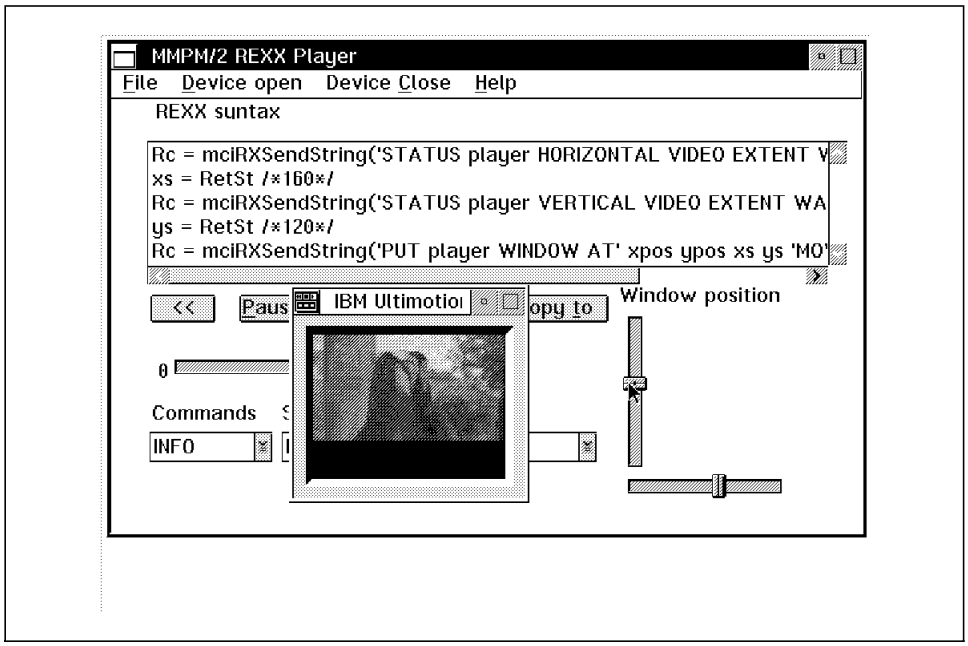


Figure 83. MMPM/2 REXX Player Position Media Player

Use the **Commands** drop down list box to demonstrate with additional commands like in Figure 84.

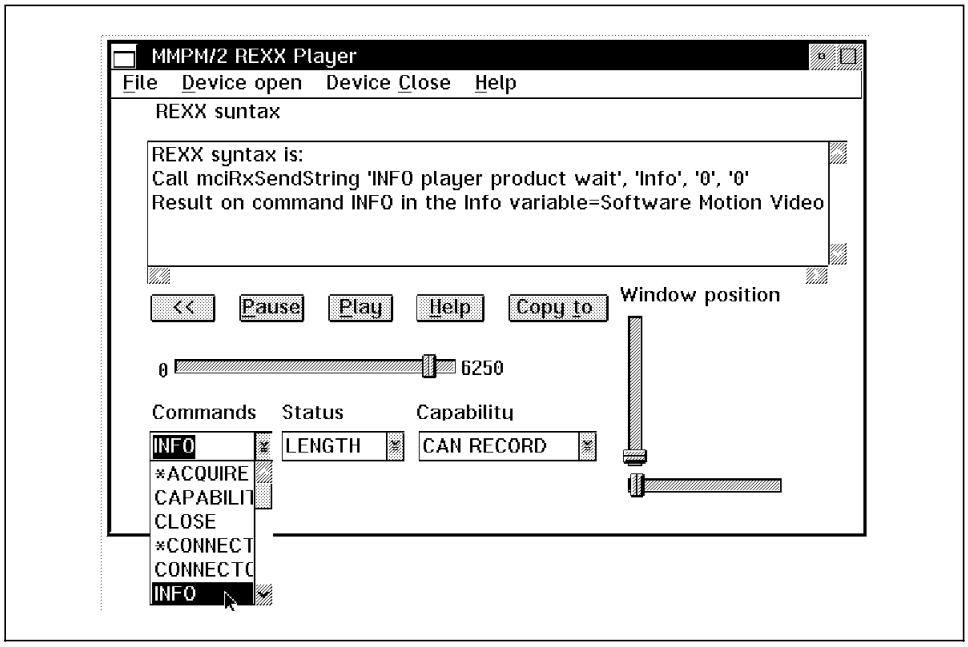


Figure 84. MMPM/2 REXX Player Command List

Note

Some commands have * in front of them. Those commands do not execute, they only provide the REXX syntax.

The **Status** list box provides status information common to all MCI devices like in Figure 85 Some information refers to the file being played, like length and position.

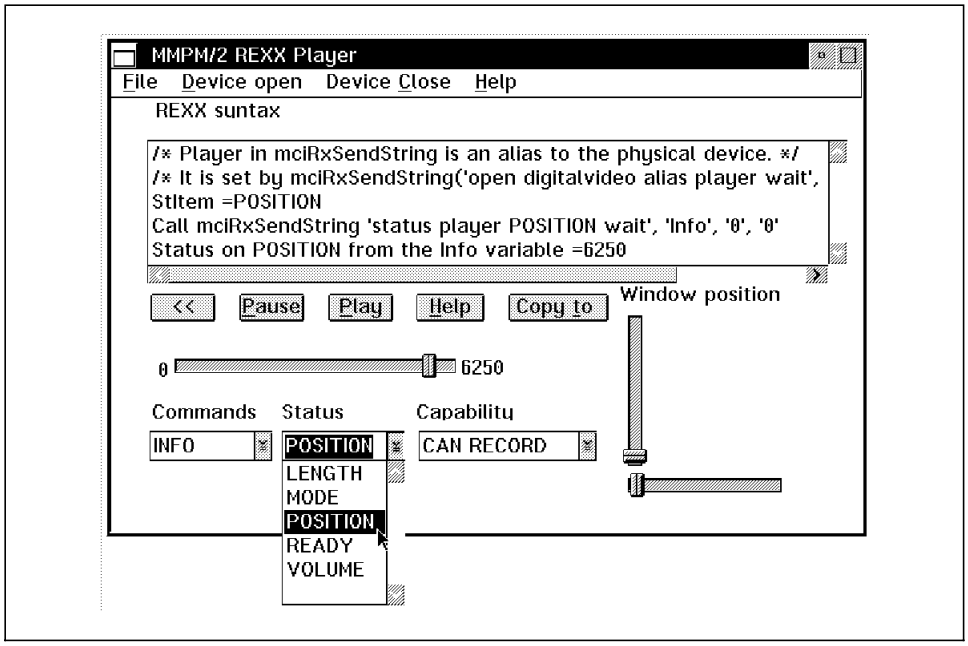


Figure 85. MMPM/2 REXX Player Status List

The **Capability** list box provides capability information regarding the selected device like in Figure 86.

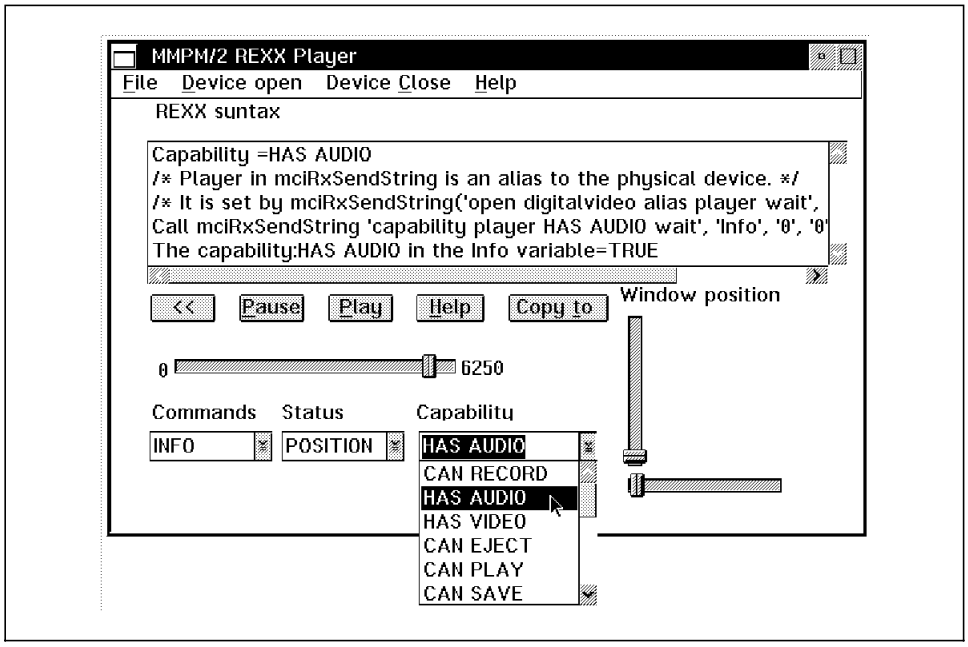


Figure 86. MMPM/2 REXX Player Status List

The code information in the REXX syntax list box can at any time be saved to a file by using the **Save to** pushbutton. This always appends to a file and does not overwrite so you can use it to gather the rexx code you want into a file. Using the **Save as** pushbutton gives you the **Save as** dialog, as inFigure 87

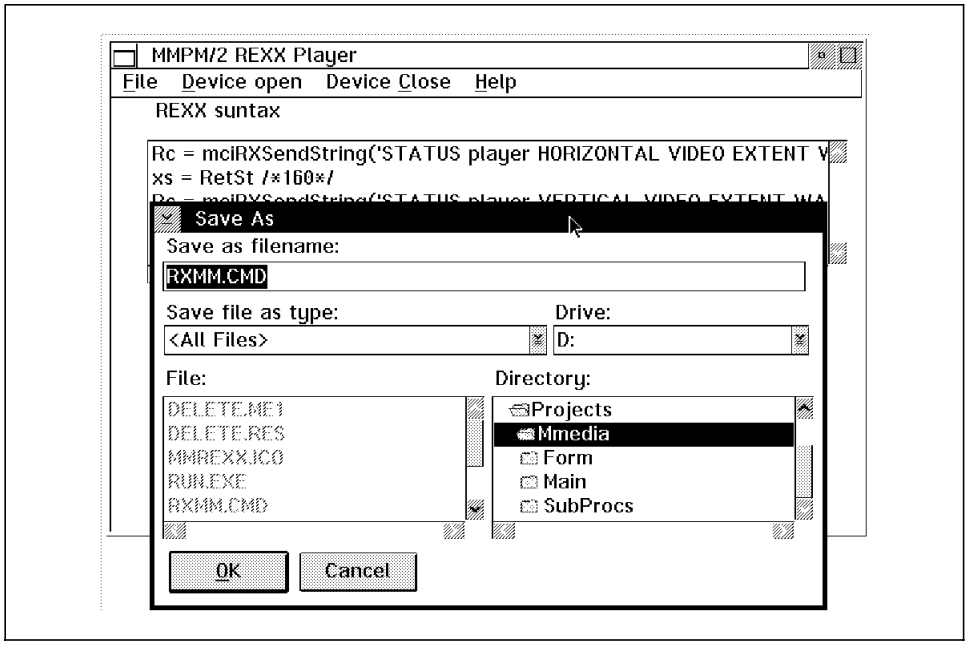


Figure 87. MMPM/2 REXX Player Save to File

Note

Also the informational messages from the REXX syntax window are copied to the file.

Choosing **Help - Information** from the menubar will give you a small message box on information about the MMPM/2 REXX Player like in Figure 88.

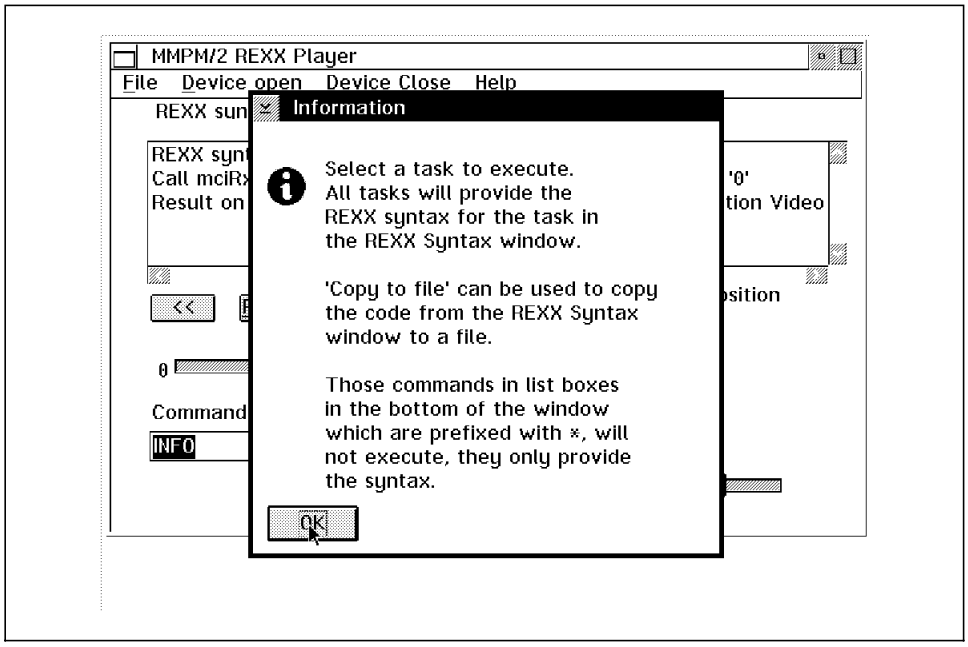


Figure 88. MMPM/2 REXX Player Information

Chapter 8. REXX Interfaces to CM/2 EHLLAPI

Communications Manager/2 (CM/2) is an extension of OS/2 that provides the capability of connecting a workstation to host machines. The physical connection to the host machine can be made by cable to a local host, or by networks to local and remote host machines. CM/2 provides a way for a workstation user to communicate with a host by emulating a 3270 or 5250 host session. In other words your workstation can act as a 3270 or 5250 display terminal. CM/2′s emulator provides a PM window that displays a host session and allows keyboard entry to the host session.

CM/2 also provides APIs to the emulator that make it possible to write programs that can manipulate host sessions. Through these APIs programs can read the host screen data, send keystrokes to the host session, send and receive files, and make changes to the PM presentation space. These APIs are contained in a package called Emulator High-Level Language Application Programming Interface (EHLLAPI). EHLLAPI can provide an interface to 3270 and 5250 sessions. This chapter will focus on REXX EHLLAPI APIs and their interaction with 3270 sessions. For more information on EHLLAPI, refer to *IBM Communications Manager/2 Version 1.0 EHLLAPI Programming Reference*.

8.1 EHLLAPI Uses

The concept of automating a host session is an interesting one, and it can be a very useful one. Here are some of the things REXX EHLLAPI programs can accomplish:

- Automate repetitive tasks.
- Mask complete applications from the user.
- Consolidate several complicated tasks into one simple task.
- Simplify existing host applications.
- Monitor response time and availability.
- Monitor events that are diverse in nature.
- Automate console operations.

One of the more common uses of EHLLAPI is to automate the use of host legacy systems for repetitive tasks. A REXX EHLLAPI program can eliminate the time consuming task of users scrolling through screen after screen on one or more host systems to perform data entry or data retrieval. The REXX

EHLLAPI program can perform these tasks automatically. In order to write a program to automate this type of task, we need to be able to do the following:

- Connect to a host session
- Read data from the host session
  - To identify which screen is currently displayed
  - To save data vital to the task that is being automated
- Send keystrokes to the host
  - To advance to the next screen
  - To write data to the screen entry fields
- Determine host availability
  - To know when your keystrokes have been accepted
- Disconnect from a host session.

8.2 Calling EHLLAPI Functions from REXX Programs

All EHLLAPI functions are invoked from REXX through the same function call. This one function, called HLLAPISRV, must be registered, since it is an external function. The first parameter of the call determines which function will be invoked. The HLLAPISRV function is located in SAAHLAPI.DLL. The subdirectory where SAAHLAPI.DLL resides on your system must appear in the LIBPATH statement in CONFIG.SYS in order use EHLLAPI. Figure 89 on page 127 shows how to register the EHLLAPI function, and then uses the HLLAPISRV function to connect to a host presentation space window. Note that the function is registered as HLLAPI, so that is what is used in the program as the function name. It is a common practice to call the EHLLAPI function HLLAPI, although it is not required. Appendix D, “CM/2 REXX EHLLAPI Reference” on page 295 contains the function names that can be used as the first parameter in HLLAPI calls.

```
/* load hllapi functions */
if Rxfuncquery('hllapi') then
  call Rxfuncadd 'HLLAPI','SAAHLAPI','HLLAPISRV'

rc= hllapi('Connect_PM','B')    /* connect to B host session window*/
```

Figure 89. Register HLLAPISRV and Connect to Presentation Space Window

8.3 Connecting and Disconnecting Host Sessions

To communicate with a host session, an application must be connected using one or more of the following functions:

- **Connect** - Allows sending or monitoring host presentation space activities.
- **Connect_PM** - Provides for Presentation Manager window manipulation and monitoring.

```
rc= hllapi('Connect','A')           /* connect to host session A */
```

Figure 90. Connect to Host Session

It is important to disconnect your application from the host session before the application terminates. This frees the host session to be connected to other applications. To disconnect an application from a host session, one or more of the following functions must be used, depending on how the application was originally connected:

- **Disconnect** - Disconnect from host session.
- **Disconnect_PM** - Disconnect from Presentation Manager window.

```
rc= hllapi('Disconnect')     /* disconnect from host session  */
```

Figure 91. Disconnect from Host Session

8.4 Reading the Host Screen

Reading host screen data is useful for a couple of reasons. First of all, for example, as your program traverses through the different screens of a host application, reading the host screen data allows your program to determine which screen is currently being displayed. Second, it may be required for your program to capture data from a host screen into REXX variables for some type of manipulation.

There are a couple of EHLLAPI functions that provide a way to read the host session screen into a variable:

- **Copy_PS** - returns a string containing the entire contents of the presentation space.
- **Copy_PS_To_Str** - returns a string containing a specified portion of the contents of the presentation space.
- **Copy_OIA** - returns a string containing the contents of the Operator Information Area (OIA).
- **Copy_Field_To_String** - transfers characters from a target field into a data string.

Copy_OIA is useful for determining if the host is input inhibited, for example. Copy_Field_To_String can be used on host screens that have defined fields. We will focus on Copy_PS and Copy_PS_To_Str in this section.

Since Copy_PS and Copy_PS_To_Str return a string representing the presentation space, this string is more useful if we know the dimensions of the presentation space. Since the presentation space in CM/2 can differ in size, we need a way to determine the current number of rows and columns. The Query_Session_Status EHLLAPI function provides a way to get this information.

8.4.1 How to Obtain the Presentation Space Dimensions

The Query_Session_Status function returns a string of data pertaining to the session requested. Part of this data is a binary representation of the number of rows and the number of columns in the presentation space. Using built-in REXX functions we can convert this binary data into decimal. Refer to Figure 92 on page 129 for the code to accomplish this. We will see how this information is used in conjunction with the Copy_PS_To_String function in 8.4.2, “Copying the Presentation Space” on page 129.

```
rc=hllapi('Query_Session_Status', 'A')  /* query A session */

parse var rc 12 bin_row 14 bin_col 16  /* get binary row, ol values */

row_num = c2d(reverse(bin_row))  /* convert to decimal */
col_num = c2d(reverse(bin_col))
say  row_num ' rows '
say col_num ' columns'
```

Figure 92. Obtain Presentation Space Row and Column Values

8.4.2 Copying the Presentation Space

Copy_PS_To_Str returns a portion of the presentation space as a string. It takes a starting position and a length as input parameters. If you know the dimensions of the presentation space and the portion of the host screen you wish to copy, this function is very useful. For example, to copy the last row of the host screen into a variable called last_row, you could invoke the code shown in Figure 93.

```
rc=hllapi('Query_Session_Status', session)
parse var rc 12 bin_row 14 bin_col 16     /*         get binary    */
row_num = c2d(reverse(bin_row))           /*    convert to decimal */
col_num = c2d(reverse(bin_col))
ps_num = row_num * col_num              /*total presentation space */
last_row_pos = ps_num - col_num + 1       /* beginning of last row */

rdrfile= hllapi('Copy_PS_to_Str',last_row_pos,col_num) /* copy row */
say last_row
```

Figure 93. Copy Last Row of Host Screen

Note - Presentation Space Positions

The presentation space is a grid made up of a specified number of rows and columns. The positions of the characters in the presentation space are numbered sequentially, beginning with position 1 in the upper left-hand corner of the presentation space, and continuing to the last position which is in the bottom right-hand corner. The positions are numbered sequentially going left to right across a row, and then wrap to the next row. For example, in a 24x80 presentation space:

- Position 1 is the first column of row 1.
- Position 80 is the last column of row 1.
- Position 81 is the first column of row 2.
- Position 1920 is the last column of row 24.

Copy_PS copies the entire presentation space to a string. If we know the dimensions of the presentation space we can create an array that represents the current host screen.

8.4.3 Searching the Presentation Space

There are times when you need to identify which host screen is currently in the presentation space. As your program is traversing through a host application's screens, for example, it is a good idea to look for a unique string value that represents a certain host screen. This can be done by using Copy_PS to copy the presentation space to a string, and then using a REXX built-in function to search for the unique string. A more straightforward method is to use the EHLLAPI Search_PS function. Search_PS can search the entire presentation space, or search a section of it starting at a specified position. The returned value is the position of the start of the string. The returned value is 0 if the string is not found. Search_PS searches the currently connected host session. For example, to search a 24x80 presentation space for the string "VM READ" in the last row:

```
pos= hllapi('Search_PS','VM READ',1841)  /* start in last row */

if (pos > 0) then                        /*     found VMREAD  */
  say 'Found VM READ in position ' pos
```

Figure 94. Search Presentation Space

8.5 Sending Keystrokes to the Host Session

Using EHLLAPI you can simulate keyboard entry to host sessions. This is useful for data entry, clearing screens, issuing commands, using PF keys and the Enter key, and any other keystroke that you could manually send to a host session. Special keys on the keypad are represented with specific character strings. For example, the Enter key is represented by the string '@E'. A complete list of keyboard mnemonics is provided in Appendix D, "CM/2 REXX EHLLAPI Reference" on page 295. The EHLLAPI function to send keystrokes to a host session is Sendkey. Sendkey takes one parameter, which is the string of keystrokes that are to be sent to the currently connected host session. Refer to Figure 95 for examples of the usage of the Sendkey function.

```
/* enter Rdrlist command */
 rc= hllapi('Sendkey','Rdrlist'||'@E')

/* clear screen          */
 rc= hllapi('Sendkey','@C')
```

Figure 95. Sendkey Function

The Copy_Str_To_PS function will copy a data string to the presentation space. The string is copied starting in the position specified on the call. This can be useful for data entry. It executes faster than Sendkey in most situations. It does not support the keyboard mnemonics listed in Appendix D, "CM/2 REXX EHLLAPI Reference" on page 295.

8.6 Determining Host Availability

In the example in Figure 95, how do we know if the Rdrlist command completed and the host is available before we issue the clear screen keystroke? Do we need to know? Depending on the situation, you probably do need to know. Consider this example. Suppose after issuing the Rdrlist command, you wanted your program to copy the presentation space to determine how many reader files there are. If the timing isn't right for whatever reason, the host may still be processing the Rdrlist command when your program issued the Copy_PS_To_Str function call. The resulting string created by Copy_PS_To_Str will not contain what you want it to. It will copy the current screen, which just contains the echo of the Rdrlist command. The host hasn't finished processing yet. In many situations the REXX EHLLAPI program will be faster than your host session.

```
rc= hllapi('Sendkey','Rdrlist'||'@E')

/* copy first row of rdrlist screen - contains number of reader files */
rdrfile= hllapi('Copy_PS_to_Str',1,80)
```

Figure 96. Invoking Rdrlist with No Host Checking

EHLLAPI provides function calls that allow you to determine if the host has completed a command and is available for more input. Depending on the environment, there are different combinations of these calls that can give more reliable feedback as to whether or not the host has completed processing keystrokes. In addition, your program must be coded so that it will wait for the host command to process before continuing. This area is probably the single most challenging aspect of EHLLAPI programming. Often programmers develop their own functions that are a combination of the EHLLAPI functions to achieve the desired results. Here is a list of EHLLAPI functions that can help in determining host availiability:

- Copy_OIA
- Pause
- Query_Host_Update
- Start_Host_Notify
- Stop_Host_Notify
- Wait

8.6.1 Using Screen Changes to Manage Host Availability

One of the safest techniques when determining if the host has completed processing of a command is to use Search_PS to look for a string on the screen that would not be there before the command is processed. It is a string that would only appear on the updated screen. If Search_PS finds the string, then the command has been processed. The Search_PS must be in a loop of some kind, since we need to wait for the host to complete the command before the program continues. For example, if the Rdrlist command is issued on the host, one of two things will happen as a result. Either there are no reader files, and the string "No files in your reader" is displayed on the screen, or there are reader files and the Rdrlist screen is displayed. We can expand on Figure 96 to wait for one of these two possibilities to happen. Take a look at Figure 97 on page 133.

```
 rc= hllapi('Sendkey','Rdrlist'||'@E')

do until (rc1>0 | rc2>0)
  /* search for unique string found in rdrlist screen */
   rc1=hllapi('Search_PS','Filename Filetype Class',81)

  /* search for no rdr files string                   */
   rc2=hllapi('Search_PS','No files in your reader',1)
end /* Do loop */

if rc1>0 then
  /* copy first row of rdrlist screen                 */
  rdrfile= hllapi('Copy_PS_to_Str',1,80)
else
  say ' No reader files '
```

Figure 97. Invoking Rdrlist with Host Checking

This method is very slow. The continuous calls to Search_PS slows down the host as it processes the Rdrlist command. Also, this solution will not leave the loop until the Rdrlist command has completed processing and one of the resulting strings has been displayed in the presentation space. The question that comes up now is what happens if neither one of those strings is displayed. For example, what if there was a message on the screen that holds the screen until a clear screen key is issued. What if the host is extremely slow? Would you want to wait for the host, or exit the program and try later? What if the host has dropped the connection? Your loop will turn into a forever loop.

There are bits in the system that determine if the host is busy, and when the host has been updated. There are EHLLAPI functions that check these bits. These functions can help you determine host availability, and determine when commands issued to the host have been processed. We have found that these functions may not always give you the desired results. It is best to devise a combination of these function calls that works in your environment. For example we have found that the OIA bits are not always reliable in determining if the host is available, since the OIA busy clock flickers sometimes and the bits turn on and off a number of times before the command has actually processed and the host is available.

8.6.2 Query Host Update Function

The Query_Host_Update function determines if the OIA or presentation space for the session has been updated. In order to use the Query_Host_Update function, a Start_Host_Notify function call must have previously been issued. The Start_Host_Notify function watches to see if the host presentation space or OIA has been updated. Query_Host_Update returns a value of 0 if no updates were made since the last call. It returns a value of 22 if the presentation space has been updated since the last call. Figure 98 is an example of the usage of the Query_Host_Update function. The Stop_Host_Notify function ends the watch on the update of the presentation space and the OIA. Any presentation space or OIA updates that occur outside of the Start_Host_Notify and Stop_Host_Notify grouping will not be recognized by Query_Host_Update.

```
/*start checking host update bit                          */
/* P option is for presentation space updates*/
 rc = hllapi('Start_Host_Notify', session, 'P')

 rc= hllapi('Sendkey','Rdrlist'||'@E')

do until host_status=22   /* presentation space updated */
  host_status= hllapi('Query_Host_Update', session)
end

/* stop checking host update bit                           */
rc=hllapi('Stop_Host_Notify', session)

/* search for unique string found in rdrlist screen        */
 rc1=hllapi('Search_PS','Filename Filetype Class',81)

/* search for no rdr files string                          */
 rc2=hllapi('Search_PS','No files in your reader',1)

 if rc1>0 then
/* copy first row of rdrlist screen                        */
   rdrfile= hllapi('Copy_PS_to_Str',1,80)
```

Figure 98. Query Host Update Function

The flaw with this approach is that the do until loop could go on forever if the host is hung. Also, there are instances where the update bit is changed before the screen is actually updated. Another approach would be to use the Pause function instead of the Query_Host_Update function.

8.6.3 Pause Function

The Pause function causes a timed pause of n 1/2-second intervals to occur. It will return a value of 0 when the timer has expired. By changing the default, however, the Pause function will end when the host presentation space or OIA is updated. It will return a value of 26 in this instance and will not wait for the timer to expire. The Set_Session_Parameters function must be used to change the default for the Pause function. It needs to be changed to ″IPAUSE″. Figure 99 shows an example usage of the Pause function.

```
/* IPAUSE to break out of pause function  */
rc= hllapi('Set_session_parms','IPAUSE')

rc=hllapi('Start_Host_Notify', session, 'P')
rc=hllapi('Sendkey','rdrlist'||'@E')

pause_value= hllapi('Pause',120,session'#') /*pause for 60 seconds */
if pause_value = 26 then do
  /* search for unique string found in rdrlist screen */
  rc1=hllapi('Search_PS','Filename Filetype Class',81)

  /* search for no rdr files string                   */
  rc2=hllapi('Search_PS','No files in your reader',1)
  if rc1>0 then do
  /*copy first row of rdrlist screen                  */
    rdrfile= hllapi('Copy_PS_to_Str',1,80)
  end
end
else
  if pause_value=0 then say 'host timeout'

rc=hllapi('Stop_Host_Notify', session)
```

Figure 99. Pause Function

There are potential problems with this approach as well. There may be situations where the OIA bit is changed before the presentation space has actually been updated with the new screen. The Pause is terminated when

either the presentation space is updated or the OIA is updated, so your program may continue before the presentation space has actually been updated.

8.6.4 Wait Function

The Wait function is used to determine if the host is available. Wait returns a value of 0 if the keyboard is unlocked and ready for input. In its default state, Wait will wait for up to one minute for the host to become available. If the host is not available after one minute, it will time out with a non-zero return value. The one minute default can be changed to wait indefinitely, although that is not recommended since your program will basically be hung as long as the host is. See Figure 100 for an example of its usage.

```
rc= hllapi('Sendkey','Rdrlist'||'@E')
rc= hllapi('Wait')
if rc=0 then do  /* host command finished processing        */
 /* search for unique string found in rdrlist screen        */
  rc1=hllapi('Search_PS','Filename Filetype Class',81)

 /* search for no rdr files string                          */
  rc2=hllapi('Search_PS','No files in your reader',1)

  if rc1>0 then
    /* copy first row of rdrlist screen                     */
    rdrfile= hllapi('Copy_PS_to_Str',1,80)
end

else if ((rc=4) | (rc=5)) then do /*host busy or kbd locked */
   say 'host timeout'
   return
end
```

Figure 100. Wait Function

8.6.5 A Sample Host Checking Algorithm

There are many creative routines that can and have been written using combinations of these functions to manage host availability and presentation space updates. We have found that Wait, Pause, and Query_Host_Update will all give return values indicating that the presentation space has been updated when in actuality it has not been. We have developed a routine that

uses a combination of these three functions to provide a more reliable check to see if the presentation space has been updated and the host is available. It allows for a host settle time, which will ignore the premature setting of the update bit. This can be adjusted to fit the environment that the program will be running on. It also has a timeout feature so that you can decide how long your program will wait for the host before giving up.

```
/* IPAUSE to break out of pause function                              */
rc= hllapi('Set_session_parms','IPAUSE')
timeout_value=60  /* 60 seconds max wait time for host                */
rc=hllapi('Start_Host_Notify', session, 'P')

rc= hllapi('Sendkey','Rdrlist'||'@E')

rc= hllapi('Wait') /* wait for update bit                             */
rc=TIME('R') /* reset timer                                           */
/* loop until update has completed or timout reached                  */
do until ((pause_value=0) | (elapsed_time>timeout_value))
  pause_value= hllapi('Pause',6,session'#') /*settle time of 3 seconds*/
  rc=hllapi('Query_Host_Update',session)   /* clears update bit       */
  elapsed_time=TIME('E')                   /* get elapsed time        */
end
rc=hllapi('Stop_Host_Notify', session)

if elapsed_time > timeout_value then do
  say 'host timeout'
  return
end

/* copy first row of rdrlist screen                                   */
rdrfile= hllapi('Copy_PS_to_Str',1,80)
```

Figure 101. Host Check Using Wait, Pause, and Query_Host_Update

8.7 A Sample EHLLAPI Program - EHLRDR.CMD

EHLRDR.CMD, which can be found on the diskette, is a REXX program that uses the EHLLAPI to determine the number of NOTE files currently in a VM Rdrlist. We will break the program up into sections and analyze how it works.

Figure 102 on page 138 shows the main routine of EHLRDR.CMD. Get_PS_Dimensions is a routine that obtains the dimensions of the host

presentation space. The program will only attempt to process the Rdrlist if the host is in a VM READY or RUNNING state. FinishUp restores all defaults and disconnects from the host presentation space.

```
parse arg session                     /* host session to work with */
session=STRIP(session)
if session = '' then
   session  = 'A'

/* load hllapi functions                                          */
if Rxfuncquery('hllapi') then
  call Rxfuncadd 'HLLAPI','SAAHLAPI','HLLAPISRV'

/* connect to session                                             */
rc= hllapi('Connect',session)
rc= hllapi('Set_session_parms','IPAUSE')  /* for Pause function    */

call Get_PS_Dimensions
ps_num = row_num * col_num       /*total presentation space        */
cms_last_row = ps_num - col_num + 1   /* beginning of last row     */

/* determine current state of VM session                          */
rc1= hllapi('Search_PS','VM READ',cms_last_row)
rc2= hllapi('Search_PS','RUNNING',cms_last_row)

if (rc1>0 | rc2>0) then     /* found VMREAD or RUNNING            */
  call ProcessRdrList
else
  say ' Host is not in a VM READY or VM RUNNING state.'
        ' Terminating program.'

Call FinishUp

return
```

Figure 102. Main Routine of EHLRDR.CMD

```
/*****************************************************/
Get_PS_Dimensions: /*Procedure Expose row_num col_num */

rc=hllapi('Query_Session_Status', session)
parse var rc 12 bin_row 14 bin_col 16 /* get binary   */
row_num = c2d(reverse(bin_row)) /* convert to decimal */
col_num = c2d(reverse(bin_col))

return
```

Figure 103. Get_PS_Dimensions Routine

```
/*****************************************************/
FinishUp:

 rc= hllapi('Disconnect')
 rc=hllapi('Reset_System')  /* restore defaults */

Return
```

Figure 104. FinishUp Routine

The Sendkey_and_wait routine is derived from the host availability check routine in Figure 101 on page 137. It will send a string of data to a host session and wait until the host has processed the data and becomes available again. Note that this means data strings passed to this routine should force the host to do some processing, which usually means an Enter key or PF key mnemonic will be concatenated to the end of the string. If the data string consists of only text data being written to the host, depending on how the host application is developed the host update bit will not be updated and the Sendkey_and_wait routine will simply timeout. Figure 105 on page 140 shows the Sendkey_and_wait routine.

```
/*****************************************************/
Sendkey_and_wait:
parse arg session, keystring
host_value=0            /* set return value                          */
timeout_value=60
rc=hllapi('Start_Host_Notify', session, 'P')
rc= hllapi('Sendkey',keystring)
rc= hllapi('Wait')     /* wait for update bit                        */
rc=TIME('R')           /* reset timer                                */

/* loop until update has completed or timout reached                 */
do until ((pause_value=0) | (elapsed_time>timeout_value))
  pause_value= hllapi('Pause',6,session'#')   /*3 second settle time  */
  qhu_value=hllapi('Query_Host_Update',session)  /* clears update bit */
  elapsed_time=TIME('E')          /* get elapsed time */
end
rc=hllapi('Stop_Host_Notify', session)

if elapsed_time > timeout_value then   /* set return value           */
  host_value=99

return host_value
```

Figure 105. Sendkey_and_wait Routine

Host_error is a routine that could be tailored for a specific environment. It is used here as an error handler for host timeouts on calls to Sendkey_and_wait.

```
/*****************************************************/
Host_error:

/* could call a diagnostic routine here to determine the problem     */
/* possibly the host dropped the connection and you must logon again */

say ' host timeout error. Terminating program.'
return
```

Figure 106. Host_error Routine

Get_To_RdrList_Screen and Leave_Rdr_List_Screen utilize the Sendkey_and_wait routine and Search_PS to get in and out of the Rdrlist environment. Notice that Get_To_RdrList_Screen will issue the clear screen keystroke to the host if necessary.

```
/*****************************************************/
Get_To_Rdrlist_Screen:
parse arg session, cms_last_row
keystring='Rdrlist'||'@E'
if Sendkey_and_wait(session, keystring) = 99 then do
   call host_error
   return 99
end

/* clear screen if necessary                                          */
rchold= hllapi('Search_PS','HOLDING',cms_last_row)
rcmore= hllapi('Search_PS','MORE...',cms_last_row)
do while ((rchold > 0 | rcmore>0))
  keystring='@C'
  if Sendkey_and_wait(session, keystring) = 99 then do
     call host_error
     return 99
  end
  rchold= hllapi('Search_PS','HOLDING',cms_last_row)
  rcmore= hllapi('Search_PS','MORE...',cms_last_row)
end /* do loop */

rc1=hllapi('Search_PS','Filename Filetype Class',81) /* start in row 2*/
rc2=hllapi('Search_PS','No files in your reader',1)
if rc2>0 then do
  say ' No reader files. Terminating program.'
  return 88
end
if rc1=0 then do
  say 'possible system error - Rdrlist screen did not appear'
  return 77
end

return 0
```

Figure 107. Get_To_RdrList_Screen Routine

```
/*****************************************************/
Leave_Rdrlist_Screen:
parse arg session

keystring='@3'           /* pf3 to leave rdrlist */
if Sendkey_and_wait(session, keystring) = 99 then do
  call host_error
  return 99
end

rcready= hllapi('Search_PS','Ready;',1)
if (rcready=0) then do
   say 'possible system error - problem leaving rdrlist'
   return 77
end

return 0
```

Figure 108. Leave_RdrList_Screen Routine

ProcessRdrList is the routine that scans the reader looking for files with a file type of NOTE. In order to manage the Rdrlist screen, the details of the layout of the screen must be incorporated into the program. This brings up another issue with writing programs using EHLLAPI that manipulate host screens. In many situations if the host application developers change the layout of a host screen, then your EHLLAPI program that uses that host screen will have to change as well.

Notice that the program reads data a line at a time from the host screen by using Copy_PS_To_Str and keeping track of the current location on the screen. The program must also be able to determine when the bottom of the screen has been reached so it can issue a PF8 keystroke to advance to the next screen.

```
/*************************************************/
ProcessRdrList: Procedure Expose session row_num col_num cms_last_row

/* turn host messages off - avoid screen holds */
keystring='SET MSG OFF'||'@E'     /* @E is the symbol for enter key */
if Sendkey_and_wait(session, keystring) = 99 then do
   call host_error
   return
end

if Get_To_RdrList_Screen(session, cms_last_row) \= 0 then
  return

/* write header to display                                        */
say 'You have notes from the following users'
say ' waiting in your VM reader:'
say
say 'USERID             NODE'
say '--------           --------'
/*look for note files                                            */
line=1              /* rdrlist next screen indicator              */
note_cnt=0       /* number of note files in rdr                   */

row_3=col_num*2 + 1  /* set position values for key rows          */
row_4=col_num*3 + 1

/* set ptr to 1st position of row 3 of screen                     */
row_ptr=row_3
last_rdr_row= (row_num*col_num) - (5*col_num) + 1
rdrfile= hllapi('Copy_PS_to_Str',row_ptr,col_num) /* copy row     */
/* string is not a blank line, therefore not end of rdr list      */
do while VERIFY(rdrfile,' ')>0
  if (Substr(rdrfile,16,8) = "NOTE    ") then do  /* check filetype */
    parse var rdrfile 31 userid 39 40 node 48
    say userid '        ' node

    note_cnt = note_cnt+1
  end
```

Figure 109 (Part 1 of 2). ProcessRdrList

```
  row_ptr=row_ptr+col_num           /* increment pointer to next row */
  if row_ptr= last_rdr_row then do  /* need to get to next screen    */
     row_ptr = row_4          /* row 3 is a repeat from prior screen */
     line=line+row_num-8      /* next screen indicator               */

     keystring='@8'
     if Sendkey_and_wait(session, keystring) = 99 then do
       call host_error
       return
     end
     rc1=hllapi('Search_PS','Line='||line,1) /* look for next screen */
      if (rc1=0) then do
        say 'possible system error - next Rdrlist screen did not appear'
        return
      end
  end  /* if then */

  /* copy next row to string                                          */
  rdrfile= hllapi('Copy_PS_to_Str',row_ptr,col_num)
end  /* do loop */
say
say 'There are a total of ' note_cnt ' notes in your VM reader'

rc=Leave_Rdrlist_Screen(session)

keystring='SET MSG ON'||'@E'             /* turn messages on          */
if Sendkey_and_wait(session, keystring) = 99 then do
  call host_error
  return
end

Return
```

Figure 109 (Part 2 of 2). ProcessRdrList

8.8 Sending and Receiving Files

The CM/2 Send and Receive functions are accessible through EHLLAPI via the Send_file and Receive_file functions respectively. The Send_file function allows you to send files from a PC to a host session. The Receive_file function allows you to receive files at a PC from a host session. To use the Send_file and Receive_file functions successfully, you must **not**:

- Be connected to the same session
- Have another file transfer application active on the same session
- Have another EHLLAPI application active on the same session

For more information on the CM/2 Send and Receive functions refer to *Communications Manager/2 User's Guide*. Defaults can be changed using the Set_session_parms function.

8.8.1 Example - EHLSF.CMD

EHLSF.CMD is a REXX program that uses the EHLLAPI Send_file function to send the user INI (OS2.INI) and the system INI (OS2SYS.INI) to a host session. The Send_file function requires the full path name of the file being sent. The full path name for the INI files is obtained from the OS2ENVIRONMENT variable in this example. The timeout for the file transfer is set for 5 minutes through a call to Set_session_parms.

The main routine checks to see if the host is in a VM READY or RUNNING state, and then disconnects. The program cannot be connected to the host session when the Send_file call is issued. The main routine then calls the SendIniFiles routine.

```
/*EHLSF.CMD                                                          */
/* send OS2.INI , OS2SYS.INI files to host for backup                */

parse arg session                     /* host session to work with  */
if session = ''  then
   session  = 'A'
 /* load hllapi functions */
 if Rxfuncquery('hllapi')  then
   call Rxfuncadd 'HLLAPI','SAAHLAPI','HLLAPISRV'

 /* connect to session */
 rc= hllapi('Connect',session)        /* connect to host session    */
 rc= hllapi('Set_session_parms','TIMEOUT=J') /* 5 minutes timeout   */
                                      /* on send, receive           */
/* check status of VM session                                        */
 rc1= hllapi('Search_PS','VM READ',1)
 rc2= hllapi('Search_PS','RUNNING',1)

/* must disconnect before sending file                               */
 rc= hllapi('Disconnect')

 if (rc1>0 | rc2>0) then     /* found VMREAD or RUNNING               */
   call SendIniFiles
 else
    say ' host is not in a VM READY or VM RUNNING state.'
    say ' Terminating program.'
return
```

Figure 110. EHLSF.CMD Main Routine

The SendIniFiles function prepares the string that is used in the Send_file call, and then issues the calls to send the INI files to the host session.

```
/*****************************************************/
SendIniFiles: Procedure Expose session

/* get path of system and user INI files                          */
os2ini=VALUE(SYSTEM_INI,,OS2ENVIRONMENT)
userini=VALUE(USER_INI,,OS2ENVIRONMENT)

/* concatenate host session , filename                            */
os2_ini_string=os2ini session||':' 'OS2SYS INI A '
user_ini_string=userini session||':' 'OS2 INI A '

/* send system INI file to host                                   */
rc=hllapi('Send_file',os2_ini_string)
if rc \=3 then
  say 'Error sending file ' os2ini ' to host session '
  say session||'.  RC = ' rc

/* send user INI file to host                                     */
rc=hllapi('Send_file',user_ini_string)
if rc \=3 then
  say 'Error sending file ' os2ini ' to host session '
  say session||'.  RC = ' rc

return
```

Figure 111. EHLSF.CMD SendIniFiles Routine

8.8.2 Example - EHLRECV.CMD

EHLRECV.CMD is a REXX program that uses the EHLLAPI Receive_file function to receive the user INI (OS2.INI) and the system INI (OS2SYS.INI) from a host session. The Receive_file function requires the full path name where the file will be received. The timeout for the file transfer is set to 5 minutes through a call to Set_session_parms.

The main routine checks to see if the host is in a VM READY or RUNNING state, and then disconnects. The program cannot be connected to the host session when the Receive_file call is issued. The main routine then calls the ReceiveIniFiles routine.

```
/*EHLRCVE.CMD                                                        */
/* receive OS2.INI , OS2SYS.INI files from host                      */

parse arg session                     /* host session to work with */
if session = ''  then
   session  = 'A'

 /* load hllapi functions                                           */
 if Rxfuncquery('hllapi')  then
   call Rxfuncadd 'HLLAPI','SAAHLAPI','HLLAPISRV'

 /* connect to session                                              */
 rc= hllapi('Connect',session)     /*      connect to host session */
 rc= hllapi('Set_session_parms','TIMEOUT=J') /* 5 minute timeout   */
                                        /* on send, receive        */

 rc1= hllapi('Search_PS','VM READ',1)
 rc2= hllapi('Search_PS','RUNNING',1)

/* must disconnect before sending file                              */
 rc= hllapi('Disconnect')

 if (rc1>0 | rc2>0) then     /* found VMREAD or RUNNING              */
   call ReceiveIniFiles
 else
    say ' host is not in a VM READY or VM RUNNING state.'
    say 'Terminating program.'

return
```

Figure 112. EHLRECV.CMD Main Routine

The ReceiveIniFiles function prepares the string that is used in the Receive_file call, and then issues the calls to receive the INI files from the host session.

```
ReceiveIniFiles: Procedure Expose session

/* get path of system and user INI files                              */
os2ini=VALUE(SYSTEM_INI,,OS2ENVIRONMENT)
userini=VALUE(USER_INI,,OS2ENVIRONMENT)

/* overlay.ini extension with .bak                                    */
os2ini=Overlay("BAK",os2ini,LENGTH(os2ini)-2)
userini=Overlay("BAK",userini,LENGTH(userini)-2)

/* concatenate host session , filename                                */
os2_ini_string= os2ini session||':'||'OS2SYS INI A '
user_ini_string= userini session||':'||'OS2 INI A '

/* receive system INI file from host */
rc=hllapi('Receive_file',os2_ini_string)
if rc \=3 then
  say 'Error receiving file ' os2ini ' from host session '
  say session||'.  RC = ' rc

/* receive user INI file from host                                    */
rc=hllapi('Receive_file',user_ini_string)
if rc \=3 then
  say 'Error receiving file ' os2ini ' from host session '
  say session||'.  RC = ' rc

return
```

Figure 113. EHLRECV.CMD ReceiveIniFiles Routine

8.9 Manipulating the Presentation Space Window

Thus far we have examined how EHLLAPI functions can manipulate the host session presentation space. The EHLLAPI can also manipulate the Presentation Manager window that contains a host session presentation space. The EHLLAPI functions that interface with the PM window are the following:

- **Change_Switch_Name** - Change the name of the session listed on the Task List.
- **Change_Window_Name** - Change the window title bar.
- **Connect_PM** - Connect the REXX application to the presentation space window.
- **Disconnect_PM** - Disconnect the REXX application from the presentation space window.
- **Get_Window_Status** - Return the current presentation space window status.
- **Lock_PMSVC** - Lock or unlock the presentation space window.
- **Query_Window_Coord** - Return the presentation space window coordinates.
- **Set_Window_Status** - Change the presentation space window status.

Similar to issuing a Connect function call to connect to a host presentation space, a Connect_PM function call is used to establish a connection between a REXX program and the host presentation space window.

Chapter 9. REXX Interfaces to DB2/2

Before DB2/2, Database Manager was the database solution for the OS/2 platform. There are some similarities between Database Manager and DB2/2. For example, both provide the Query Manager facility, a Graphical User Interface (GUI) application that can be used to perform many functions, from creating and authorizing databases to issuing SQL statements and updating tables. DB2/2 has some advantages over Database Manager, however. DB2/2 has better performance and provides more consistency across platforms. Since DB2* is IBM′s SAA database product, DB2 is available in some form on most other platforms, for example VM, MVS, and OS/400. Distributed Database Connection Services/2* (DDCS/2) and other products provide the opportunity to link DB2/2 databases with DB2 databases on other platforms, paving the way for database client/server applications.

DB2/2 provides an interface to REXX programs in three basic ways:

1. The SQLDBS DB2/2 API allows REXX programs to invoke command-like versions of DB2/2′s API set.
2. The SQLEXEC DB2/2 API allows REXX programs to invoke SQL statements.
3. DB2/2 provides data structures that are accessible by REXX programs.

This chapter discusses how these interfaces can be used to create useful REXX programs. Example programs are provided and explained. The installation and setup required for REXX programs to access remote workstation databases is also provided. Please refer to Appendix B, “OS/2 DB2/2 REXX Reference” on page 245 for information on SQLDBS and SQLEXEC statements, as well as the DB2/2 data structures.

9.1 DB2/2 Installation and Setup

When installing DB2/2, you are prompted to enter the environment that your workstation will be operating in:

1. Stand-alone
2. Client
3. Client with local databases
4. Server

In our environment we chose to have two Personal System/2* machines, each with the server version of DB/2 installed. The workstations were both on the same LAN, with both using LAPS and LAN Requester with NetBIOS. So in effect, we have a multiple server environment, where each workstation can be the client of the other. In other words, workstation 1 can act as a client and access databases on workstation 2, and workstation 2 can act as a client and access databases on workstation 1. In order to simplify things, we will refer to a client workstation and a server workstation for the rest of this chapter. Keep in mind that the client workstation has its own local databases as well. We found that if coded properly, the same REXX program can access and manipulate databases on either the local or remote workstation. That is, it is transparent to the program which database is being accessed. If cataloged, DB2/2 will find the database′s location.

9.2 How to Register DB2/2 Functions

The DB2/2 APIs provided for REXX are external function packages. They must be registered by your REXX program, just like any other external function package, by using RxFuncAdd. The two APIs are SQLDBS and SQLEXEC. Just like other external function packages, once registered these functions are available to all REXX programs running on your system until they are dropped. When they are dropped, all REXX programs running on your system lose access to these functions. Therefore we recommend that you do not drop these functions at the end of your programs.

Also, for workstations that run REXX DB2/2 applications on a regular basis, it is wise to write a REXX procedure that registers these functions and invoke that procedure in the STARTUP.CMD. This removes the overhead of registering these functions from your DB2/2 REXX applications. Figure 114 on page 153 shows the REXX code for registering SQLDBS and SQLEXEC.

```
/* Register SQLDBS if not already registered                          */
if Rxfuncquery('SQLDBS') <> 0 then do
rc = RxFuncAdd('SQLDBS' , 'SQLAR' , 'SQLDBS')
  if rc \= 0 then do
    say "Error registering SQLDBS: rc = " rc
    return
  end  /* Do */
end

/* Register SQLEXEC if not already registered                          */
if Rxfuncquery('SQLEXEC') <> 0 then do
  rc = RxFuncAdd('SQLEXEC', 'SQLAR', 'SQLEXEC')
  if RC \= 0 then do
    say "Error registering SQLEXEC: rc = " rc
    return
  end  /* Do */
end
```

Figure 114. Registering SQLDBS and SQLEXEC

9.3 User Profile Management (UPM)

User Profile Management Services is an icon on the desktop. Through UPM you can log on to a LAN (if you are connected to one), your local logon user ID, and remote nodes (for example workstations). For this discussion on DB2/2, we are interested in the local and node logons.

Access to databases located on a workstation is controlled through the workstation's local logon user ID. The default user ID is USERID. The default password is PASSWORD. These can be changed using UPM. Through this local logon you can authorize other nodes to be able to access your workstation. In order for other nodes(for example remote workstations) to be able to access your DB2/2 databases, those nodes must be authorized in UPM. For example, the server workstation must authorize the client nodes. To authorize a remote node (or group of remote nodes):

1. Double click on the **User Profile Management Services** icon.
2. Double click on the **User Profile Management** icon. If you are not logged on to your local logon, you will be prompted to do so.

3. A **User Profile** window will then appear. Your user type should be Administrator, since this is your machine. Choose **Manage** from the action bar.
4. Choose **Manage Users** (or **Manage Group** to give a group of nodes the same authorization).
5. From here you are prompted to authorize new users, with radio buttons for logon access (make sure it is yes if you want them to be able to access your databases). Also, if you choose to require a password when the node logs on to your machine, be sure that the user or application on that node knows the password or they won't be able to access your databases.

When a DB2/2 command is issued that attempts to access a cataloged database on a remote node, if the workstation is not already logged on to the remote node, DB2/2 will bring up a logon window, prompting you to log on to that node. This is where the check will actually be made to see if the workstation has UPM authority from the remote node.

9.4 DB2/2 Database Administration

In order for a client workstation to be able to access the server's databases and tables, certain authorizations have to take place on the server workstation. In addition, the client workstation has to update its DB2/2 system catalog with information about the server databases. REXX programs can perform many database administration functions through the use of the SQLDBS and SQLEXEC APIs. See Appendix B, "OS/2 DB2/2 REXX Reference" on page 245 for a complete listing of SQLDBS APIs and SQL statements.

9.4.1 Server Workstation Database Administration

DB2/2 provides database security functions allowing the server to grant and revoke database access to clients. These authorizations are specific by client. In other words, it can be set up so that some clients have access to a certain database, while others do not. You can also determine whether or not you will allow a client to create a table in the server database. These types of database authorizations can be done through the Query Manager. They can also be done in REXX programs through SQL statements. The following examples are part of the DB22DBA.CMD which is on the diskette. Note that the workstation running the REXX program must have system administrator authority in order for these database administration commands to run successfully. The first example in Figure 115 on page 155 shows how

to grant access to a database for a client using the SQL GRANT statement. The second example shows how to revoke access to a database from a client using the SQL REVOKE statement. Note that the GRANT and REVOKE statements cannot be issued directly with SQLEXEC. They must be included in a string and then invoked with SQLEXEC using the EXECUTE IMMEDIATE statement. For the syntax of the GRANT and REVOKE statements and other SQL statements, as well as a breakdown of which SQL statements can be issued to SQLEXEC directly and which must use EXECUTE IMMEDIATE, refer to Appendix B, "OS/2 DB2/2 REXX Reference" on page 245.

```
pull dbname                  /* database name                        */
pull authname                /* node (workstation id) that           */
                                  /* you wish to authorize           */

/* connect to database required for GRANT statement                  */
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then do
  say ' Could not grant access to database ' ,
        dbname ' for user ' authname
  return
end

/* GRANT cannot be executed directly from REXX                       */
/* EXECUTE IMMEDIATE needs to be used                                */
stmt1 ='GRANT CONNECT ON DATABASE TO ' authname
call sqlexec 'EXECUTE IMMEDIATE :stmt1';

if ( SQLCA.SQLCODE <> 0) then
  say ' Could not grant access to database ' ,
        dbname ' for user ' authname
else
 say ' Granted access to database ' ,
        dbname ' for user ' authname

/* clean up - disconnect from database -                             */
call sqlexec 'CONNECT RESET';

return
```

Figure 115. Grant Access to Database

```
pull dbname                  /* database name                        */
pull authname                /* node (workstation id) that           */
                             /* you wish to revoke                   */

/* connect to database required for REVOKE statement                 */
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then do
  say ' Could not revoke access from database ' ,
        dbname ' for user ' authname
  return
end

/* REVOKE cannot be executed directly from REXX               */
/* EXECUTE IMMEDIATE needs to be used                              */
stmt1 ='REVOKE CONNECT ON DATABASE FROM ' authname
call sqlexec 'EXECUTE IMMEDIATE :stmt1';

if ( SQLCA.SQLCODE <> 0) then
  say ' Could not revoke access from database ' ,
        dbname ' for user ' authname
else
 say ' Revoked access from database ' ,
        dbname ' for user ' authname

/* clean up - disconnect from database -                           */
call sqlexec 'CONNECT RESET';

return
```

Figure 116. Revoke Access from Database

Granting a client access to a server database does not necessarily mean that the client has access to tables and/or views in the database. DB2/2 provides another level of security, where the server can selectively grant and revoke specific types of access to tables and views within the database. These authorizations are specific by client. In other words, it can be set up so that some clients have access to certain tables in a database, while others do not. You can also determine whether or not you will allow a client to update a table in the server database. These types of authorizations can be done through the Query Manager. They can also be done in REXX programs through SQL statements. The following examples are part of the DB22DBA.CMD which is on the diskette. The first example shows how to grant access to a table for a client using the SQL GRANT statement. The

second example shows how to revoke access to a table from a client using the SQL REVOKE statement.

```
pull dbname                 /* database name                          */
pull tablename              /* table name                             */
pull authname               /* node (workstation id) that             */
                            /* you wish to authorize                  */

/* connect to database required for GRANT statement                   */
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then do
  say ' Could not grant access to table '
        tablename ' for user ' authname
  return
end

/* GRANT cannot be executed directly from REXX                        */
/* EXECUTE IMMEDIATE needs to be used                                 */
stmt1 ='GRANT SELECT ON TABLE ' tablename ' TO ' authname
call sqlexec 'EXECUTE IMMEDIATE :stmt1';

if ( SQLCA.SQLCODE <> 0) then
  say ' Could not grant access to table ' ,
        tablename ' for user ' authname
else
 say ' Granted access to database ' ,
        dbname ' for user ' authname

/* clean up - disconnect from database -                              */
call sqlexec 'CONNECT RESET';

return
```

Figure 117. Grant Access to Table

```
pull dbname                 /* database name                          */
pull tablename              /* table name                             */
pull authname               /* node (workstation id) that             */
                                  /* you wish to revoke               */

/* connect to database required for REVOKE statement                  */
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then do
  say ' Could not revoke access from table ' ,
        tablename ' for user ' authname
  return
end

/* REVOKE cannot be executed directly from REXX                       */
/* EXECUTE IMMEDIATE needs to be used                                 */
stmt1 ='REVOKE SELECT ON TABLE ' tablename ' FROM ' authname
call sqlexec 'EXECUTE IMMEDIATE :stmt1';

if ( SQLCA.SQLCODE <> 0) then
  say ' Could not revoke access from table ' ,
        tablename ' for user ' authname
else
 say ' Revoked access from table ' ,
        tablename ' for user ' authname

/* clean up - disconnect from database -                              */
call sqlexec 'CONNECT RESET';

return
```

Figure 118. Revoke Access from Table

9.4.2 Client Workstation Database Administration

Each workstation that has DB2/2 installed also has a database directory and a node directory. The DB2/2 database directory contains information about databases. Database name, database alias name, workstation name where database resides, and database type are examples of what is entered in the directory for each database. When DB2/2 is instructed to use a database in some way (whether through Query Manager or a running program), DB2/2 uses the directory to determine where the database is located. If the database is on another node, the node directory must be accessed to get information about the node.

The node directory contains information such as node (workstation) name, adapter number, and protocol used to connect to the node. This information is used by DB2/2 to access remote databases.

Client workstations that need to access server workstation database(s) have to catalog the server′s node in the client′s node directory, and the server′s database(s) in the client′s database directory. These tasks can be accomplished through the DBM command set, and can also be done through REXX programs using the DB2/2 APIs.

The following examples are part of the DB22DBA.CMD which is on the diskette. The first example, Figure 119 shows how to catalog a node in the node directory using the CATALOG DB2/2 API. The node is connected via APPC. APPN** and NetBIOS are also supported and examples are found in the DB22DBA.CMD on the diskette.

```
/*****************************************************/
/* APPC_CONNECTION                                   */
/*   This routine catalogs an APPC node              */
/*****************************************************/

  say ' Please enter the NODENAME'
  pull nodename
  say ' Please enter the Local LU'    /*CM/2 SNA workstation name */
  pull locallu

  say ' Please enter the Partner LU'  /* CM/2 SNA server name     */
  pull partnerlu

  say ' Please enter the SNA transmission service mode '
  pull mode

  /* build string                                                 */
  SQL_EndString = 'LOCAL 'locallu 'REMOTE 'partnerlu 'MODE 'mode
  SQL_String = 'CATALOG APPC NODE 'nodename SQL_EndString
  call SQLDBS SQL_String
  if SQLCA.SQLCODE = 0 then do
    say ' Node 'nodename 'was sucessfully cataloged.'
  else '
    say ' catalog failed, return code = ' SQLCA.SQLCODE

  return
```

Figure 119. Catalog APPC Node

The example in Figure 120 removes a node entry from the node directory using the UNCATALOG statement. This is from DB22DBA.CMD on the diskette. This example works for APPC, APPN, and NetBIOS nodes.

```
/*******************************************************/
/* UNCATALOG_NODE                                      */
/*   This routine uncatalogs a node                    */
/*******************************************************/

  say ' Please enter the name of the node to be uncataloged'
  pull nodename
  call SQLDBS 'UNCATALOG NODE 'nodename
  if SQLCA.SQLCODE = 0 then
    say ' Node 'nodename 'was successfully uncataloged.'
  else
    say ' Node 'nodename ,
          ' could not be uncataloged. Error = ' SQLCA.SQLCODE
  return
```

Figure 120. Uncatalog Node

The example in Figure 121, part of DB22DBA.CMD on the diskette, shows how to catalog a remote database on the database directory using the CATALOG DB2/2 API.

```
/*******************************************************/
/* CATALOG_REMOTE                                      */
/*   This routine catalogs a remote database.          */
/*******************************************************/
  say 'Please enter the name of the database to be cataloged'
  pull dbname

  say 'Please enter the database alias name'
  pull alias

  say ' Please enter the node name this database is to be cataloged on:'
  pull node

  /* build string                                     */
  if alias = "" then
    SQL_String = 'CATALOG DATABASE 'dbname  'AT NODE 'node
  else
    SQL_String = 'CATALOG DATABASE 'dbname 'AS 'alias ' AT NODE 'node

  /* issue catalog database statement             */
  call SQLDBS SQL_String
  if SQLCA.SQLCODE = 0 then
    say ' Database 'dbname ' at 'node ,
        'was cataloged with the alias name' alias
  else
    say ' Database 'dbname ' at 'node ,
        ' was not cataloged. Error= 'SQLCA.SQLCODE
  return
```

Figure 121. Catalog Remote Database

The example in Figure 122 on page 163, part of DB22DBA.CMD on the diskette, uncatalogs a database on the database directory using the UNCATALOG DB2/2 API.

```
/************************************************************/
/* UNCATALOG_DATABASE                                       */
/*   This routine removes a database from the system catalog. */
/************************************************************/
  say ' Please enter the name of the database to be uncataloged'
  pull dbname
  call SQLDBS 'UNCATALOG DATABASE 'dbname
  if SQLCA.SQLCODE = 0 then
    say ' Database 'dbname 'was successfully uncataloged.'
  else
    say ' Database 'dbname ' was not uncataloged. Error= ' SQLCA.SQLCODE
   return
```

Figure 122. UnCatalog Remote Database

9.5 Embedding Structured Query Language (SQL) Statements in REXX Programs

SQL is available on all DB2 platforms. It is a powerful language designed with many functions that allow you to use and manipulate the data in DB2 databases. Some examples of what SQL statements can do:

1. Create tables and views.
2. Add rows to a table.
3. Update existing rows.
4. Create reports based on select conditions.
5. Grant authorities to a user.

In DB2/2, you can invoke SQL statements by using the Query Manager. You can also embed SQL statements in REXX programs.

9.5.1 Static vs. Dynamic SQL

Static SQL statements are statements that are prepared prior to the execution of the program that contains them. The complete SQL statement must be known prior to the compilation of the program. During compilation, an executable form of the SQL statement is created. Having said that, since OS/2 REXX is an interpreted language and not a compiled language, static SQL statements are not possible in REXX. All SQL statements in REXX

programs are prepared when the program runs. The term used for this type of statement is dynamic SQL statement.

Both static and dynamic SQL statements have advantages. Static SQL statements often process faster than dynamic SQL statements because the overhead of preparing the statement is done before the program actually runs. However, dynamic statements offer more flexibility because the actual form of the SQL statement does not need to be known before the program runs. For example, if you want to use the SELECT statement to query a table and load the results into variables, in a static SQL situation you would need to know the number of columns in the table and their names before your program is compiled. If there are changes to the table, chances are your program would have to be recompiled. Using a dynamic SQL statement in this situation, the number of columns, their names, lengths, and data types do not need to be known in advance. They can be obtained during the execution of the program. This use of dynamic SQL is called a varying list SELECT, and it can be very useful. There is an example of this on the diskette called SELECT.CMD. This will also be discussed in greater detail later in this chapter.

There are certain SQL statements that can be passed directly to the SQLEXEC API. Others require the use of the PREPARE statement before a call to SQLEXEC can be made, or they must be prefaced by the EXECUTE IMMEDIATE statement. A breakdown of the SQL statements is provided in Appendix B, “OS/2 DB2/2 REXX Reference” on page 245.

9.5.2 SELECT Statement

Before we get into the varying list use of the SELECT statement discussed earlier, take a look at a less complex usage of a SELECT in a REXX program. The example in Figure 123 on page 165 is taken from GETTABLE.CMD, which can be found on the diskette. In this example, the table SYSIBM.SYSTABLES is being queried in order to get a list of tables that are associated with a given database.

Note the convention for table names. It is the creator name concatenated to the table name with a period. So in this case the creator is SYSIBM and the table name is SYSTABLES. If the creator name is not used when referring to a table, the default is the workstation ID that the program is running on. We recommend that you always use the full table name, as this will make your code flexible in that it can refer to tables on other systems. The columns in SYSIBM.SYSTABLES that we want to retrieve are the table name and the table creator. For each row retrieved, we display the table name and table creator. The SELECT could conceivably produce a result of multiple rows.

The way that the program traverses through the rows is by using a cursor. The cursor points to the row currently being processed, beginning with the first row. The cursor is incremented to point to the next row by DB2/2 automatically when you request the next row using the FETCH statement. These are the steps used to code this type of program:

1. CONNECT to the database.
2. Create the SELECT statement.
3. Use the PREPARE statement to dynamically build the SELECT statement.
4. Use the DECLARE statement to define a cursor and associate the cursor to the SELECT statement.
5. Use the OPEN statement to initialize the cursor pointing to the first row.
6. Use the FETCH statement in a loop to retrieve rows into variables.
7. Use the CLOSE statement to release the cursor.
8. Use the COMMIT statement to complete the unit of work.

```
/*   This function  lists all tables for a given database.  */
/************************************************************/
parse arg dbname                       /* database name      */

/**** connect to database  *****/
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine

st = "SELECT name,creator FROM sysibm.systables " ,  1
         " WHERE creator <> ? AND creator <> ?";

call sqlexec 'PREPARE s1 FROM &:.st';                          2
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine

call sqlexec 'DECLARE c1 CURSOR FOR s1';                  3
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine

parm_var1= "SYSIBM";           /* do not view system tables */
parm_var2= "QRWSYS";           /* do not view system tables */

call sqlexec 'OPEN c1 USING &:.parm_var1,&:.parm_var2';   4
```

Figure 123 (Part 1 of 2). SELECT Statement Example

```
call sqlexec 'FETCH c1 INTO &:.table_name,&:.creator_name';   5
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine

/* loop through query results, display table names          */
do while ( SQLCA.SQLCODE = 0 )
  if (SQLCA.SQLCODE = 0) then do
    say 'Table = ' table_name  '              Creator = ' creator_name
    say ""
  end
  call sqlexec 'FETCH c1 INTO &:.table_name,&:.creator_name';   6
end    /* end Do */

call sqlexec 'CLOSE c1';
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine
call sqlexec 'COMMIT';
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine
/* clean up - disconnect from database                      */
call sqlexec 'CONNECT RESET';
if ( SQLCA.SQLCODE <> 0) then call Error_handling routine
return
```

Figure 123 (Part 2 of 2). SELECT Statement Example

Notes:

1 The SELECT statement is created using ? as a place holder for an unknown value. This place holder will be resolved in the OPEN cursor statement.

2 The PREPARE statement validates the SQL SELECT statement and prepares it for execution. Note that statement names must fall in the range S1 to S100.

3 The DECLARE statement associates cursor C1 with statement S1. Note that cursor names must fall in the range C1 to C100, with C51 to C100 only used for cursors declared with the WITH HOLD option.

4 The OPEN statement replaces the question marks in the SELECT statement with the values of parm_var1 and parm_var2 respectively. It also initalizes cursor C1 to point to the first row that satisfies the SELECT conditions.

5 The FETCH statement is moving the NAME and CREATOR columns from the current row into the REXX variables table_name and creator_name respectively.

6 Place another FETCH statement inside a loop to retrieve the remaining rows.

9.5.3 Varying List SELECT

For situations where it is required to query, update or delete rows in a table, and for whatever reason the table column names cannot be hard coded into the program, the varying list SELECT technique can be used. It may be that you do not know ahead of time how many columns are in a table or what their names are. It may be that you need your program to be flexible enough to work on different tables, and adapt to changes in those tables. The term varying list SELECT means that when the program is started, the number and types of columns to be returned are not known. DB2/2 has a data structure call SQLDA which provides very useful information about the columns in a table. These are the steps used to code this type of program:

1. CONNECT to the database.
2. Create the SELECT statement.
3. Use the DECLARE statement to define a cursor and associate the cursor to the SELECT statement.
4. Use the PREPARE statement with the INTO clause to dynamically build the SELECT statement and specify a REXX variable into which column information from the SQLDA will be loaded.
5. Use the DESCRIBE statement to load column information from the SQLDA into a REXX array variable.
6. Use the OPEN statement to initialize the cursor pointing to the first row.
7. Use the FETCH statement with the USING DESCRIPTOR clause in a loop to retrieve rows into the REXX array variable.
8. Use the CLOSE statement to release the cursor.
9. Use the COMMIT statement to complete the unit of work.

The structure of the SQLDA can be found in Appendix B, “OS/2 DB2/2 REXX Reference” on page 245. It builds a two-dimensional array consisting of n columns with 5 elements for each column. These 5 elements contain information about the column. One comment regarding the SQLLEN element: SQLLEN contains the length of a column. We have noticed that if the column has a decimal length, the SQLLEN element will not be filled in for that

column. For example, if a column has a length of 7.2, the SQLLEN element will be uninitialized, meaning its value is ″SQLLEN″.

The example in Figure 124, taken from SELECT.CMD on the diskette, is an example of the varying list SELECT. This example queries a table, and writes each row of the query results to a file.

```
/*  Run a varying list SELECT and load  the results into TABLE.DAT */

/**** connect to database  *****/
call sqlexec 'CONNECT TO ' dbname ' IN SHARE MODE';
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

st1 = ″SELECT *  FROM ″ creator||'.'||tablename
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

call SQLEXEC 'PREPARE s1 INTO &:.sqldavar FROM :st1'        1
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

call SQLEXEC 'DESCRIBE s1 INTO &:.sqldavar'                 2
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

call SQLEXEC 'OPEN c1'
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine
```

Figure 124 (Part 1 of 2). Varying List SELECT

```
call SQLEXEC 'FETCH c1 USING DESCRIPTOR &:.sqldavar'            3
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine

'ERASE TABLE.DAT  > NULL'  /* erase TABLE.DAT if it already exists */

do while (SQLCA.SQLCODE = 0)         /* loop through rows          */
   outrec = ''
   do col = 1 to sqldavar.sqld    /* loop through columns in a row */
      if  (sqldavar.col.sqlind) = -1  then  /* null field    */   4
         outrec = outrec CENTER('-',sqldavar.col.sqllen)
      else
      outrec = outrec sqldavar.col.sqldata           5
   end /* do */
   call LINEOUT 'TABLE.DAT',outrec      /* write output record     */
   call SQLEXEC 'FETCH c1 USING DESCRIPTOR &:.sqldavar' /* next row*/
end /* do */
call sqlexec 'CLOSE c1';
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine
call SQLEXEC 'COMMIT'
 if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine
call sqlexec 'CONNECT RESET';
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine
return
```

Figure 124 (Part 2 of 2). Varying List SELECT

Notes:

1 The PREPARE statement validates the SELECT statement and prepares it for execution. It names the REXX variable that the SQLDA information will be loaded to.

2 The DESCRIBE statement loads column information into the REXX array variable.

3 The FETCH statement with the USING DESCRIPTOR clause loads the row data into the REXX array variable.

4 The SQLIND field of the SQLDA tells if the field is null.

5 The SQLDATA field of the SQLDA contains the actual data stored in the column for the current row. Loop through columns of a row and build an output record. Note that SQLDA.SQLD contains the number of columns in the row.

9.5.4 Changing Table Data

Adding rows to a table, deleting rows from a table, and updating existing table rows can all be accomplished through REXX programs using SQL statements. To perform these actions on a table, the user must have update authority on the table.

9.5.5 Adding Rows to a Table

In an application that is adding rows to a table, the techniques discussed in the previous section on varying list SELECT can be very helpful. Issuing the PREPARE statement for a SELECT statement using the INTO clause loads the SQLDA column information of a table into a variable. This column information can be useful in edit checking rows before you attempt to add them to a table. The SELECT is never actually executed, but it serves a purpose by allowing the SQLDA information to become available via the PREPARE statement. Edit checking on data types, column length, and null value is possible using the information from the SQLDA. The process for adding rows in this method is:

1. CONNECT to the database.
2. Build a dummy SELECT statement.
3. PREPARE the SELECT statement using the INTO clause.
4. Use SQLDA information to prompt user for input, perform edit checking, etc.
5. Build an INSERT statement.
6. Issue an EXECUTE IMMEDIATE on the INSERT statement.
7. COMMIT the work.

The example in Figure 125 on page 171 shows the usage of the INSERT statement. This example is taken from ADDREC.CMD on the diskette, which shows the preparation of the SELECT statement as well.

```
creator = 'WORKID'
tbname = 'BASEBALL'
fields =  lastname||','||firstname||','||middle  /* column names   */
values = 'MANTLE'||','||'MICKEY'||','||'C'       /* new row values */

st1 = "INSERT INTO " creator||'.'||tbname "("fields") VALUES ("values")"
call SQLEXEC 'EXECUTE IMMEDIATE :st1'
if ( SQLCA.SQLCODE <> 0) then call Error_Handling_routine
```

Figure 125. Adding a Row

9.5.6 Updating Rows

There are several ways to update existing table rows in REXX programs using the SQL UPDATE statement. Updating row by row, using a cursor, is a similar process to the SELECT examples discussed previously. An example of this type of update will be explained later in this section. Mass updates of multiple rows is another way to use the UPDATE statement. That is, certain rows of the table are selected by using a WHERE clause. These rows are then each updated in the same columns with the same information. This update of potentially multiple rows is performed with one UPDATE statement. This is a very powerful use of the UPDATE statement. The example in Figure 126 on page 172 shows how to do an update to change all employees with DEPT '951A' to DEPT '884A'.

Updating rows one at a time, instead of a mass update, may be preferable for some applications. It may be that for each row selected, a user prompt is required to receive changes for that particular row. As discussed in 9.5.5, "Adding Rows to a Table" on page 170, obtaining the SQLDA information can be very useful for getting the row definition and edit checking the user input before the update is attempted, although it is not necessary. The example in Figure 127 on page 172 is an example of updating row by row. It is taken from UPDTREC.CMD on the diskette.

```
/* Mass Update                                                    */
creator='WORKID'
tbname='EMPLOYEE'
old_dept='951A'
new_dept='884A'

call sqlexec 'CONNECT TO ' dbname
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine

st1 = 'UPDATE ' creator||'.'||tbname ' SET DEPT= '||new_dept ,
       ' WHERE DEPT='||old_dept

call sqlexec 'EXECUTE IMMEDIATE &:.st1'
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine

return
```

Figure 126. Mass Update

```
/* Update rows                                                    */
column1 = 'PLAYER'
column2 = 'POSITION'
column3 = 'TEAM'
call sqlexec 'CONNECT TO ' dbname
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
st1 ="SELECT " column1||','||column2 " FROM  WORKID.ROSTER WHERE " ,  1
            column3||"= 'YANKEES'  FOR UPDATE OF " column2

call SQLEXEC 'DECLARE c1 CURSOR FOR s1'
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
call SQLEXEC 'PREPARE s1 FROM &:.st1'
call SQLEXEC 'OPEN c1'
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
```

Figure 127 (Part 1 of 2). Update Row by Row

```
call SQLEXEC 'FETCH c1 INTO &:.var1,&:.var2'              2
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
do while (SQLCA.SQLCODE = 0)    /* loop through rows selected */
  /* display fields, prompt user for change value             */
  say column1 ' = ' var1   ' , ' column2 ' = ' var2
  say 'Enter new value for ' column2
  pull newvalue                                           3

  /* build update statement using value entered by user       */
  updt = 'UPDATE WORKID.ROSTER SET ' column2||"='"||newvalue||"'"|| ' ,
           WHERE CURRENT OF c1'                                   4
  call SQLEXEC 'EXECUTE IMMEDIATE :updt'
  if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
  call SQLEXEC 'FETCH c1 INTO &:.var1,&:.var2'
end /* do */

/* clean up - close cursor and commit work                    */
call sqlexec 'CLOSE c1';
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
call SQLEXEC 'COMMIT'
if ( SQLCA.SQLCODE <> 0) then Call Error_Handling_Routine
return
```

Figure 127 (Part 2 of 2). Update Row by Row

Notes:

1 Selecting PLAYER and POSITION columns from the table only for rows where the team is YANKEES. Column position is updated.

2 FETCH acts on the row that the cursor c1 is pointing to. Currently, c1 is pointing to the first row that satisfied the SELECT conditions. The FETCH loads the value of columns PLAYER and POSITION into variables var1 and var2 respectively.

3 The current values for PLAYER and POSITION for this row are displayed, and the user is prompted to enter a new value for POSITION. This is where using the SQLDA could provide column information useful in edit checking.

4 For current row, POSITION column is updated with user entered value.

9.6 Error Handling

DB2/2 provides information in the SQLCA data structure that is useful in return code checking and error determination. The complete SQLCA data structure is provided in Appendix B, "OS/2 DB2/2 REXX Reference" on page 245. SQLCA.SQCODE is filled with a return code value after every call to the SQLEXEC and SQLDBS APIs. Another source of information is the SQLMSG variable, which contains the description of the SQL return code found in SQLCA.SQLCODE. A way to get a more detailed explanation of the SQL error is to bring up the DB2/2 online documentation. An example of this is provided in Figure 128, which is taken from SQLERR.CMD on the diskette.

```
/* SQLERR                                                          */
arg sqlcode

sqlcode = STRIP(sqlcode,'L','-')                /* remove - sign    */
sqlcode = RIGHT(sqlcode,4,'0')       /* pad with zeros on the left  */
'VIEW DBMSG.INF SQL'||sqlcode      /* bring up DBM information file */

return
```

Figure 128. SQL Error Handling

9.7 Testing Observations

Our testing environment for the sample code found on the diskette, as described at the beginning of this chapter, was two PS/2 machines on the same LAN.

Here are a few observations we made that apply to this environment as well as others:

1. Before accessing a database on a remote workstation, the remote database must have been started. If not you will most likely receive a SQL -30080 return code, which is a timeout condition.
2. If the remote user has the database open in Query Manager, you may be locked out of the database - again a return code of -30080.
3. To allow your REXX code to work on remote databases, you must refer to tables with their full name (owner.tablename).
4. Make sure you have set up all the User Profile Management, database, and table authorizations you need!

9.8 Database Application Remote Interface (DARI)

The DARI is provided by DB2/2 to help improve performance of applications that are accessing remote databases. The remote database must be on a DB2/2 Server machine. The concept is simple. To reduce network traffic, keep your DB2/2 procedures on the server machine, not the client machine. This way the client does not have to send program instructions over the network, since the program already exists on the server. The client just needs to tell the server to run the program. To do this, a small procedure is needed on the client machine to tell the server to start a program. Chapter 6 of the *IBM Database 2 OS/2 Programming Reference* provides examples of client and server procedures and how to invoke them. Additional performance benefits may be realized because only the rows that are actually needed by the client are returned over the network to the client. Also, in most installations, the server machine is equipped with increased memory, disk space, and possibly a faster processor than a client machine.

Chapter 10. Visual REXX Builders

Visual REXX builders in OS/2 are software products that interface with the Presentation Manager (PM), utilizing PM′s Graphical User Interface (GUI) features. Visual REXX builders provide a GUI for the design and development of REXX programs. The real exciting part is that the REXX programs that are developed can take advantage of PM as well. For example, a visual REXX builder product can provide function calls that allow a REXX program to create and manipulate:

- Windows
- Dialog boxes
- Entry fields
- Radio buttons
- Action bars

Visual REXX builders provide a way to create application prototypes quickly. They also provide a way to take existing REXX applications and convert them into applications that take advantage of OS/2′s PM interface. Two of the most popular visual REXX builders on the market are Watcom′s VX-REXX, and Hockware′s VisPro/REXX. In this chapter we take an existing REXX application and convert it into a VX-REXX program, as well as a VisPro/REXX program. Refer to “Related Publications” on page xxii for a listing of VX-REXX and VisPro/REXX books that you may find helpful.

10.1 VisPro/REXX

VisPro/REXX is a visual programming environment for the OS/2 2.1 REXX language. It is completely integrated with the OS/2 2.1 Workplace Shell. VisPro/REXX offers:

- Multiple views including Layout, Event tree and List
- Drag-Drop programming
- Pop-up menus
- Setting notebooks
- Buttons, Lists, Graphics, Sliders and all the CUA ′91 controls
- Business graphics

VisPro/REXX supports:

- APPC, EHLLAPI, DB2/2 and all other external functions that OS/2 REXX supports.
- The OS/2 font and color palettes.

VisPro/REXX can be useful if you, for instance wish to:

- Quickly prototype and develop OS/2 2.1 CUA ′91 applications.
- Generate a small, single .EXE file for license-free distribution.
- Build client/server programs.
- Migrate existing REXX procedures to the OS/2 2.1 GUI environment.

System requirements for VisPro/REXX are:

- OS/2 2.1 with at least 5MB memory
- At least 2MB of free hard disk space

The Refresh Release V1.1 shipped 26th August 1993 adds support for:

- Easy multithreading - the user interface is separate from the REXX code.
- STDIO window - command output/input by a separate window.
- Easy command interface - just enclose external commands in quotes.

VisPro/REXX ships in two versions, the full version and the BRONZE edition. The BRONZE version differs from the current release in that it lacks:

- Container control
- Slider control
- Notebook control
- Business graphics

An evaluation package is also available from HockWare. This package includes a demo of VisPro/REXX and the evaluation copy. The evaluation copy has some limitations on the size of the application you can create, and it does not allow you to create an .EXE file.

10.2 VX-REXX

Watcom's VX-REXX is an easy to use development environment for creating applications that leverage the capabilities of OS/2 2.1 and exploit the Presentation Manager Graphical User Interface. The following are some of the features that are a part of VX-REXX:

- Project management
- Graphic design of PM objects
- Customization of object properties
- Drag and drop objects to create source code
- Support for APPC, EHLLAPI, DB2/2 and all other external functions that OS/2 REXX 2. supports
- Source level debugger
- No run time licensing fees
- Creation of executable files

10.3 Example - SELECT.CMD

SELECT.CMD is a REXX program that uses DB2/2 APIs to select data from a table. Subroutines GETDB.CMD and GETTABLE.CMD are called from SELECT.CMD. All three of these programs are explained in detail in Chapter 9, "REXX Interfaces to DB2/2" on page 151. It uses STDIN and STDOUT to communicate with the user. It displays all existing databases and prompts the user to select one. It then displays all tables for the database selected and prompts the user to select one. It then loads all rows of the table selected into a file and opens an E editor session on the file. We took this program and converted it into a visual REXX program using VX-REXX. We did the same using VisPro/REXX. This example should give you an idea of how much more effective and easy to use a GUI is versus STDIN and STDOUT from a user's perspective. It also will provide you with examples of how to create visual REXX programs containing basic GUI features in both VisPro/REXX and VX-REXX.

Visual REXX programming is more of an event-driven, object-oriented environment than traditional top down, procedural programming. There are a number of different ways to visualize even a small application like this one. You need to think about what makes the most sense from a user perspective and a programming perspective.

Before jumping into writing the application, it is wise to spend some time planning how the GUI will look and operate. For the SELECT.CMD program we decided to have a main window with a list box, listing all available databases. Once the user selects a database, a second window, our Table window, appears. This window contains a list box, listing all tables for the selected database. Once the user selects a table, the query is run and an E editor session is opened containing the query results. When the user wishes to leave the Table window, they can click on the Cancel push button. To leave the application, they can select the Exit option from the menu bar on the primary window.

There are other details which need to be thought about. For example, will the secondary window be modal, modeless, or system modal? We chose modal, which means that other windows in the application cannot be activated while the Table window is active. System modal means that no other window or icon on the desktop can be activated while the Table window is active. Modeless in this situation means that the user can activate other windows while the Table window is active. These are ways to control user activity. You need to think about what potential problems can be caused by user actions. There are other options to consider that we will look at as we go through the examples.

10.4 SELECT.CMD with VisPro/REXX

The following example is a step by step approach for converting the SELECT.CMD standard REXX application into a VisPro/REXX application. The level of VisPro/REXX used is Release 1.1.

10.4.1 Initial Setup

To open a VisPro/REXX session, double click on the **HockWare VisPro/REXX** icon to open the Projects folder. By default the Projects folder has the following objects:

- Project
- Samples
- Color Palette
- Font Palette
- REXX Information
- Tutorial/Demo

To start creating SELECT.CMD with VisPro/REXX, drag a project from the Project template icon using mouse button 2. The fastest way to change the name of the new project is to press down the Alt-key and click on mouse button 1 and write the new name: SELECT.

To open the project double click on the new project and you have a new folder with three icons in it:

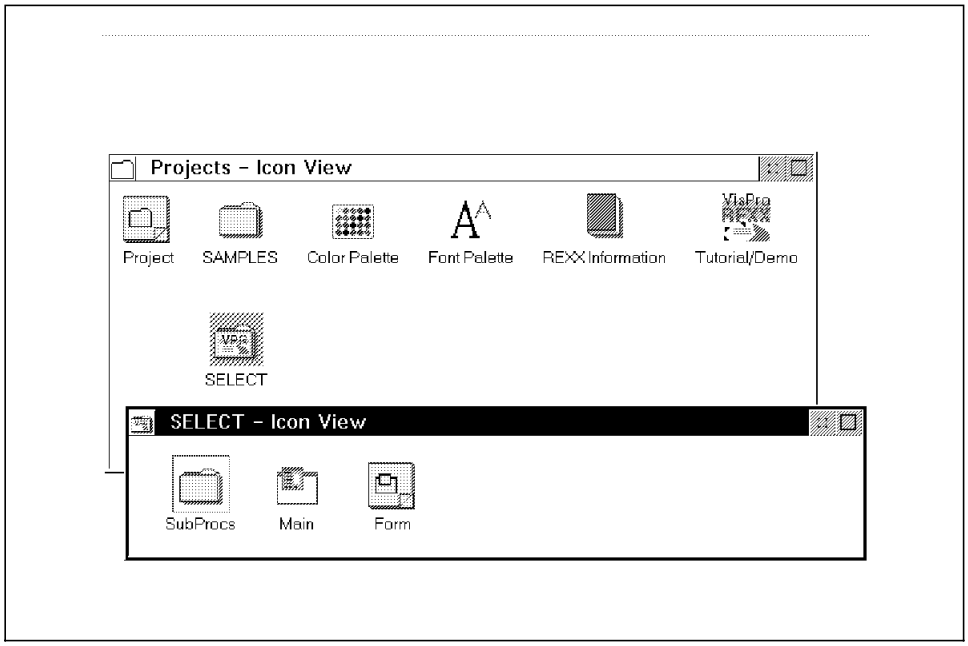


Figure 129. Project Folder in VisPro/REXX

The **Main** icon contains the main form of the new project, the **Form** icon can be used to create additional windows for the program and the **SubProcs** can be used for general routines for the program.

10.4.2 The Main Form

To start creating the program first double click on **Main**. If you are starting a VisPro/REXX project for the first time, you are prompted for your name and the serial number of your VisPro/REXX copy. You now get the Layout view of the main form with icons representing the objects that can be used for visually creating a window.

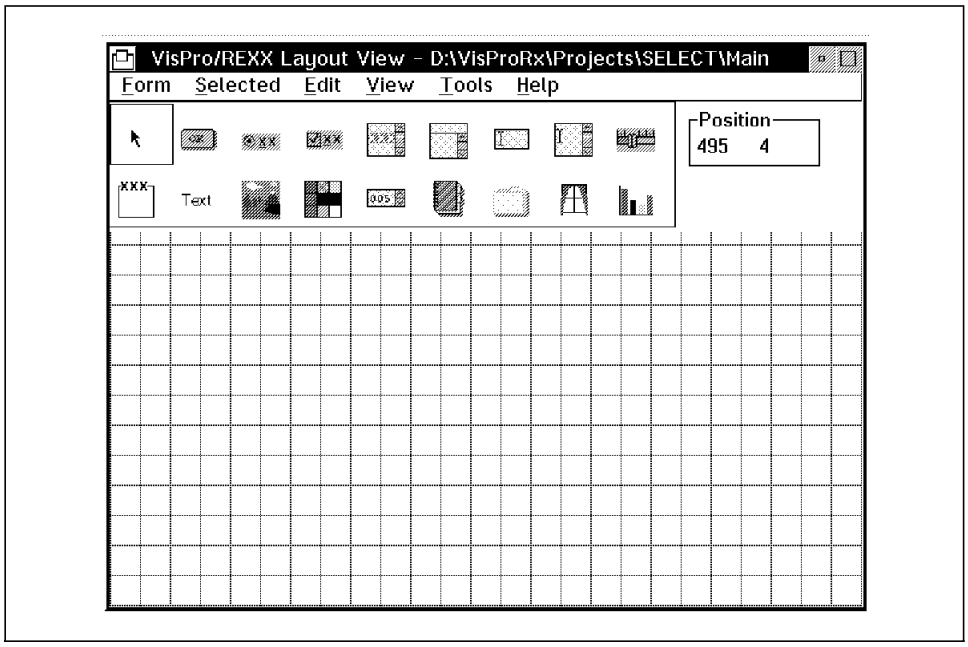


Figure 130. Layout View in VisPro/REXX

The main form also has a menu bar with items used to create, test and customize the project. The menu bar items include:

- **Form**
  - **Open** for opening the settings for the form or the views to the form.
  - **When** for creating events associated wit opening or closing the form or a subform.
  - **Test** for testing the program.
  - **Build** for building a standalone .EXE file.
  - **Save** for saving the project.
- **Selected**
  - **Open Settings...** for opening the settings of selected objects. You must first select objects with mouse button 1.
  - **When** for creating events associated with objects.
  - **Arrange** for arranging objects.
- **Edit** for cut and paste operations.
- **View** for different views of the form.

- **Tools** for dragging objects to the form.
- **Help** for online help.

Objects can be dragged to a window by first selecting them with mouse button 1 and then positioning the pointer where you want the object and pressing mouse button 1 again. To resize the objects they must first be selected with mouse button 1 and then you can use mouse button 2 to resize them. The properties of an object can then be changed by pressing mouse button 2 inside the object and selecting **Open Settings**. This gives you a notebook of the changeable properties unique to each object. The objects in VisPro/REXX are:

BusinessGraphic	An object for creating business graphics. .
CheckBox	A box that the user can toggle on or off by clicking the pointing device or using the keyboard.
ComboBox	A combination of the EntryField and ListBox objects. Text can be entered in to the entry field. If the user selects an item from the list box, it is copied into the entry field.
DescriptiveText	Used to display text.
EntryField	Provides method for user to input data.
Container	A container for objects.
Graphic	A method to display images, such as bitmaps, pointers and icons.
GroupBox	A container for other objects.
ListBox	List of items from which the user can make a selection.
MultiLineEntryField	Provides method for user to input multiple lines of data.
Notebook	An object to create notebooks.
Plain Window	Provides a free form window, where you can for instance capture mouse and keyboard events.
PushButton	A button that the user can select by clicking the pointing device or using the keyboard.
RadioButton	A button usually found in a group that the user can select by clicking the pointing device or using the keyboard. Only one radio button in the group can be selected at a time.

Spin button	A scrollable ring of choices from which the user can make a selection.
Slider	A slider to create for instance progress indicators. .
Valueset	A set of graphics or text objects. .

10.4.3 Main Window Layout

In the main form for SELECT.CMD you need three objects: a list box for displaying the available databases, a push button for selecting a database, and a text field.

1. Select the list box with mouse button 1 and position the mouse pointer to where you want the lower-left corner of the list box.
2. Press mouse button 2 inside the list box and select **Open Settings...**.
3. Deselect **Horizontal scroll bar**.
4. Close settings.
5. Resize the list box using mouse button 2.
6. Select and position a text field just above the list box.
7. Change the caption of the text field to ″Select a Database″ by holding down the Alt key and pressing mouse button 1, and then edit the text.
8. Select and position the push button on the right side next to the list box and change the caption to ″~ Press to Select″ by using Alt and mouse button 1.

Now we have the general layout for the main form:

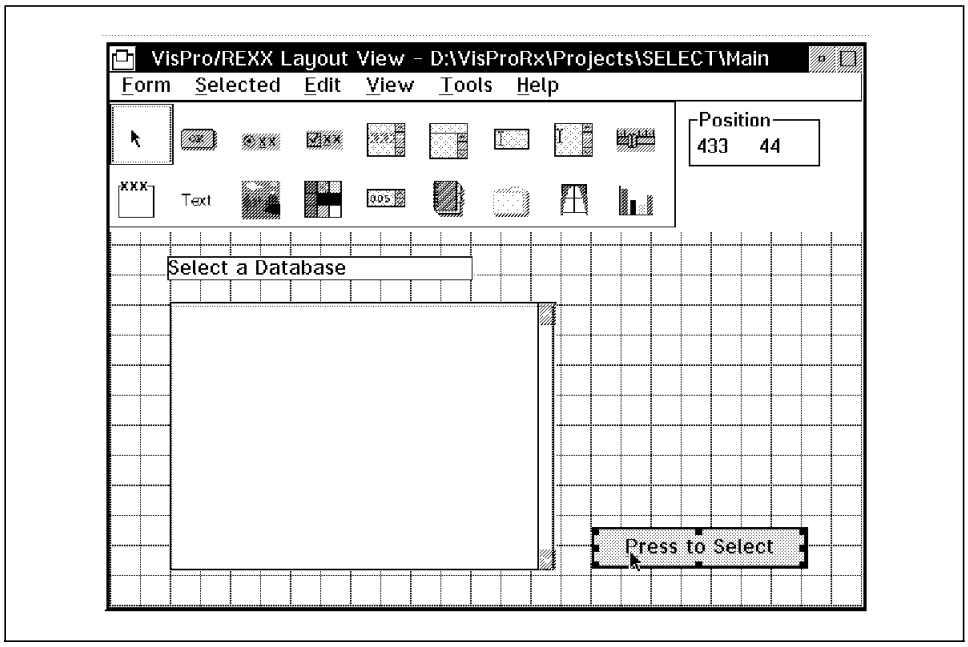


Figure 131. Main Form Layout in VisPro/REXX

There are some additional items that we want to change. For instance the default style for the form is plain which means that the form is not sizeable. To change this click on **Form** then click on **Open** and **Settings**. This gives you the settings notebook for the main form:

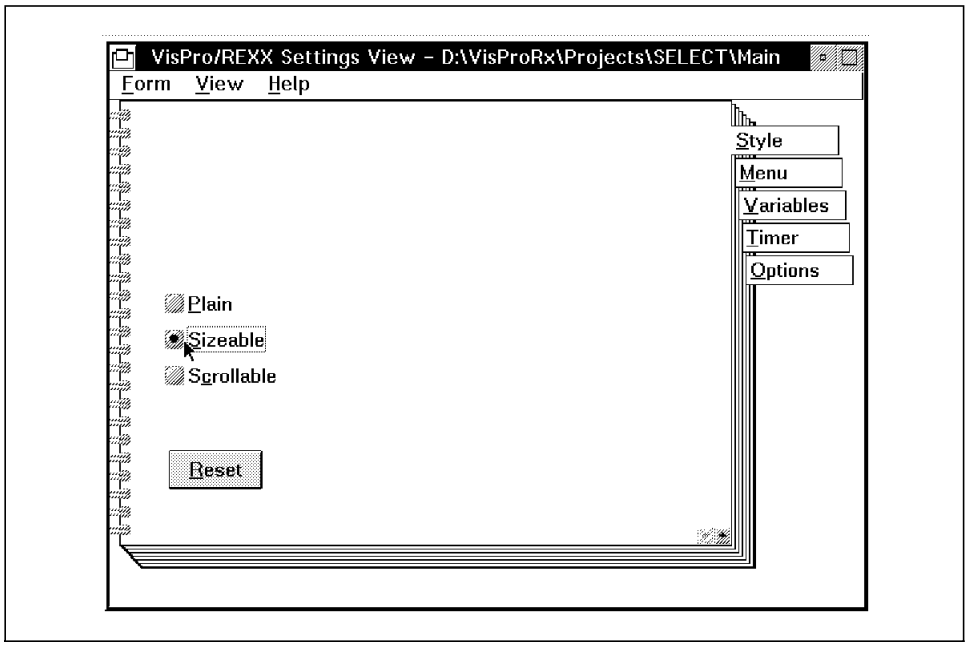


Figure 132. Form Settings Notebook in VisPro/REXX

The settings notebook allows you to change the appearance of the form. The **Style** page allows you to change the form to plain, sizeable or scrollable. Press the **Sizeable** radio button to change the form to sizeable.

10.4.4 Adding a Menu Bar

It is also possible to create a menu bar through the **Menu** page of the settings notebook. To add a menu bar to the form click on the **Menu** page and click on **Add Menu**. A new submenu item is now added to the Menu Bar Designer. Click on the submenu icon and change the caption of the submenu to ~ **File** using Alt-mouse button 1. The ~ character before File provides an accelerator function to the menu item and the character after the ~ is underscored. The menu is then accessible by pressing **Alt-F**. Now click on the menu icon on the left side of the text to apply the changed caption and to bring focus on the **File** menu item. Then add an item to the menu by pressing **Add item**. Click on the new menu item and change its caption to **E**~ **xit** using Alt-mouse button 1. Click on the menu icon on the left side of the text to apply the changed caption and to bring focus on the **Exit** menu item. Your Menu Bar Designer should now look like Figure 133 on page 187.

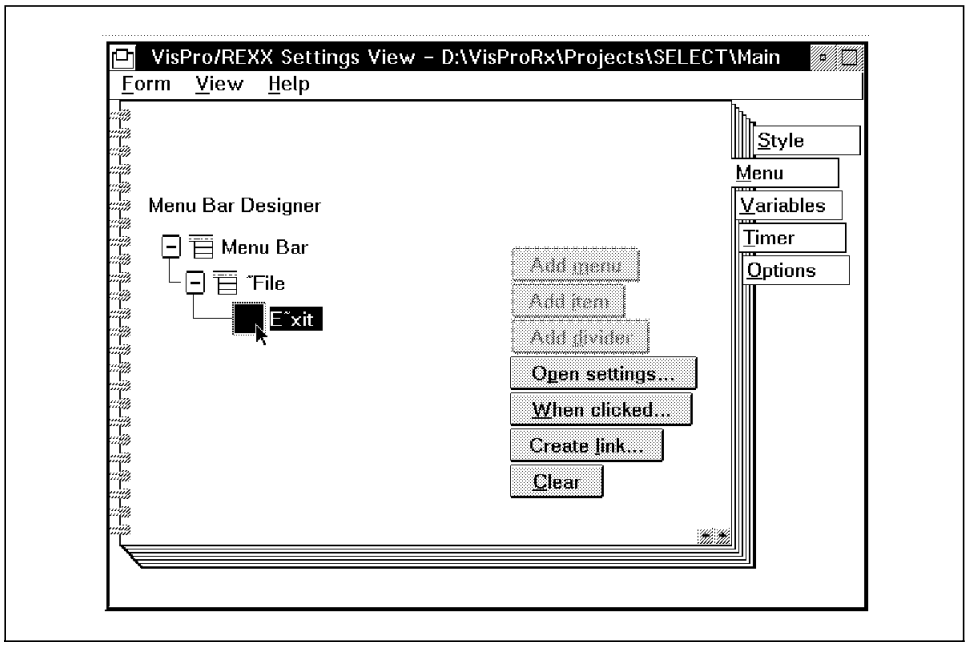


Figure 133. Menu Bar Designer in VisPro/REXX

To add an event to the **Exit** menu item press **When clicked**. You now get the code window for the Exit event as shown in Figure 134 on page 188. The code window allows you to write code associated with the Exit event. There is a preset line of code: **Arg window self** at the beginning of the code window. Do not remove this code for it gives the window handle and the event handle to your code.

In VisPro/REXX there are several ways to code your application. You can either write the code yourself, add code from the **Add** menu or drag objects into the code window and then select the code from a selection list. In this case we use the Add menu to add code to close the window. Click mouse button 2 inside the code window to get the pop up menu for the code window. Click on **Add** to get the Add menu. Choose **Window management** and **Close window**. VisPro/REXX now adds the following line of code to the code window:

```
CALL VpWindow window,'CLOSE'
```

Your code window should look like Figure 134 on page 188.

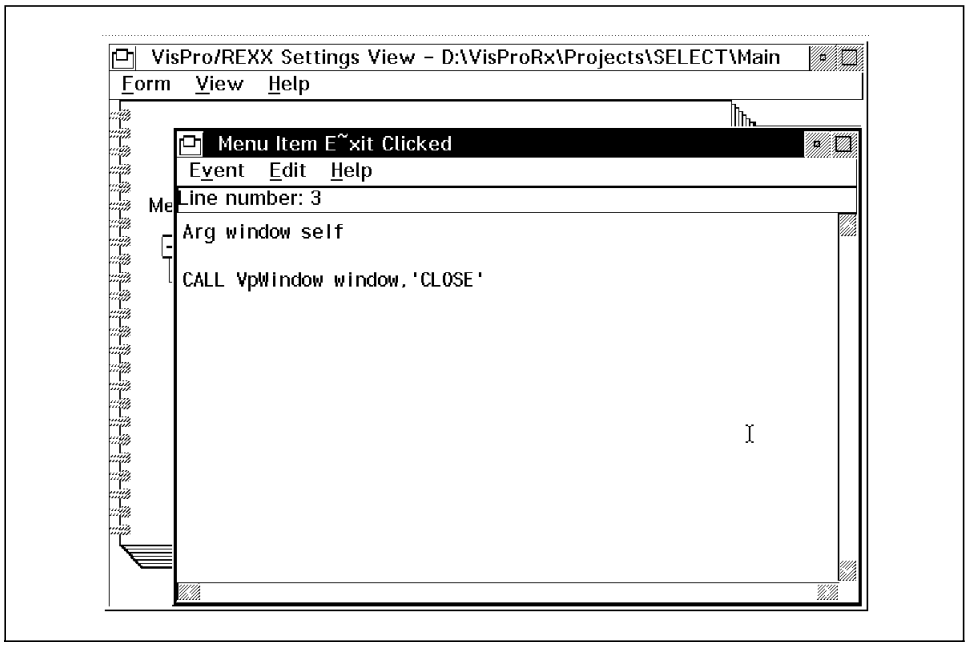


Figure 134. Code Window in VisPro/REXX

Close the code window. Close Settings notebook. That's all you need; choosing **Exit** from the menu now closes the program.

If you want to test the program, for example, to see what the form looks like and how your events work you can do it at any time by selecting **Test** from the **Form** menu. We now have the layout of the main form and one event but to make the program useful we need to add the code from the SELECT.CMD to the program.

10.4.5 Copying REXX Code

To insert the code to load all available databases to the listbox when the program starts we use the **When Opened** event.

1. Select **Form**
2. Select **When**
3. Select **Opened**

You now get the code window for the event that is associated with starting the program.

In the code window first change the title bar of the form when it is opened. To do this:

1. Place the cursor on the next line after **Arg window**
2. Press **mouse button 2**
3. Select **Add**
4. Select **Window management**
5. Select **Set window title**
6. Change **value** to *"***Available Databases***"*

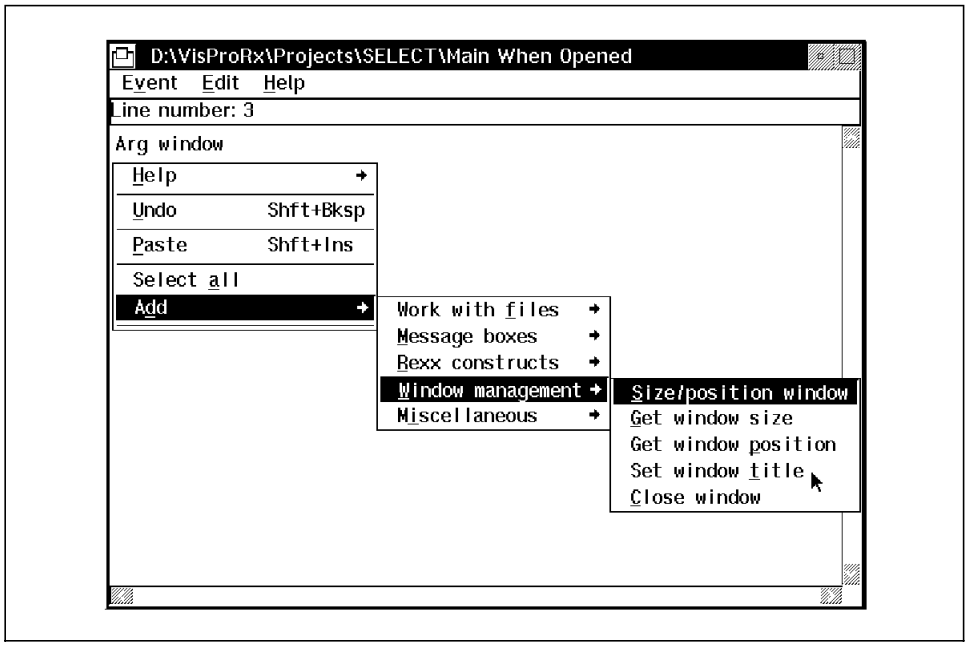


Figure 135. Add Window Management in Code Window in VisPro/REXX

SELECT.CMD calls GETDB.CMD to list all available (cataloged) databases. This is the code we want to use in the main form to display the databases in a list box when the form is opened. Edit GETDB.CMD with an editor and select all lines and copy them to clipboard. To do this with E-editor:

1. Select **Edit**
2. Select **Select all**
3. Select **Edit**
4. Select **Copy**

Now go back to the **When opened** event and select **Edit - Paste** or mouse button 2 and Paste. The code from GETDB.CMD has to be changed a little bit to work in a graphical frontend. For instance VisPro/REXX 1.0 has no input/output console so the **Say**, **Pull** and **CLS** instructions do not work. Even though Version 1.1 has a console it is not the correct way to do screen I/O in a graphical front-end. The next step is to provide the message output in message boxes.

In the beginning of GETDB.CMD there are two parts to register the SQLDBS and SQLEXEC functions. If the registering is not successful then an error message is displayed using say:

```
if rxfuncquery('SQLDBS') <> 0 then do
  rcy = rxfuncadd( 'SQLDBS', 'SQLAR', 'SQLDBS' );
  if RCY \= 0 then do
    say "Error registering SQLDBS: rc = " rcy
    return
  end
end
```

In a graphical interface we can instead show a message box showing the error message:

1. Insert a line just before the **say** instruction.
2. Position the cursor on the empty line and press mouse button 2.
3. Select **Add**.
4. Select **Message boxes**.
5. Select **Plain**.

You now get the following line of code:

```
response=VpMessageBox(window,'title','message')
```

1. Replace the 'title' text with a title for the message box, in this case 'Error'.
2. Replace the 'message' text with a message text for the message box, in this case 'Error registering SQLDBS: rc = ' rc .
3. Delete the line with the **Say** instruction.

Repeat the above steps for the **SQLEXEC** registration part.

The following code lists all cataloged databases on your system.

```
/* loop through list of databases and display them
do i=1 to scan_db.2       /* scan_db.2 contains number of databases
   call SQLDBS 'GET DATABASE DIRECTORY ENTRY :scan_db.1 USING :entry'
   say '     '     entry.2
   say
end   /* end do loop */
```

Now we want to display the databases in a list box instead of using **Say** to display them. This can be done using drag and drop programming.

10.4.6 Drag and Drop Programming

1. Insert a line just before the **Say** instruction.
2. Go to the main form and select **View - Event tree view**.
3. Position the Event Tree View and When Opened windows so that you can drag the **List Box** icon from the Event Tree View with mouse button 2 to inside the code in When Opened as shown in Figure 136.

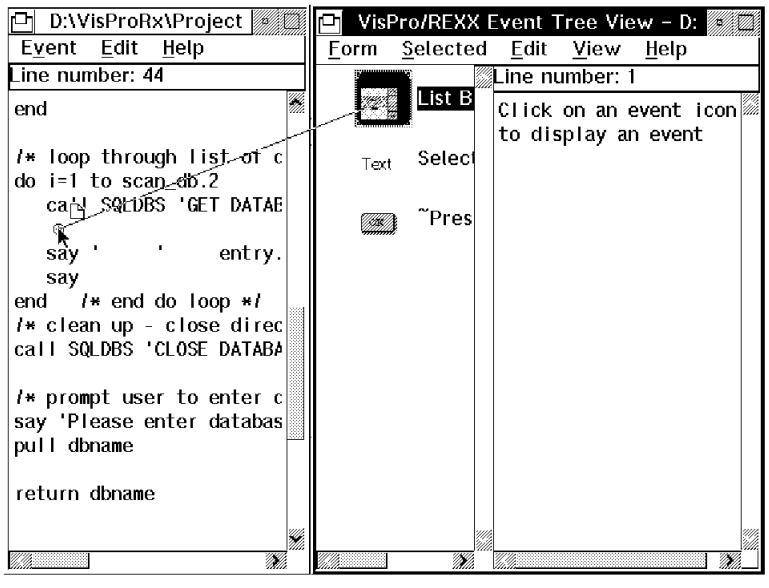


Figure 136. Drag and Drop Programming in VisPro/REXX

When mouse button 2 is released you get a **Create Link** selection list containing all activities you can perform regarding the object that is being dragged as shown in Figure 137 on page 192.

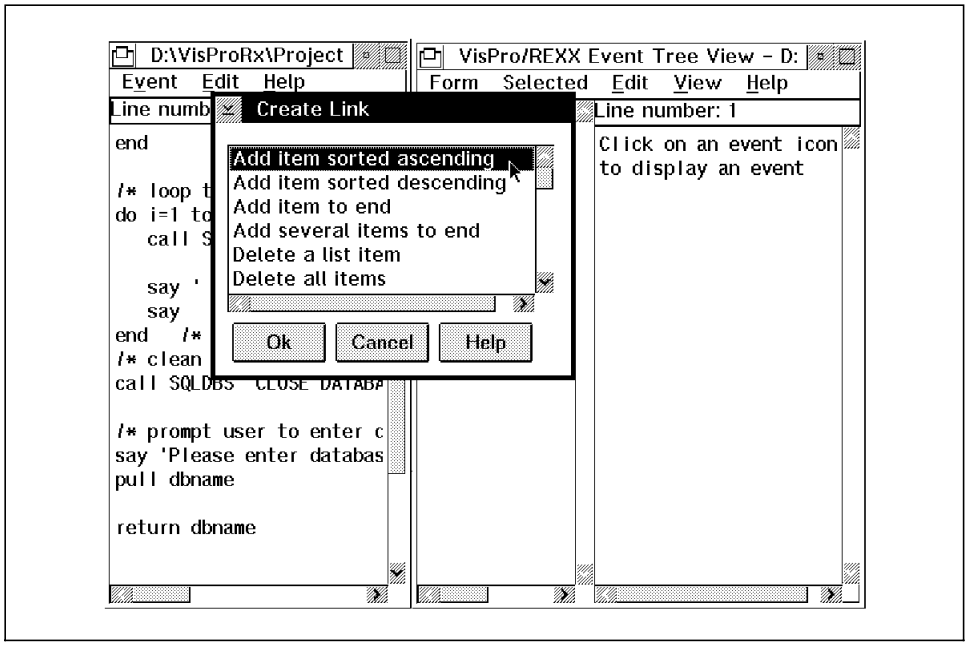


Figure 137. Create Link in VisPro/REXX

4. Select **Add item sorted ascending** which will produce the following line of code in the program:

```
/* Add item sorted ascending  List Box */
CALL VpAddItem window,1000,'ASCENDING',value
```

5. Change **value** to **entry.2** which is the variable containing the database name.
6. Delete all the other **Say** and **Pull** instructions and the **CLS** instruction from the code because all other messages are handled through the SQLERR.CMD routine.
7. Close the code window.

Now you have a program that lists all cataloged databases in a list box. The next stage is to enable selecting a database from the list box and listing all its tables.

10.4.7 Creating a Secondary Form

A secondary form can be created in the Project icon view for the SELECT project by dragging a new form from the **Form** icon. The form must be dragged into the same folder as the main form. Then do the following:

1. Rename the subform to **Tables** using Alt - mouse button 1.
2. Open the Tables form by double clicking on it. Add a list box, a push button and a text field to it like in the main form.
3. Change the caption of the push button to ~ **Cancel** using Alt - mouse button 1.
4. Add the close window event to the pushbutton by pointing the mouse pointer at it and selecting: **mouse button 2 - When - Clicked/selected** and then **mouse button 2 - Add - Window management - Close window**.
5. Delete the caption of the text field using Alt - Mouse button 1. The caption will be given a value in the **When Opened** event described in 10.4.9, “List Tables” on page 196.

The secondary form should now look like Figure 138.

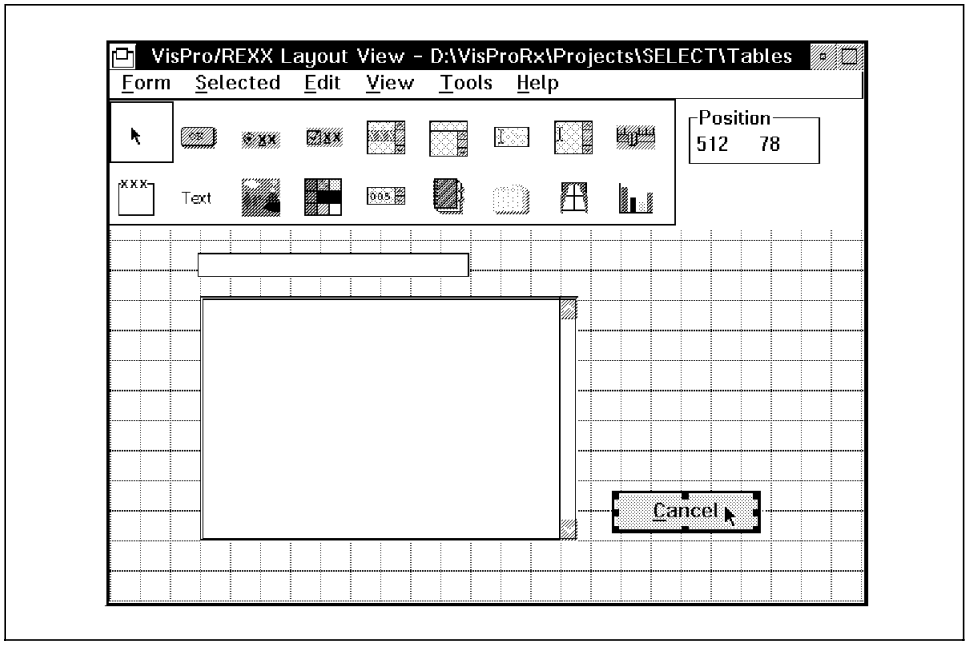


Figure 138. Tables Form in VisPro/REXX

Leave the Tables form open and go back to the main form to create the events for opening the subform.

10.4.8 Creating Events

To create an event for displaying the Tables form when the **Press to Select** push button is pressed do the following:

1. Position the mouse pointer inside the push button.
2. Press mouse button 2.
3. Select **When**.
4. Select **Clicked/selected**.

You now get the code window for the event associated with pressing the push button. When pressing the push button you want to know what the selected value in the list box is. To do this drag the list box into the code window using mouse button 2. Select **Get item value**. You now get the following code:

```
value = VpGetItemValue(window,1000)
```

Change **value** to **dbname** as we will later use this variable to list tables in GETTABLE.CMD. Open the Tables form and then select **Form - Create link** from the menu bar. Then go back to the **Clicked/selected** event for the push button in the **Main** form and use paste. This will create a link from the push button to the secondary form so that when the push button is clicked the secondary form is opened. VisPro/REXX now inserts the following code:

```
value=VpOpenForm(window, 257, 'topic name')
```

Change '**topic name**' to **dbname** without quotes. Topic is a special variable in VisPro/REXX which can be used to pass arguments between forms. The **Topic** variable is also globally known to all events in the form.

Note

A VisPro/REXX program can also receive parameters through use of the Topic variable in the **Main** form.

To ensure that something is selected add **If dbname <> '' then** into the code. Then:

1. Insert a line just after the **VpOpenForm** instruction.
2. Position the cursor on the empty line and press mouse button 2.
3. Select **Add**.
4. Select **Message boxes**.
5. Select **Plain**.
6. Replace the 'title' text with a title for the message box, in this case 'Error'.
7. Replace the 'message' text with a message text for the message box, in this case 'Highlight Database before Selecting.'

The push button click event should now contain the following code:

```
 scale="0.9".
/* Get item value  List Box */
dbname = VpGetItemValue(window,1000)
/* Open the form D:\VisProRx\Projects\Project!5\Form!1*/
If dbname <> ''  then
  value=VpOpenForm(window, 257, dbname)
else
  response=VpMessageBox(window,'Error','Highlight Database Before Selecting')
```

Copy this code to the clipboard and:

1. Close the code window.
2. Position the mouse pointer inside the list box in the **main** form.
3. Press **mouse button 2**.
4. Select **When**.
5. Select **Mouse button 1 double click**.
6. Paste the code from the push button.

Now the same code in the **Main** form will be executed both when the push button is pressed and when a line is double clicked in the list box.

10.4.9 List Tables

To list the tables belonging to a database go to the secondary form: Tables.

1. Select **Form**.
2. Select **When**.
3. Select **Opened**.

Now you are in the code view of the event that is executed when the Tables form is opened. First change the title bar of the form:

1. Place the cursor on the next line after Arg window.
2. Press **mouse button 2**.
3. Select **Add**.
4. Select **Window management**.
5. Select **Set window title**.
6. Change **value** to '**Select a Table to Query**'.

Then add **dbname = Topic** into the code to get the dbname that was passed as an argument to the form.

Note

All events in VisPro/REXX are handled separately so variables are unique to events. Therefore you have to pass values either by using the Topic special variable or setting variables as global variables in the Variables page of the form settings. Variables that are set as global in the main form are global to all events and forms. Variables that are set as global on subforms are global to the events in that form only.

Next set the caption of the text field:

1. Drag the text field into the code window using mouse button 2.
2. Select **Set item value**.
3. Change value to **dbname** '**Tables**'.

Next copy all the lines from GETTABLE.CMD using an editor and paste them into the code window.

10.4.10 GETTABLE.CMD

GETTABLE.CMD takes dbname as an argument but since we already got the dbname from the Topic variable the **parse arg dbname** line must be removed. To fill the list box with table names do the following:

1. Insert a new line and position the cursor after the following line in the code:

```
say 'Table = ' STRIP(creator_name)||'.'||STRIP(table_name)
```

2. Change the line so it will read:

```
Table = STRIP(creator_name)||'.'||STRIP(table_name)
```

3. Drag the list box from the **Tables** form into the empty line using mouse button 2 and release the button.
4. Select **Add item sorted ascending**.
5. Change **value** to **Table**.

Now remove all **Say**, **Pull** and **CLS** statements from the code. The return values can also be removed since the program does not return anything. Instead of **Return "ERROR"** it should just have **Return** for the sake of clarity.

10.4.11 SubProcs - SQLERR.CMD

SQLERR.CMD is used for handling error codes from various DB2/2 commands and is called by many events in this program. To make this code available to all the events it must be made a subroutine. This can be done by copying SQLERR.CMD to the SubProcs folder located in the Project folder for SELECT project. The copying can be done for instance by opening the directory containing SQLERR.CMD through the Drives icon for the drive containing the file and dragging the .CMD file to the SubProcs folder with mouse button 2 while holding down the Ctrl key.

Note

- When adding subroutines to a SubProcs folder, make certain that the file name does not have an extension and is in uppercase.
- All subroutines within the SubProcs folder will be included as part of the resulting .EXE file when doing a build.

10.4.12 Show Table Rows

The last part of the program takes the table name from the list box in the Tables form, selects all lines in the table and displays them using E - editor. For this we first need the selected value from the table:

1. Position the mouse pointer inside the list box in the **Tables** form.
2. Press **mouse button 2**.
3. Select **When**.
4. Select **Mouse button 1 double click**.
5. Drag the list box into the code with mouse button 2 and release button.
6. Select **Get item value** from the **Create link** list.
7. Change value to creator_table
8. Add **dbname = Topic** to the code.
9. Copy and paste lines from SELECT.CMD from the following statement to the end of the file:

```
call sqlexec 'CONNECT RESET';
```

10. Remove all **Say** statements.

10.4.13 Build the Application

The application is now ready to be built into a stand-alone .EXE file. First test the application using **Form - Test**. If everything works use **Form - Build** to build the .EXE file.

Note

When VisPro/REXX builds the application it does not give a message saying when the build process is finished. You just have to wait until disk activity stops to ensure that the build process is finished. Trying to run the RUN.EXE file before the build process is finished will result in an error message and may cause some problems later.

VisPro/REXX always names the .EXE file to RUN.EXE, which can then be renamed to anything you want.

10.4.14 Tip on Adding an Icon to the .EXE file

VisPro/REXX has no menu-driven interface to include an .ICO file along with the .EXE file. If you wish to include an ICON with the application and still be able to distribute just one file, do the following:

1. Build your .EXE as normal.
2. Edit DELETE.ME1 and add a line at the end like this:

```
ICON 1 "X:\\MYDIR\\MY.ICO".
```

3. Make sure there is a carriage return after the above line.
4. Get an OS/2 window up, and go to your project directory, that is:

```
cd C:\VISPRORX\PROJECTS\MYPROJECT.
```

5. Type:

```
RC -R DELETE.ME1.
```

6. When this finishes successfully, type the following:

```
COPY C:\VISPRORX\TEMPLATE.EXE RUN.EXE.
```

7. Then type:

```
RC DELETE.RES RUN.EXE.
```

You now have a default icon for your EXE!

10.5 SELECT.CMD with VX-REXX

The following example is a step-by-step approach for converting the SELECT.CMD standard REXX application into a VX-REXX application. The level of VX-REXX used is Release 1.01.

10.5.1 Initial Setup

To open a VX-REXX session, double click on the **WATCOM VX-REXX** icon. Then double click on the **VX-REXX** icon. This opens a new VX-REXX project environment. The tool palette on the right contains icons that represent different types of Presentation Manager objects. These can be used to customize windows. Here is a description of the objects represented in the tool palette:

CheckBox A box that the user can toggle on or off by clicking the pointing device or using the keyboard.

ComboBox A combination of the EntryField and ListBox objects. Text can be entered into the entry field. If the user selects an item from the list box, it is copied into the entry field.

DescriptiveText Used to display text.

DropDownComboBox A combination of the EntryField and ListBox objects. Text can be entered into the entry field. When the arrow button is selected, the list box appears. If the user selects an item from the list box, it is copied into the entry field.

EntryField Provides method for user to input data.

GroupBox A container for other objects.

ImagePushButton A PushButton object that can have a bitmap or icon as the face of the button.

ImageRadioButton	A RadioButton object that can have a bitmap or icon as the face of the button.
ListBox	List of items from which the user can make a selection.
MultiLineEntryField	Provides method for user to input multiple lines of data.
PictureBox	A method to display images, such as bitmaps and metafiles.
PushButton	A button that the user can select by clicking the pointing device or using the keyboard.
RadioButton	A button usually found in a group that the user can select by clicking the pointing device or using the keyboard. Only one radio button in the group can be selected at a time.
Spin button	A scrollable ring of choices from which the user can make a selection.

The Window1 object has already been created, since at least one window will be needed for any visual REXX application. Here are a couple of quick pointers about the VX-REXX menu bar:

- Use **Project** for saving projects, opening new projects, and creating .exe files out of projects.
- Use **Windows** for a list of all windows in the application (**Window list**) and a list of all REXX subroutines in the application (**Section list**).
- Use **Run** to test your application.

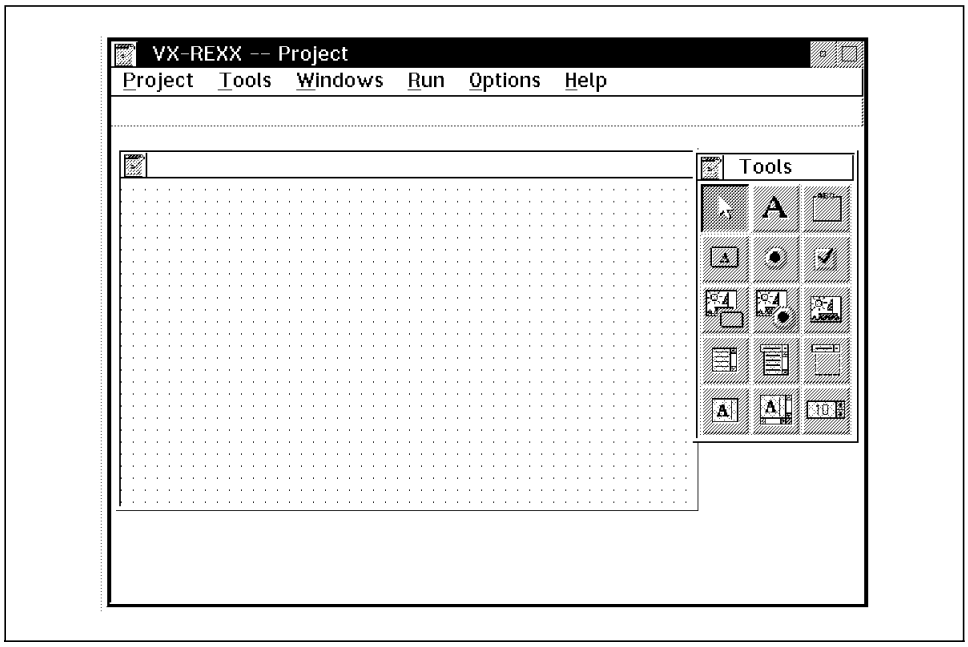


Figure 139. VX-REXX Initial Screen

10.5.2 Primary Window Setup (Window1)

Object Window1 is the first thing displayed when the SELECT application is executed. We need to customize this window in a few ways. First, we can put a window title in the title bar:

1. With the mouse pointer inside Window1, click on mousebutton 2.
2. Click on **Properties**. These are the settings for Window1. There are many customizations that can be made here, from sizing the window to chaging background color.
3. To update the title bar with the name of our application, bring up the **Text** page by clicking on **Text**. In the **Caption** field, enter the following string:

   ```
   Available Databases
   ```

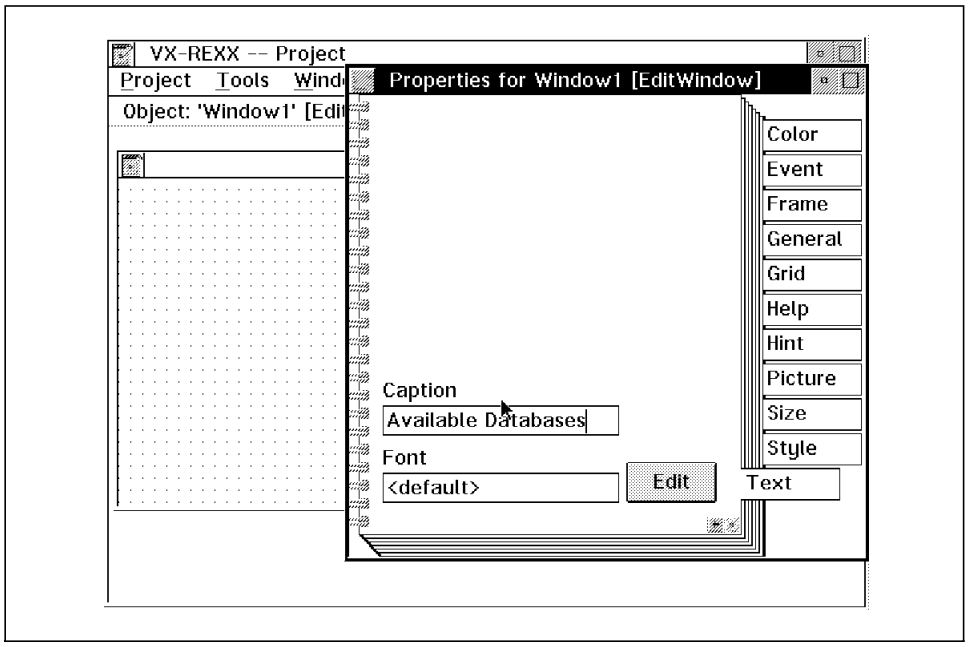


Figure 140. Text Page of Window1 Properties

4. While in the Properties window, we can provide maximize and minimize buttons for the window, along with a system menu by selecting those options on the **Frame** page.

5. Close the **Properties** window. Notice the changes we just made are now reflected in Window1.

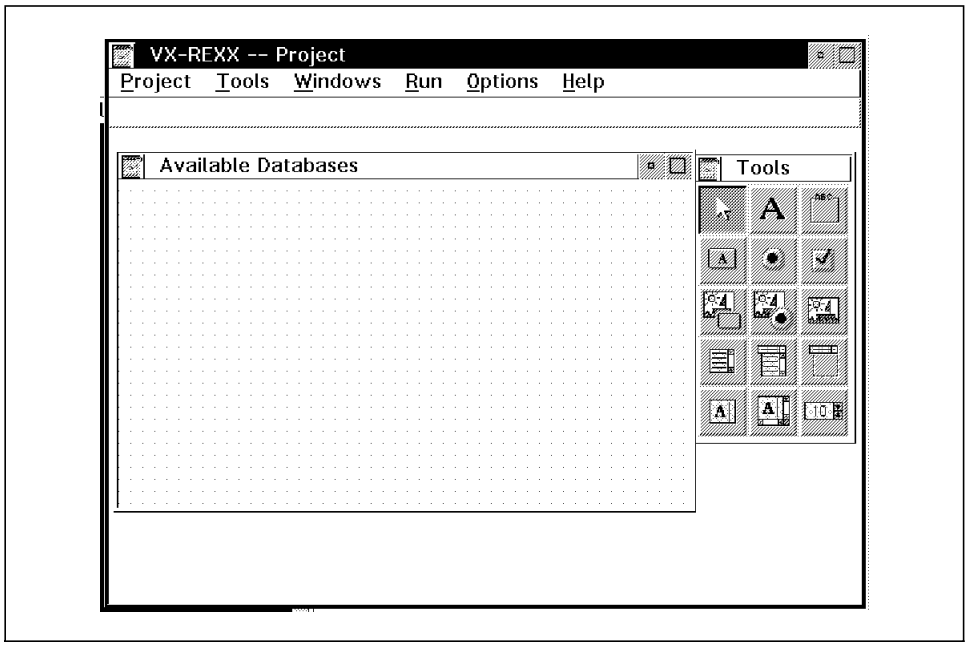


Figure 141. Window1

Next we need to create a menu bar for Window1. To do this:

1. With the mouse pointer inside Window1, click on mousebutton 2.
2. Click on **Menu editor...**. This is where you can create menu bar items.
3. Create a level 1 menu item called File by typing the word File in the **Caption** entry field.
4. Click on **Insert**. The variable name Menu1 is generated for you. Although you can change this variable name, we will leave it as is for this example. The menu item File is highlighted in the list box on the right.
5. Now we need to create a level 2 menu item called Close file. In the list box, aim the mouse pointer at the area just under the File item.
6. Click on mousebutton 1 to highlight the area just below File.
7. Use the **Level** spin button to make sure the **Level** field is 2.
8. Type the words Close file in the **Caption** entry field.
9. Click on **Insert**. The variable name Menu2 is generated for you.
10. The up and down arrow selections on the editor are used to position the menu items if necessary.

Accelerator keys are used as an alternative way to choose a menu item. For example, type the letter C in the **Accelerator** entry field. Now click on **Change** to have this change relected. Now a user can choose the Close file option by selecting it with the mouse or by typing the accelerator key C.

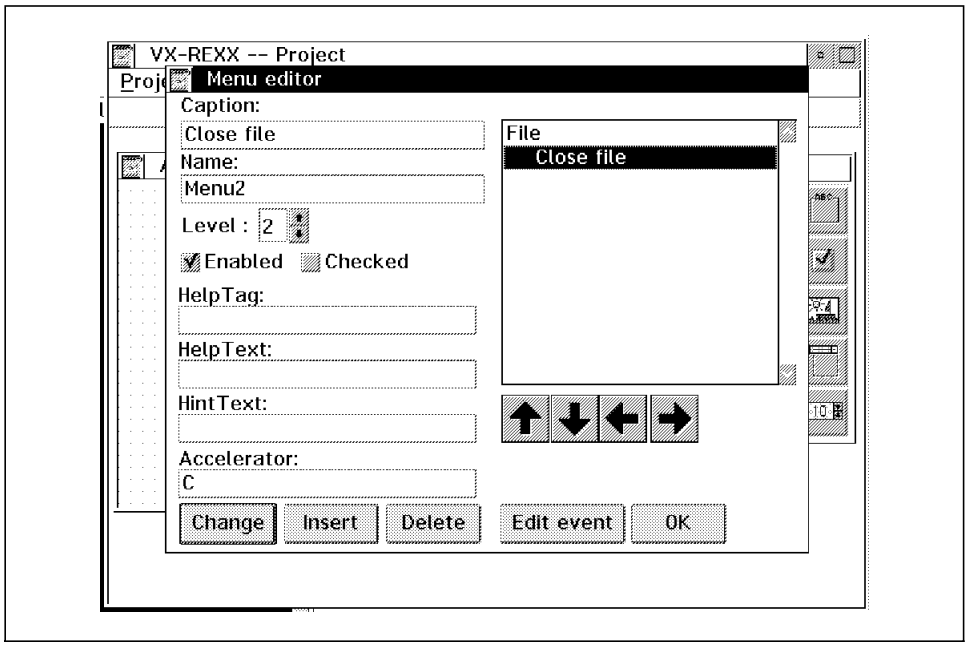


Figure 142. Menu Editor

When the user selects Close file, we want the program to end. To put some REXX code behind this menu item, click on **Edit event** with the Close file menu item highlighted. This brings up an edit session for subroutine Menu2_Click. Whatever code is placed here will execute whenever a user clicks on Close file. Since we want the program to end, type:

```
call quit
```

This will end the program. How did we know that? We read the manual. Quit is a VX-REXX generated subroutine that ends the application. Close the edit session. We are now done with the menu editor. Click on **OK** to close the menu editor window.

Next we need to create a ListBox object. To do this:

1. Select the ListBox object type from the tool palette by pointing to it and clicking on mousebutton 1. If you are not sure which icon represents the ListBox object, remember that a description of what the mouse is

pointing to is located on the bar just below the VX-REXX menu bar. You can also refer to 10.5.1, " Initial Setup" on page 200.

2. Move the mouse over Window1 to where you want to place the ListBox object.
3. Hold down mousebutton 1 and stretch out the dimensions of the ListBox object to the size that you want. Notice that the starting point is the upper-left corner of the ListBox object, so you must stretch it out to the right and down.

We also want to create a push button. This will be pushed by the user to select a database. Use the tool palette to create a push button in the same manner that we just created the ListBox object. We need to customize this push button. With the mouse pointing at the push button, click on mousebutton 2. Click on **Properties**. Go to the **Text** page to change the caption to:

```
Press to Select
```

We can create an accelerator key for objects as well. A way to create an accelerator is to use the tilde character in the caption of the object. The first character after the tilde is the accelerator. For example, to create an accelerator character P for the push button, change the caption to ~Press to Select. Close the **Properties** window. Notice that the push button display has the P underlined. This signifies to the user that P is the accelerator for this object. Note that when an object is activated, the black dots on the border can be used to resize the object.

Using the tool palette create a DescriptiveText object whose caption is:

```
Select a Database
```

Place it just above the ListBox object.

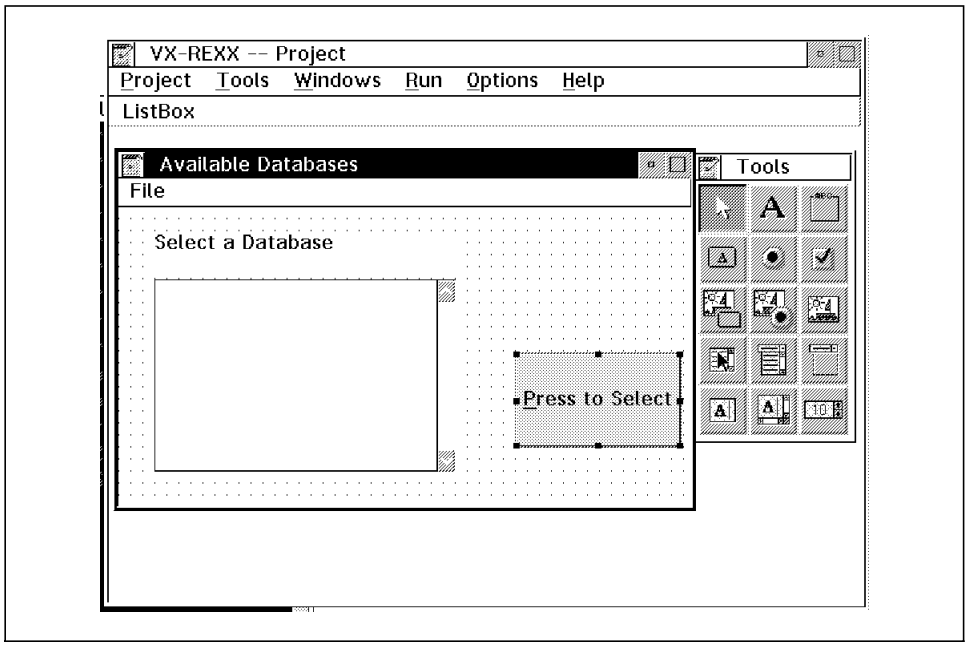


Figure 143. Window1 Customized

10.5.3 Program Initialization

Now that the primary window appears the way we want it to, we can look at how to take our existing SELECT.CMD application and incorporate it into this VX-REXX application that we are building. When the SELECT.CMD is invoked, the GETDB.CMD routine is called. This routine does a number of things before the user is prompted to do anything:

- If necessary, the functions SQLDBS and SQLEXEC are registered.
- If necessary, the Database Manager is started.
- A list of all available databases is displayed.

These things need to be done in the VX-REXX version of this application as well. In VX-REXX applications there is a routine called Init that is invoked when an application is started. So we can copy the lines of code from our GETDB.CMD into the Init routine. To bring up an edit session of the Init routine, do the following:

1. Click on **Windows** on the VX-REXX menu bar.
2. Click on **Section List**. This brings up a list of all the REXX routines in this application.

3. Double click on **Init**.

An easy way to copy the lines of code from the GETDB.CMD program into the Init routine is to create a subroutine in the Init routine and use the Import facility to place the GETDB.CMD code in the subroutine:

1. From the Init edit session, create a subroutine called GetDB. Make a call to the GetDB just before the call to VRMethod that activates the window.
2. Position the cursor under the GetDB label you just created.
3. Click on **Edit** on the edit session menu bar.
4. Click on **Import...**.
5. Find the correct drive and directory and then highlight the GetDB.CMD file.
6. Click on **Import**.

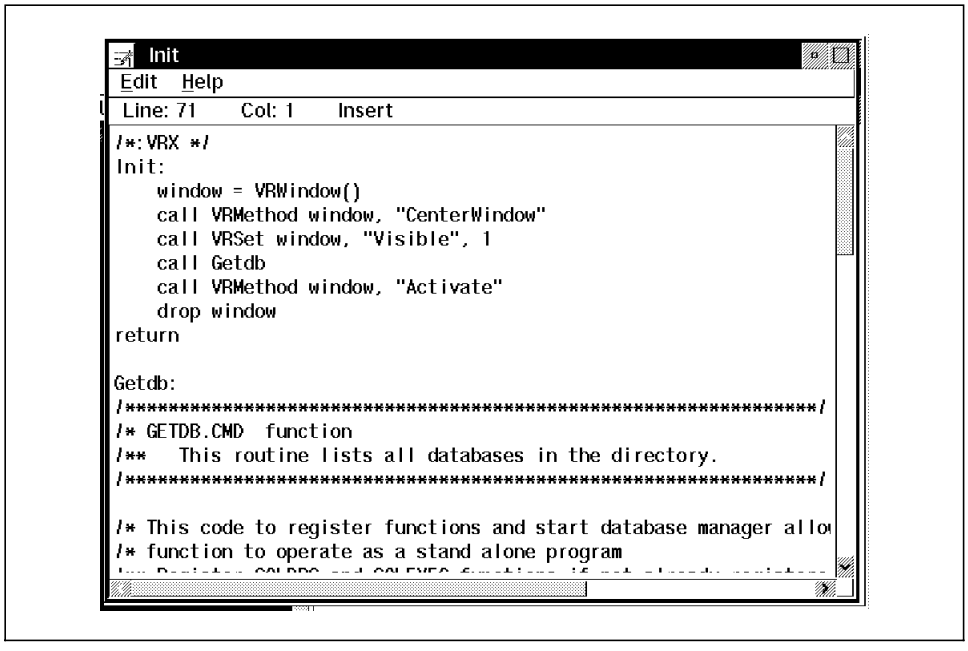


Figure 144. Init Routine

We need to replace the Say commands that displayed error notices with calls to VRMessage, which creates message boxes. Since we are in a visual, GUI environment, message boxes are more desirable than straight text displays using the Say command. To display a dialog message box, do the following:

1. Position the cursor in the edit session where you want to insert the VRMessage call. For example, we will create a message box to replace the following statement:

   ```
   say "Error registering SQLDBS: rc = " rc
   ```

2. Click on **Edit** on the edit session menu bar.
3. Click on **Insert code...**.
4. Double click on **Message Dialog** in the **Dialog** list.
5. Type the following into the **Text** entry field:

   ```
   "Error registering SQLDBS. RC = " rc
   ```

   This is the text string that appears in the message box. Make sure the **Quoted string** box is unchecked. If this box is checked, quotes will be put around whatever you enter in that field.
6. Type the word Error in the **Title** entry field. This is the title of the message box. Leave the **Quoted string** box checked.
7. Choose **Error** from the **Icon type** list box. This is the icon that will appear in the message box.
8. Type the following into the **Button list** entry field.

   ```
   ;OK
   ```

   The colon is used as a separator. OK is the caption for the button that will appear on the message box. More than one button can be displayed.
9. We will not use a Default or Escape parameter for this example. Click on **OK**.

The code for the message box is created. Use this technique to replace all Say statements where appropriate. Informational Say statements can be removed, since the GUI will make them unnecessary.

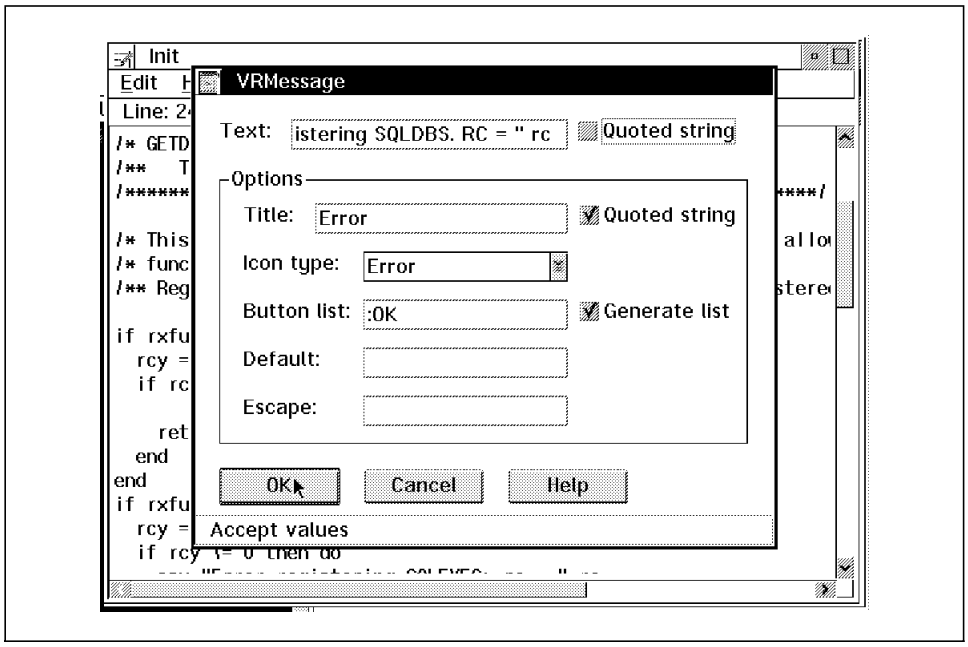


Figure 145. Creating a Message Box

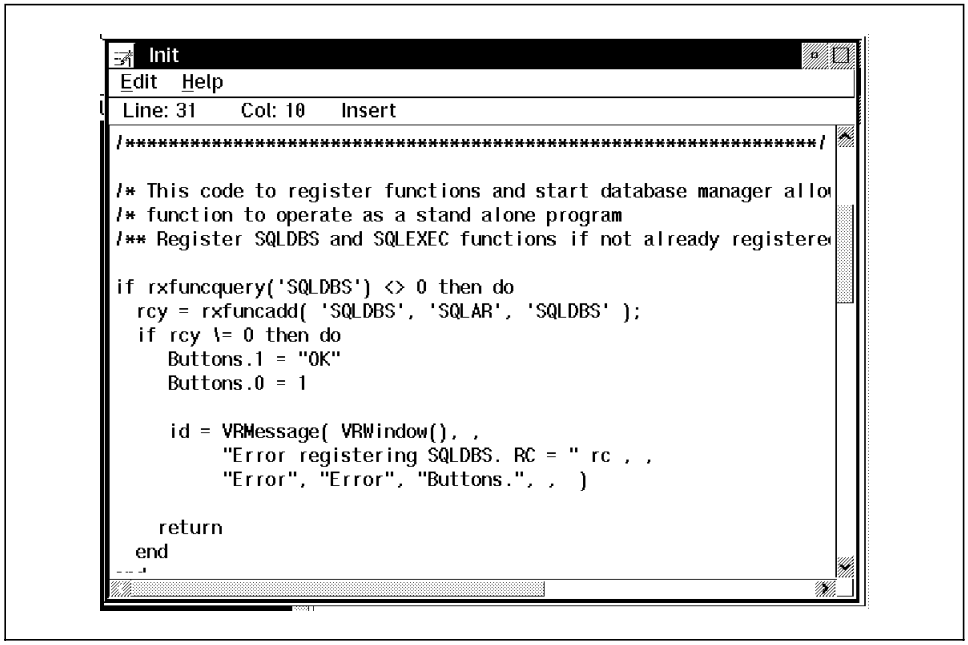


Figure 146. Message Box Code

There is one other thing we need to take care of here. How do we load the database names into the ListBox object? In the loop where the database names are read from the database catalog and displayed on the screen, we need to change that to load the ListBox object instead of displaying to the screen. VX-REXX provides a visual way to add this code. Do the following:

1. In the edit session of the Init routine, move the cursor to a new line where you want the code to be added. This new line of code should replace the following line:

   ```
   say '     '     entry.2
   ```

2. With the mouse pointing to the ListBox object, hold down mousebutton 2 and drag the listbox over to the edit session. You will see a line connecting the ListBox object and the edit session.

3. Let go of mousebutton 2 where you want the code to be added.

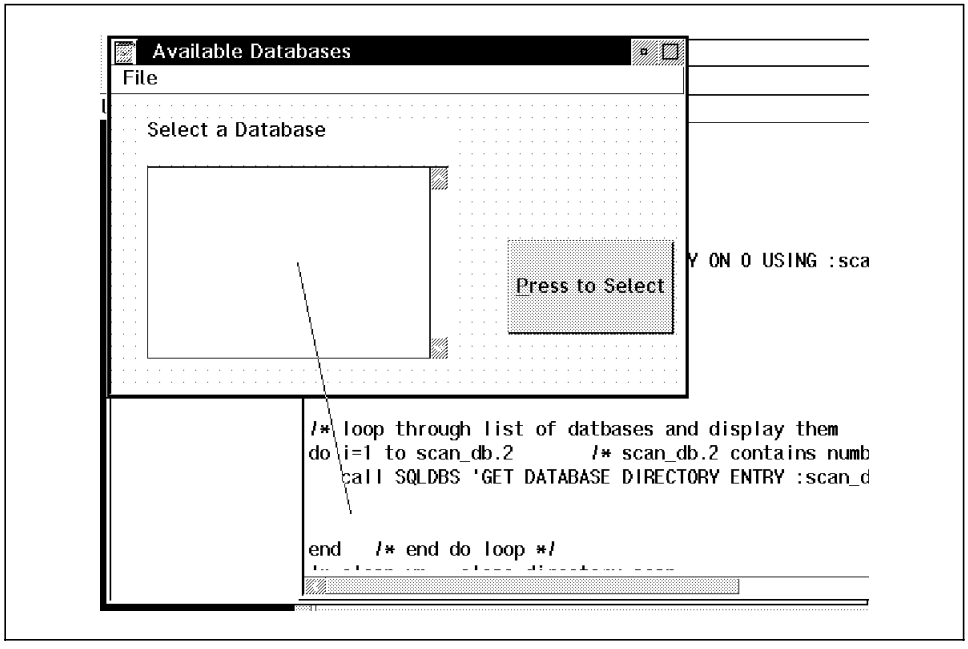


Figure 147. Drag ListBox Object

4. A window will appear. There are a list of actions that you can take for ListBox objects. In this case we want to add a string to the ListBox object, so double click on **Add a String**.

5. You are prompted to enter a string name. This is the value or variable that will be added to the ListBox object. We are using the variable entry.2, so type that in. It is a variable so uncheck the **Quoted string** box.
6. Click on **OK**. The call to VX-REXX function VRMethod with the correct parameters is added to the Init routine automatically.
7. Remove the pull statement. The user will interact with the program by clicking on a database to select it. It is no longer necessary for the user to type in a database name.
8. The return statement no longer needs to return a value.
9. Close the Init edit session.

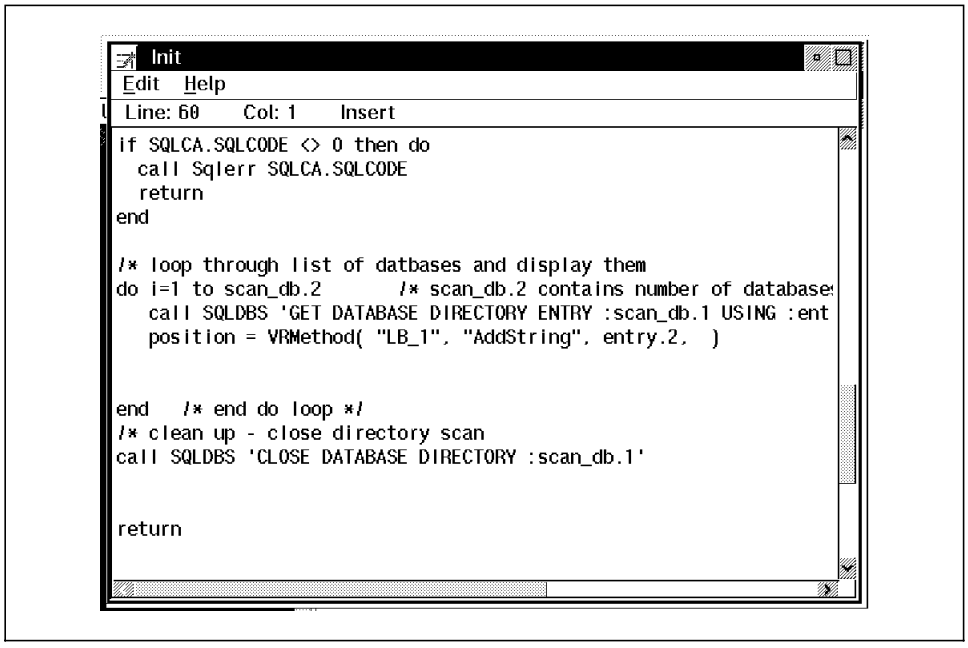


Figure 148. Init Routine Code to Load ListBox Object

10.5.4 Create the Table Window

When the user selects a database, we need to bring up a secondary window that displays a ListBox object containing all the tables for that database. To create a secondary window, do the following:

1. Click on **Windows** on the VX-REXX menu bar.
2. Click on **Window list**.

3. This brings up a list of all windows in the application. So far we just have Window1. Click on **Window** on the Windows menu bar.
4. Click on **New Window**. You have created a secondary window called SW_1.

Now we need to customize the window. Refer to 10.5.2, " Primary Window Setup (Window1)" on page 202 if necessary for a description of the customization of Window1. Customize window SW_1 as follows:

1. Change the title bar caption to:

   ```
   Select a Table to Query
   ```

2. Provide maximize and minimize buttons, and a system menu for the window.
3. Change the name of the window from SW_1 to Table using the **General** page of the **Properties** window.
4. Make this a modal window by using the **Style** page of the **Properties** window.
5. Create a ListBox object.
6. Create a DescriptiveText object, and position it just above the ListBox object. No caption is needed. We will add a caption later in 10.5.7, "Creating the Table Window" on page 218.
7. Create a push button object. The caption should be:

   ```
   Cancel
   ```

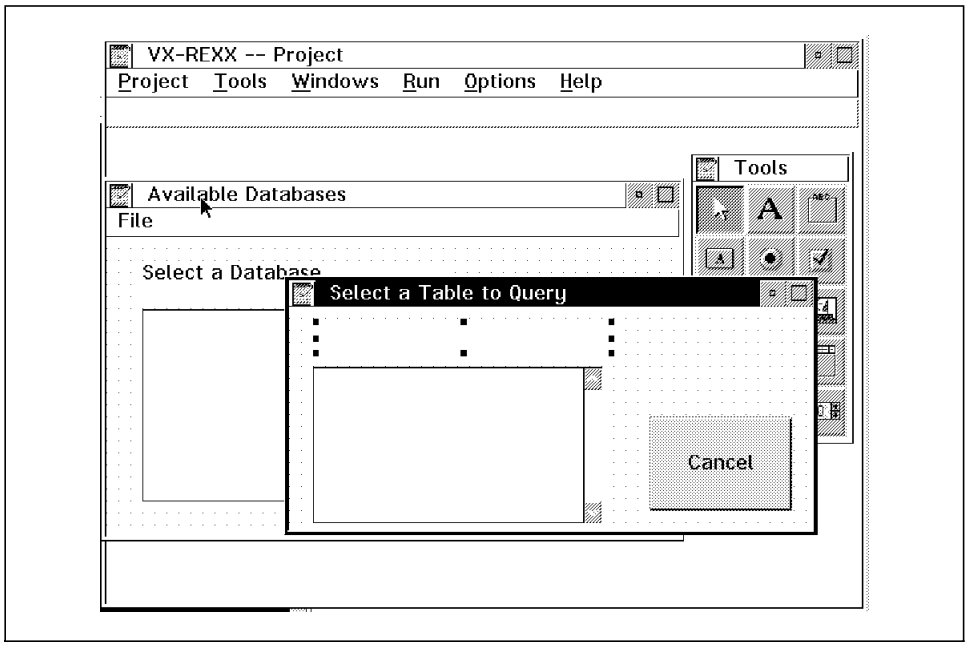


Figure 149. Table Window

10.5.5 Selecting a Database

The user needs to be able to select a database from the ListBox object in Window1. We have a push button on Window1 that we want the user to click on when they have highlighted the database they want to use. This will be the event that triggers our REXX code to bring up the Table window, with its ListBox object loaded with the selected database's table names. To put the REXX code in the right place, do the following:

1. With the mouse pointing to the push button on Window1, click on mousebutton 2.
2. Click on **Events**. This brings up a list of possible events for push buttons.
3. Click on **Click**. This will bring up an edit session for the routine PB_1_Click. PB_1_Click will be invoked when a user clicks on the push button.

Before we can create a list of table names, we need to know the name of the database selected by the user. To create this code, we can use the same drag and drop method we used to load the database ListBox object:

1. Drag and drop the database ListBox object to the edit session.

2. Double click on **Selected** in the **Get Property** list of the **Select an action** window. This will generate the code that gets the position of the selected field in the ListBox object. Position refers to a row of the ListBox object.

3. Perform drag and drop with the database ListBox object again. This time double click on **Get a string** from the **Select an action** window. Enter the variable name selected in the **Position** field. This variable holds the position of the selected field.

4. Click on **OK**. This will generate the code that gets the selected string from the ListBox object and places it in a variable called string.

5. Rename the variable named string to dbname. The rest of the application expects the database name to be in dbname.

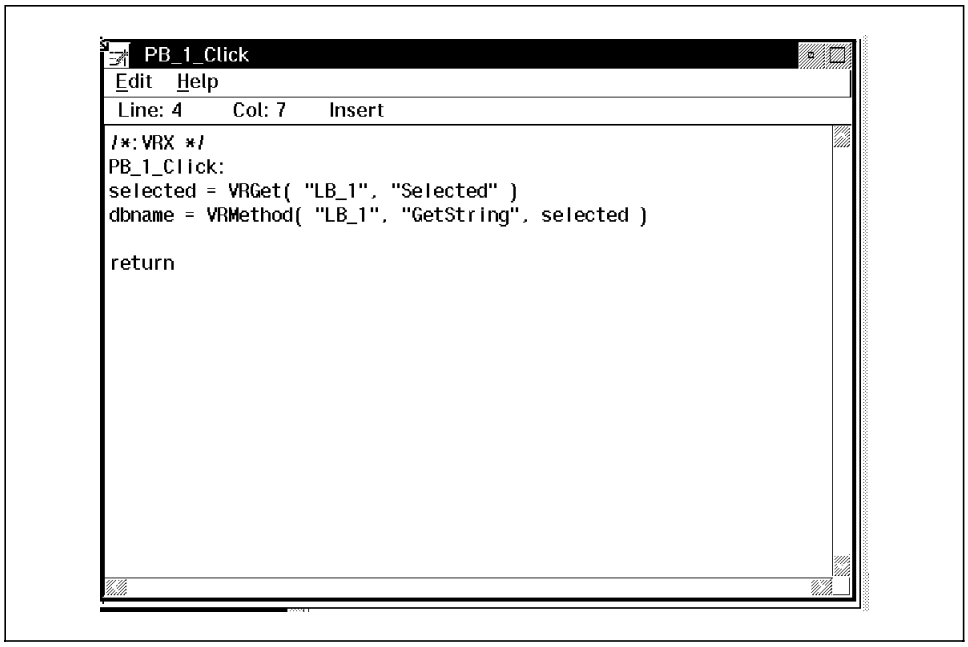


Figure 150. PB_1_Click Routine

We need to think about all the possible things a user can do with this window. For example, what if they haven't highlighted a database yet and they click on the push button, signifying they have selected a database? We need some code to handle this situation. If no field is highlighted in the ListBox object, the VRGet function will return a 0 for the position of the selected field. We need to check for this and display a message box to the user. To display a dialog message box, do the following:

1. In the edit session for PB_1_Click, click on **Edit** on the edit session menu bar.
2. Click on **Insert code...**.
3. Double click on **Message Dialog** in the **Dialog** list.
4. Type the following into the **Text** entry field:

   ```
   Highlight Database Before Selecting
   ```

   This is the text string that appears in the message box.
5. Type the word Error in the **Title** entry field. This is the title of the message box.
6. Choose **Error** from the **Icon type** list box. This is the icon that will appear in the message box.
7. Type the following into the **Button list** entry field.

   ```
   ;OK
   ```

   The colon is used as a separator. OK is the caption for the button that will appear on the message box. More than one button can be displayed.
8. We will not use Default or Escape parameters for this example. Click on **OK**.

The code for the message box is created. We need to write an IF statement so that this message box is only displayed if no database was selected.

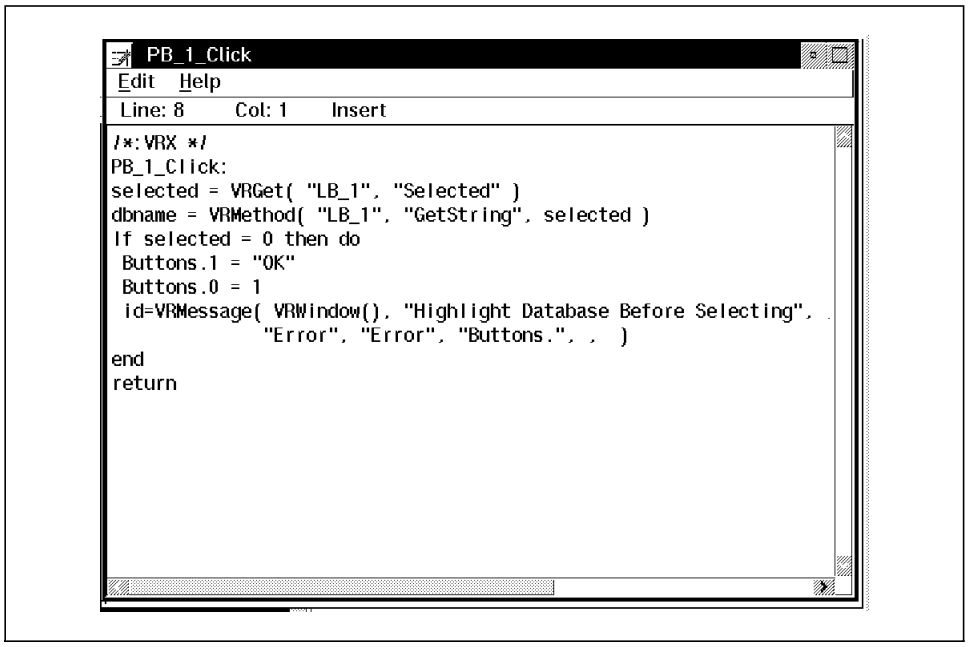


Figure 151. PB_1_Click Routine with Generated Code

10.5.6 Loading the Table Window

We are still in the PB_1_Click routine. We have the name of the database selected, so we are ready to bring up the Table window. To generate the code to create a secondary window, do the following:

1. Position the cursor in the edit session where you want the code to load a secondary window to be added.
2. Click on **Edit** on the edit session menu bar.
3. Click on **Insert code...**.
4. Double click on **Load a secondary window** from the **Object** list.
5. Enter Table in the window field, since this is the name of the table we want to load. Uncheck the **Quoted string** box, since Table is a variable name.
6. Click on **OK**. The call to load the Table window is generated.
7. Close the edit session.

Now the user can click on the Window1 push button to select a database. We also want to provide the user with the option of selecting a database by

double clicking on a database in the ListBox object. We need to do the following:

1. With the mouse pointing at the Window1 ListBox object, click on mousebutton 2.
2. Click on **Events**.
3. Click on **Double click**.

Now we are in the edit session for routine LB_1_DoubleClick. The routine is named this way because the ListBox object is named LB_1 by default (we could have changed it), and the event that will trigger this code to run is a double click on LB_1. We want to get the name of the database selected and load the Table window. Does this sound familiar? We already wrote this code. It is in the PB_1_Click routine. So we can just put the following code in the LB_1_DoubleClick routine:

```
call PB_1_Click
```

This is one way to handle common routines. Some may say this is a dangerous method, because if you ever delete the push button, you must remember to move the PB_1_Click routine code into the LB_1_DoubleClick routine. Another way to handle this would be to create a general routine (more about these in 10.5.10, " General Routines" on page 222) that contains the code, and have both PB_1_Click and LB_1_Click call the general routine. Close the edit session.

10.5.7 Creating the Table Window

Now we can import the code from GETTABLE.CMD into our application. We want this code to run when the Table window is created. There is a routine that is called when the Table window is created. To bring up an edit session for that routine, do the following:

1. With your mouse pointing at the Table window, click on mousebutton 2.
2. Click on **Events**.
3. Click on **Create**.

We are now in an edit session for the routine called Table_Create. To include the code from GETTABLE.CMD, follow the same method of importing code as was used in 10.5.3, " Program Initialization" on page 207. Here are some pointers:

1. To load the table names into the ListBox object in the Table window, follow the same procedure that was used to load database names into the Window1 ListBox object. Remember that this code is located in the Init routine if you need a reference. Use the drag and drop technique to

generate the call to load the ListBox object. The code that must be changed is:

```
say 'Table = ' STRIP(creator_name)||'.'||STRIP(table_name)
```

2. Remember to remove unnecessary Say statements and replace error messages with calls to VRMessage.
3. The parse arg dbname line is not needed, since dbname is a global variable.
4. The pull statement is not necessary, since the user will select the table from the ListBox object.
5. A value does not need to be returned from this routine.

It would be nice if there was a caption in the Table window that contained the name of the database these tables belonged to. We have a DescriptiveText object in the Table window, positioned just above the ListBox object. In the Table_create routine we can use the VRSet call to set the caption for the object to the name of the database that was selected. Refer to Figure 152. We are now finished with the TableCreate routine.

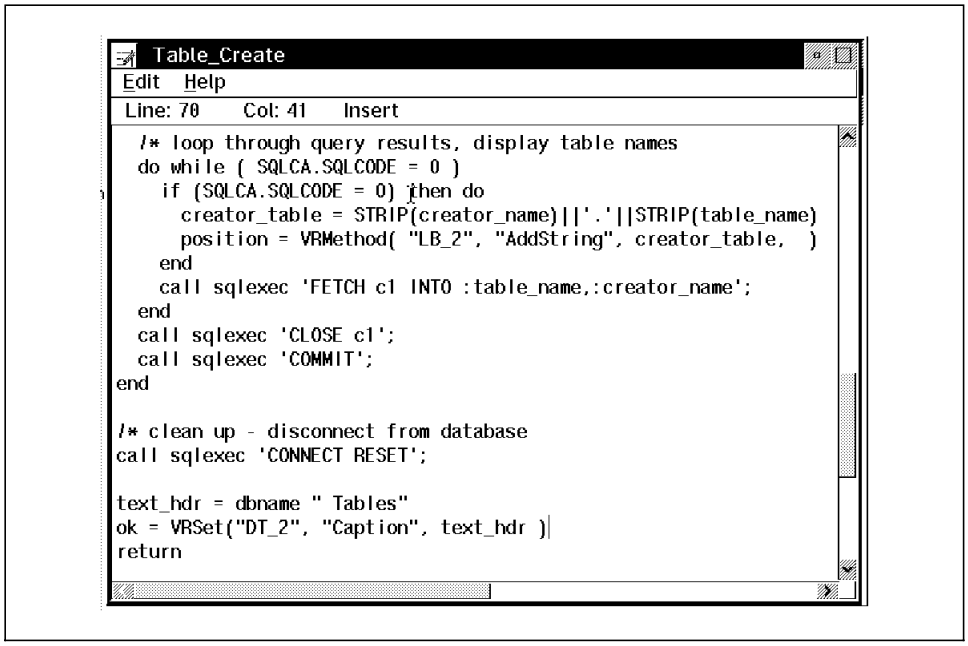


Figure 152. TableCreate Routine

10.5.8 Selecting a Table

We will use a double click on a table in the Table window's ListBox object as the method by which the user can select a table. This event will trigger a REXX subroutine. This subroutine is where we need to put the code from SELECT.CMD that runs the DB2/2 SELECT statement, loads the results into a file, and brings up the file in an E editor session. To get the code in the right subroutine, do the following:

1. With the mouse pointing at the Table window ListBox object, click on mousebutton 2.
2. Click on **Events**.
3. Click on **Double click**.
4. Now we are in the edit session. Try to cut and paste the code needed from SELECT.CMD. Cut and paste is an alternative to the import technique, which we used for GETTABLE.CMD and GETDB.CMD. You will need to copy the code from just below the call to GETTABLE down to the end of the program.
5. In order to get the name of the table selected, follow the same drag and drop technique used to get the database name selected from the ListBox object in Window1. A description of this is located in 10.5.5, " Selecting a Database" on page 214.
6. Remember to replace or remove Say statements.
7. Close the edit session when you are finished.

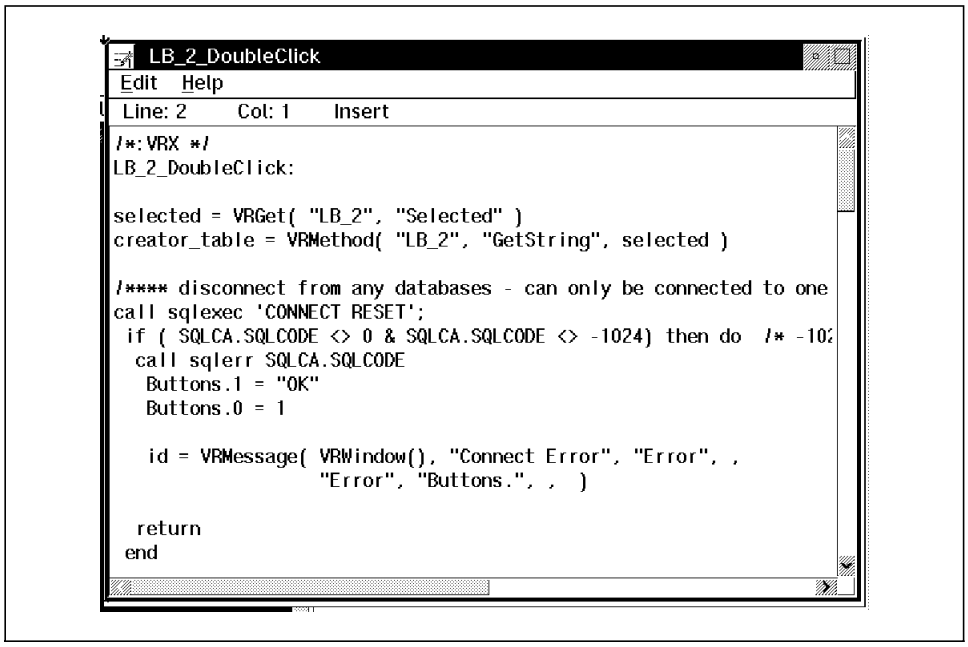


Figure 153. LB_2_DoubleClick Routine

10.5.9 Cancel from Table Window

Since the Table window is modal, when the Table window is active the user cannot make any other window in the application active. We need to provide the user with a way to leave the Table window. We will use the Cancel push button that we created on the Table window.

1. With the mouse pointing to the push button on the Table window, click on mousebutton 2.
2. Click on **Events**.
3. Click on **Click**. This brings up an edit session for the subroutine that is invoked when the user clicks on the push button. We need to put code here that will remove the Table window.
4. Use the VRDestroy function. Add the following code:

   ```
   call VRDestroy "Table"
   ```

5. Close the edit session.

10.5.10 General Routines

There are two types of routines in VX-REXX: event routines, and general routines. So far, we have looked primarily at event routines. These are routines that are invoked based on an event that was caused by the user interacting with the GUI. For example, clicking on a push button is a user initiated event. General routines are routines that are not invoked as a direct result of a GUI event. They are routines in the more traditional sense, used to break up a a program into logical units. In the SELECT.CMD program, there is an error handling routine called SQLERR.CMD. We want to include this routine in our VX-REXX version. It will be a general routine since it is not tied to a particular event directly. Do the following to bring up an edit session for a new general routine:

1. Click on **Windows** on the VX-REXX menu bar.
2. Click on **Section list**. This is the current list of routines in the application.
3. Click on **Section** on the Sections menu bar.
4. Click on **New...**.
5. Type Sqlerr for the name of the new section.
6. Click on **OK**.
7. Import the SQLERR.CMD code into the edit session.
8. Close the edit session.

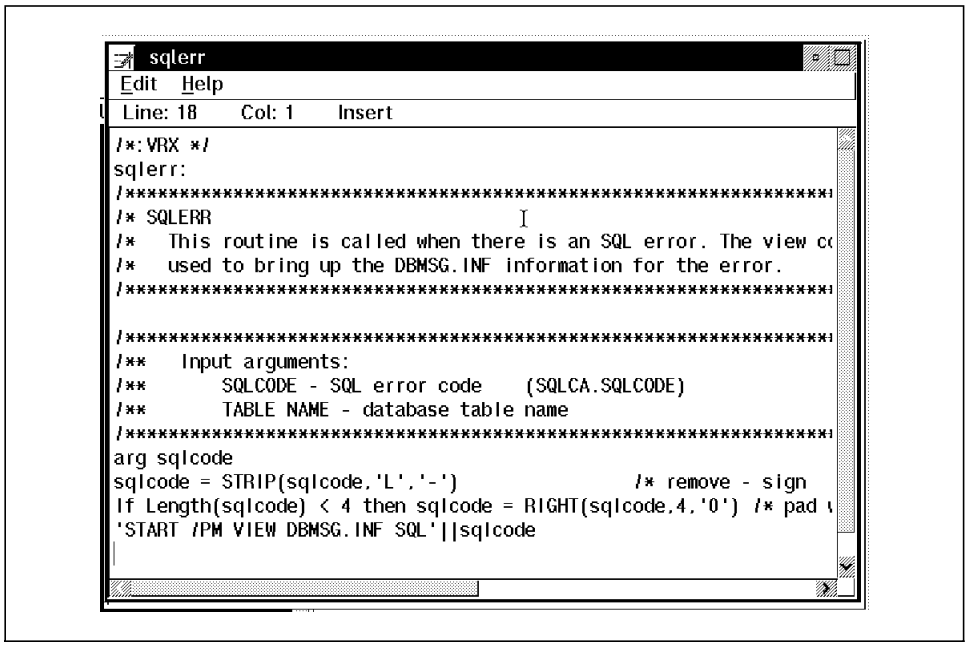


Figure 154. Sqlerr Routine

10.5.11 Testing Applications

VX-REXX provides a very easy to use test facility. You can run an application from within the development environment. You can also run the application in debug mode. The debugger contains all the main debug features like setting breakpoints, watching variables, stepping through lines of code, etc. To test your application, do the following:

1. Click on **Run** from the VX-REXX menu bar.
2. Click on **Run project** or **Debug project**.

There are a couple of common mistakes that we experienced using the VX-REXX application:

- When function calls are generated for you, be sure there are no quotes around variable names in the parameter list.
- When importing code, be sure to remove the extra return statement if necessary.

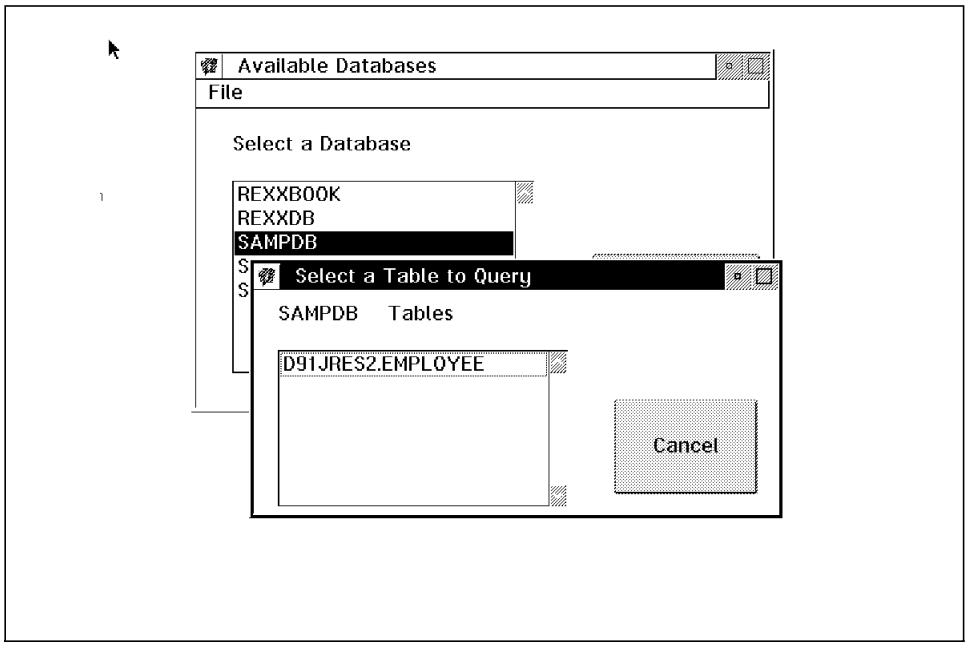


Figure 155. Select Application

10.5.12 Creating the Executable Version

VX-REXX provides a facility to take an application and create an .EXE version. This version can be executed on any machine that has the VROBJ.DLL of the VX-REXX package installed and placed in the LIBPATH of CONFIG.SYS. This run time DLL does not require a license. To create the .EXE version:

1. Click on **Project** on the VX-REXX menu bar.
2. Click on **Make EXE file...**.

The files associated with the SELECT project are:

- SEL121.VRP
- SEL121.VRX
- SEL121.VRY
- SEL121.EXE

you can invoke the SEL121 program in the same manner as you would invoke any executable file.

Appendix A. REXX Syntax Diagrams

We have included this appendix to give you a quick reference to the syntax of OS/2 REXX. For complete details of the parameters and usage of these statements, please refer to the OS/2 REXX online documentation or the *Procedures Language/2 REXX Reference,* of the *OS/2 2.0 Technical Library*.

Throughout this section, syntax is described using the structure defined below:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▸▸— symbol indicates the beginning of a statement.

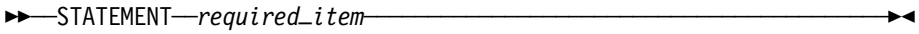
  The —▸ symbol indicates that the statement syntax is continued on the next line.

  The ▸— symbol indicates that a statement is continued from the previous line.

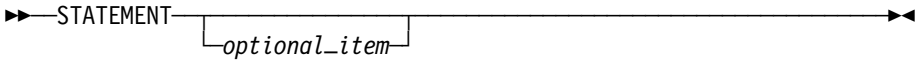
  The —▸◂ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ▸— symbol and end with the —▸ symbol.

- Required items appear on the horizontal line (the main path).

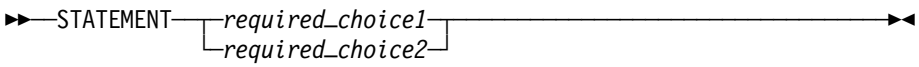


- Optional items appear below the main path.

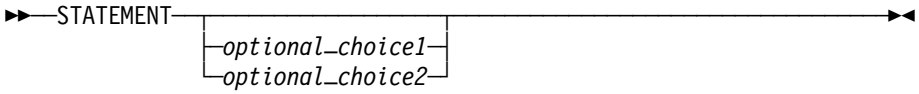


- If you can choose from two or more items, they appear vertically, in a stack.

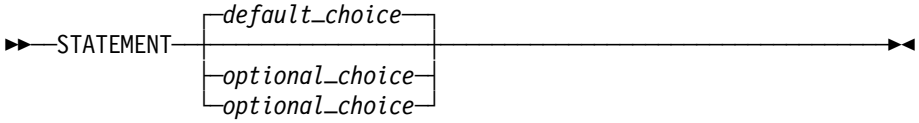
  If you *must* choose one of the items, one item of the stack appears on the main path.



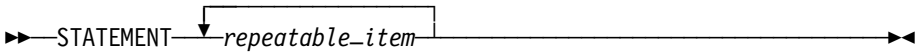
  If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



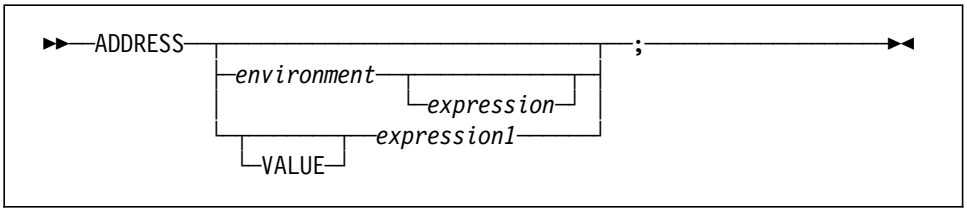
- An arrow returning to the left above the main line indicates an item that can be repeated.

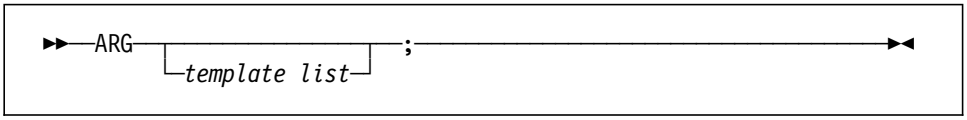


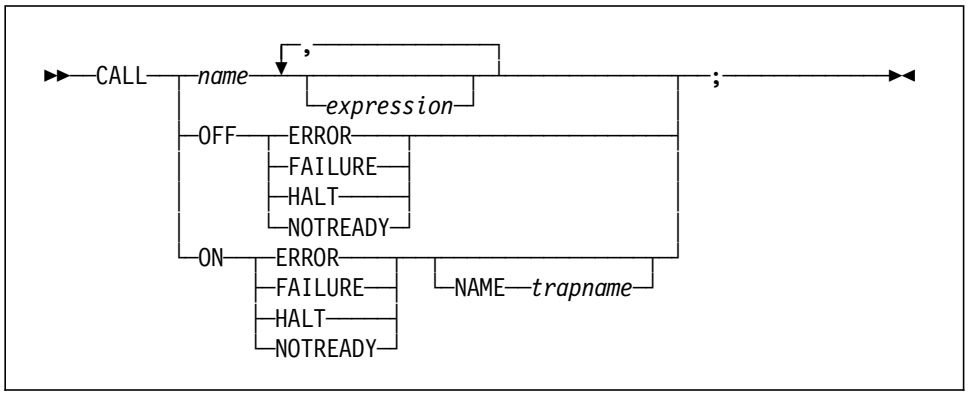
  A repeat arrow above a stack indicates that you can repeat the items in the stack.

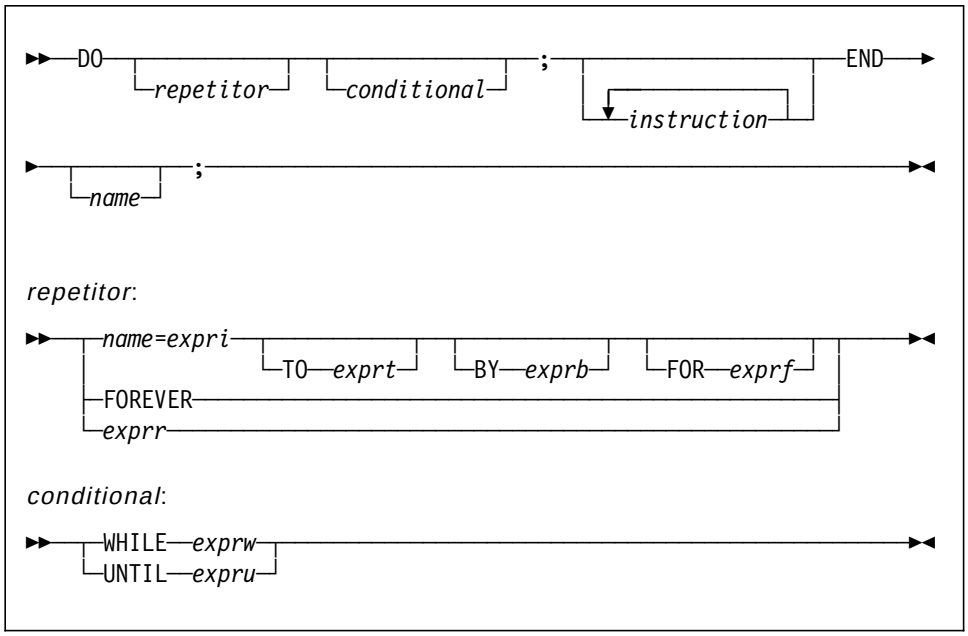
- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

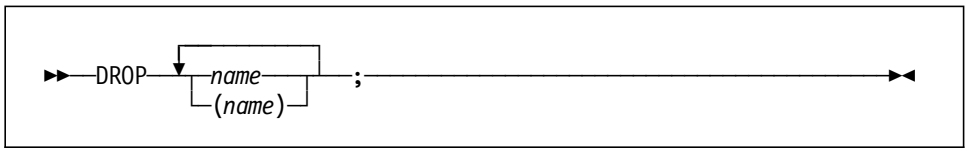
A.1 Keyword Instructions

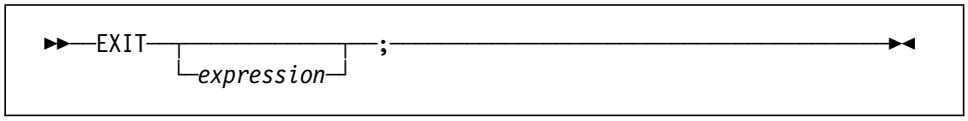


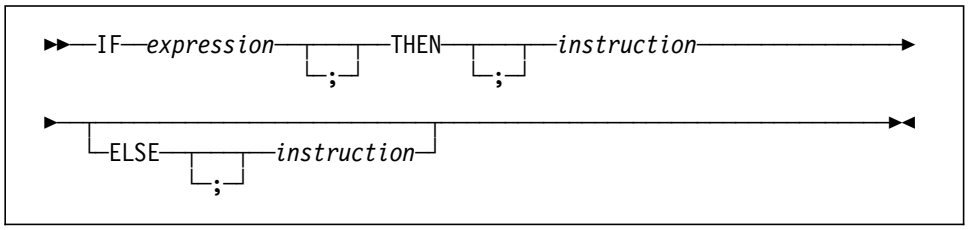


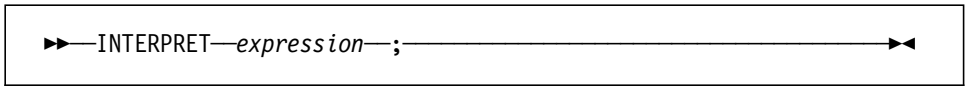

CALL
name
,
expression
;
OFF
ERROR
FAILURE
HALT
NOTREADY
ON
ERROR
FAILURE
HALT
NOTREADY
NAME
trapname

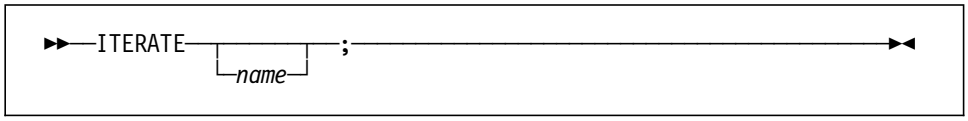

DO
repetitor
conditional
;
instruction
END
name
;
repetitor:
name=expri
TO
exprt
BY
exprb
FOR
exprf
FOREVER
exprr
conditional:
WHILE
exprw
UNTIL
expru

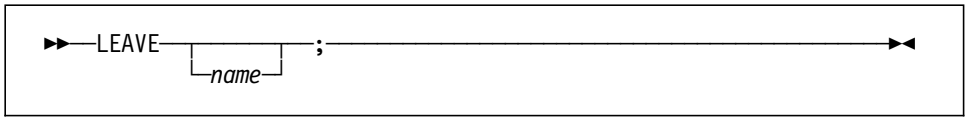

DROP
name
(name)
;

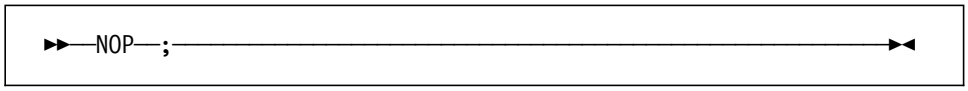

EXIT
expression
;

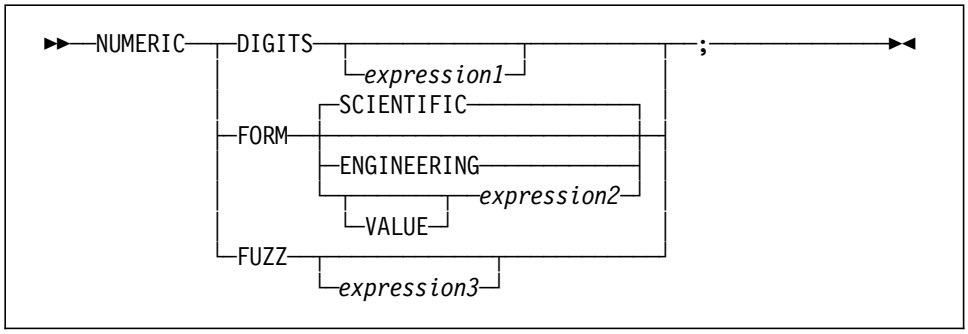

IF expression ; THEN ; instruction
ELSE ; instruction

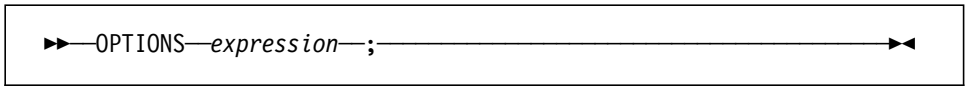

INTERPRET expression ;

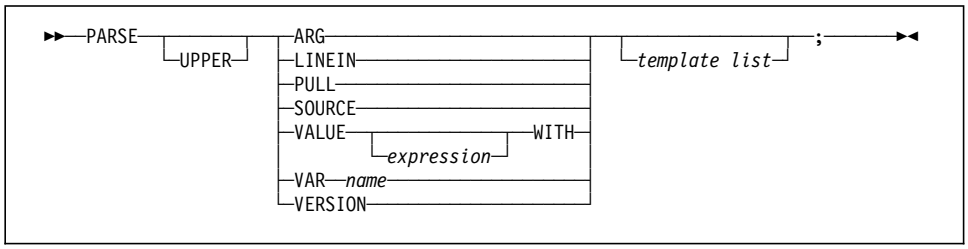

ITERATE ; name

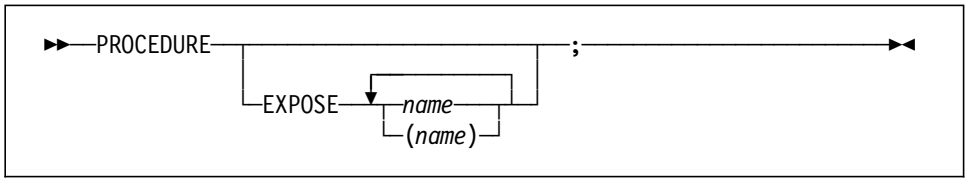

LEAVE ; name

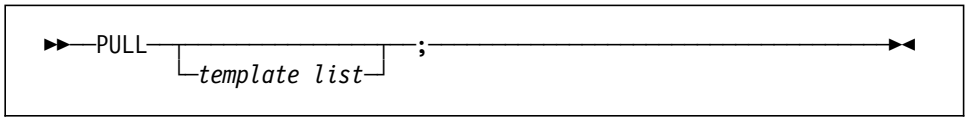

NOP ;

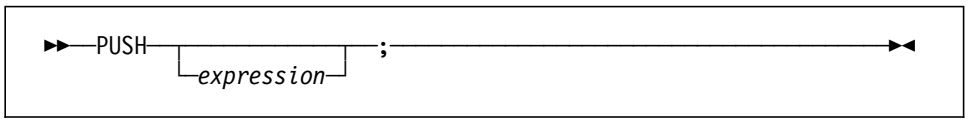

NUMERIC
DIGITS ; expression1
FORM SCIENTIFIC ENGINEERING VALUE expression2
FUZZ ; expression3

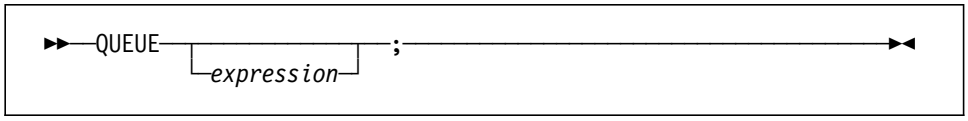

OPTIONS expression ;

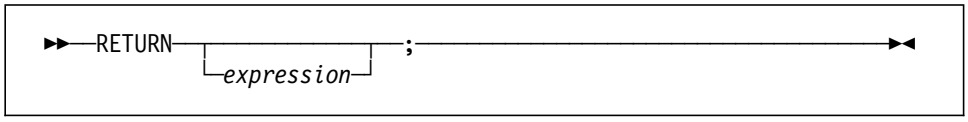

PARSE
UPPER
ARG
LINEIN
PULL
SOURCE
VALUE
expression
WITH
VAR
name
VERSION
template list
;

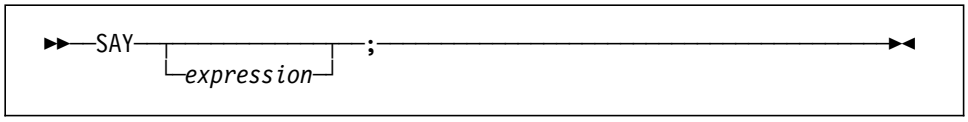

PROCEDURE
EXPOSE
name
(name)
;

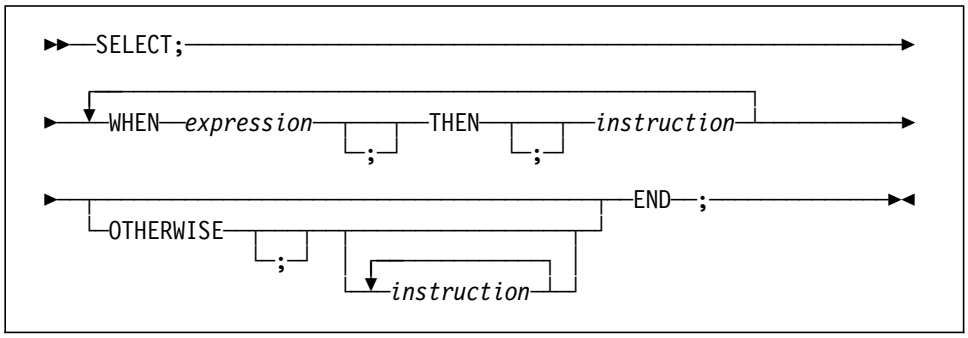

PULL
template list
;

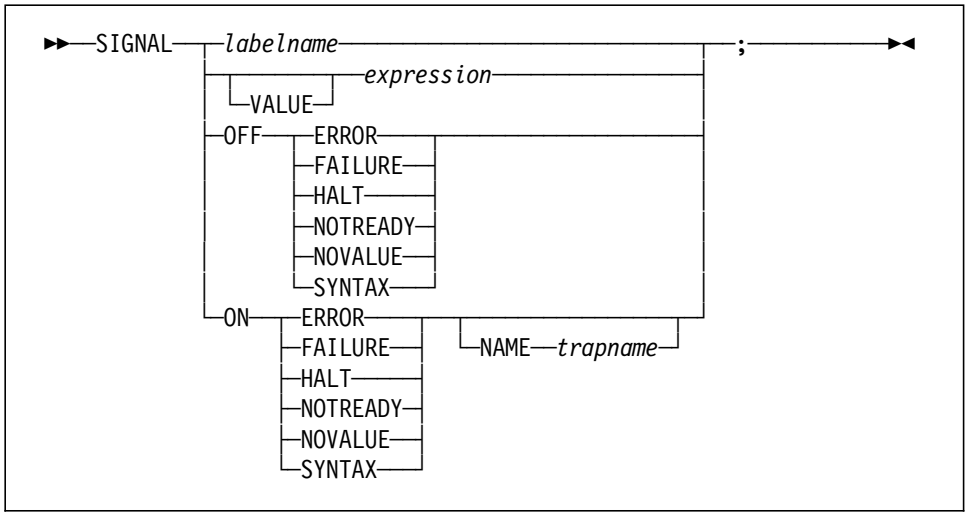

PUSH
expression
;

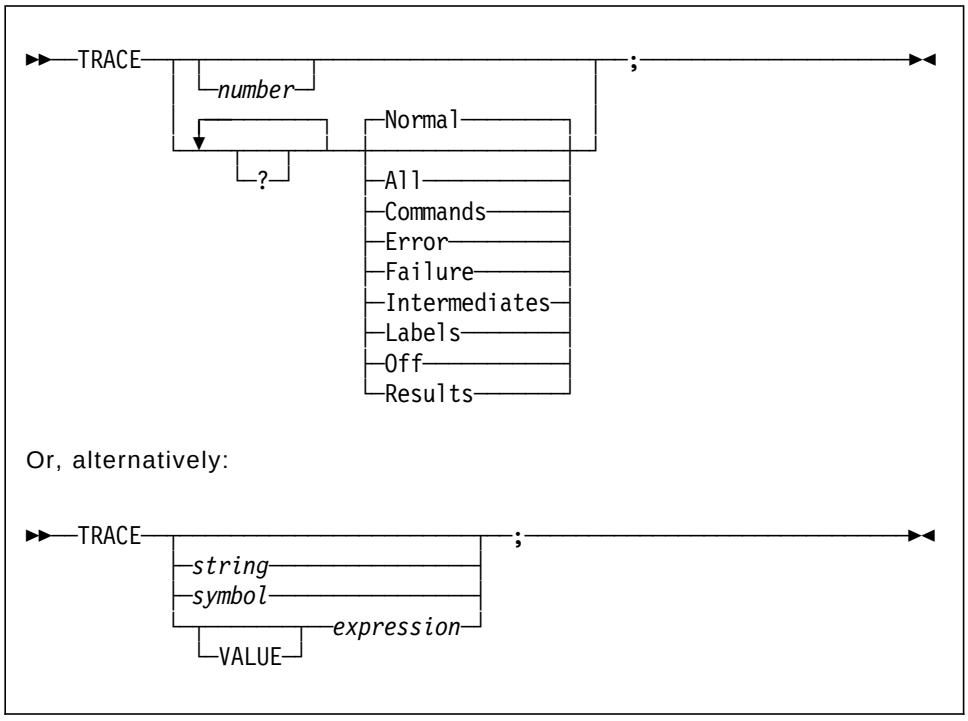

QUEUE
expression
;


RETURN
expression
;


SAY
expression
;

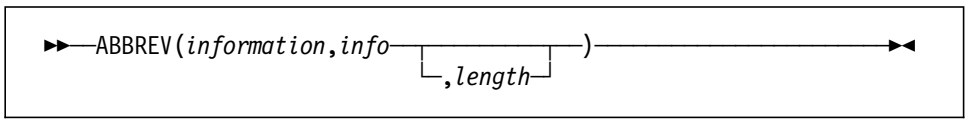

SELECT;
WHEN
expression
;
THEN
;
instruction
OTHERWISE
;
instruction
END
;

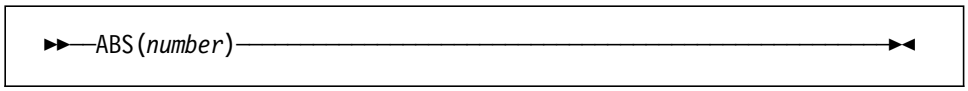

SIGNAL
labelname
;
VALUE
expression
OFF
ERROR
FAILURE
HALT
NOTREADY
NOVALUE
SYNTAX
ON
ERROR
FAILURE
HALT
NOTREADY
NOVALUE
SYNTAX
NAME
trapname

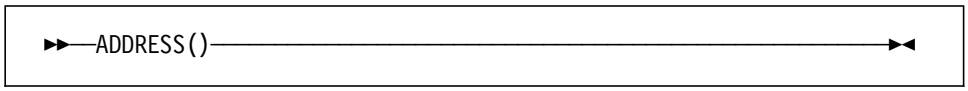


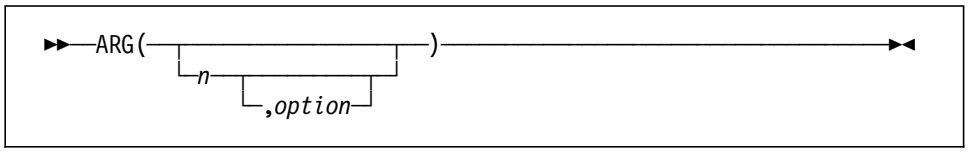
A.2 Functions

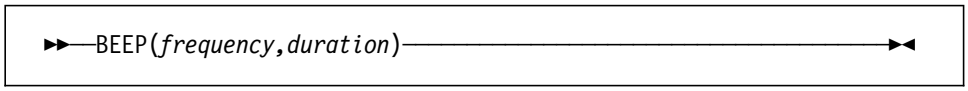
A.2.1 Built-in Functions

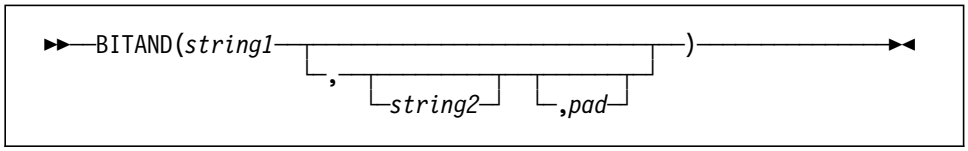


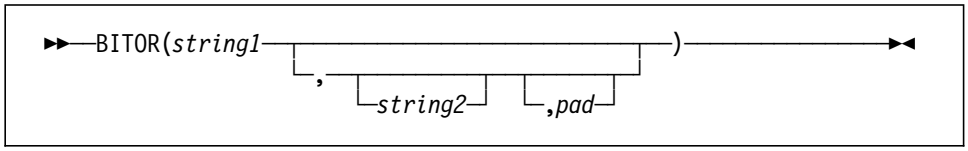


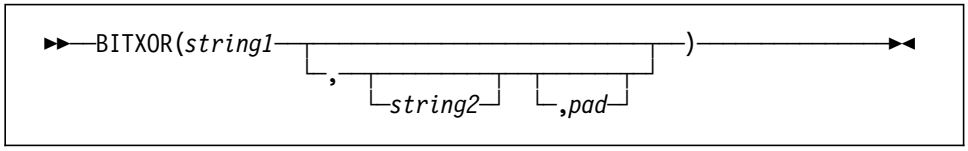


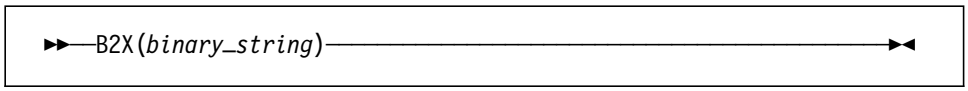

ARG(
n
,option
)

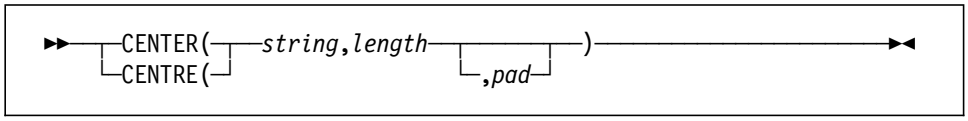

BEEP(frequency,duration)

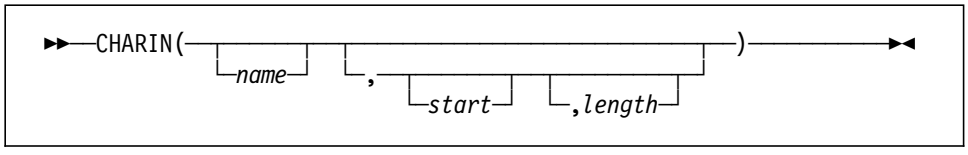

BITAND(string1
,
string2
,pad
)

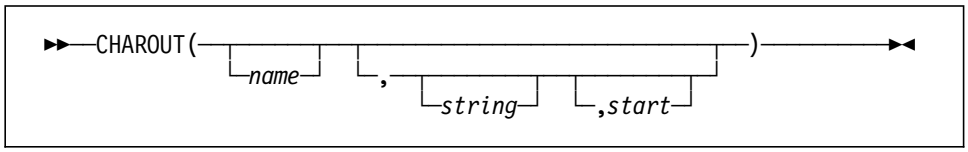

BITOR(string1
,
string2
,pad
)

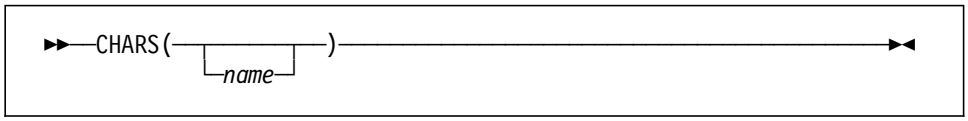

BITXOR(string1
,
string2
,pad
)

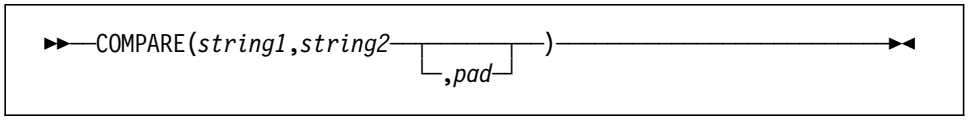

B2X(binary_string)

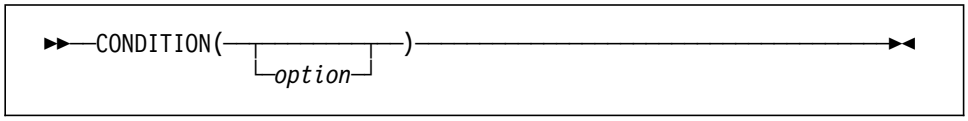

CENTER(
CENTRE(
string,length
,pad
)

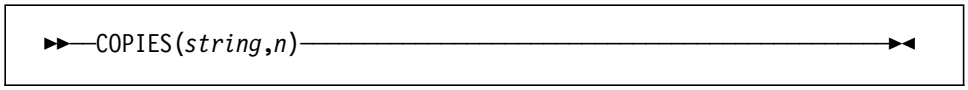

CHARIN(
name
,
start
,length
)

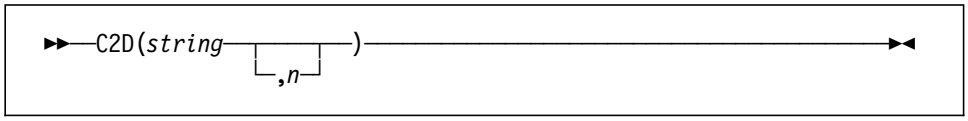

CHAROUT(
name
,
string
,start
)

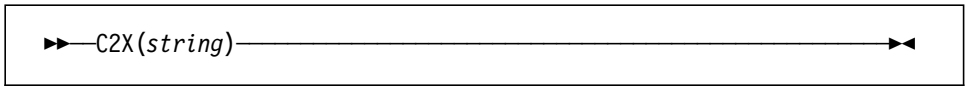

CHARS(
name
)

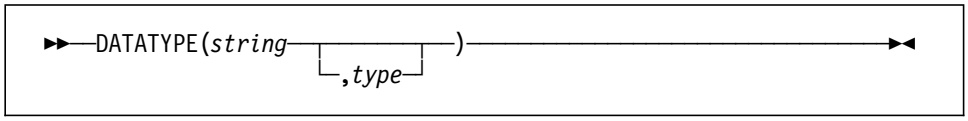

COMPARE(string1,string2
,pad
)

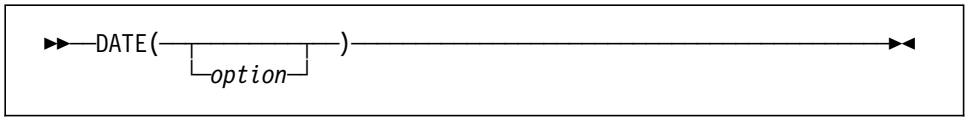

CONDITION(
option
)

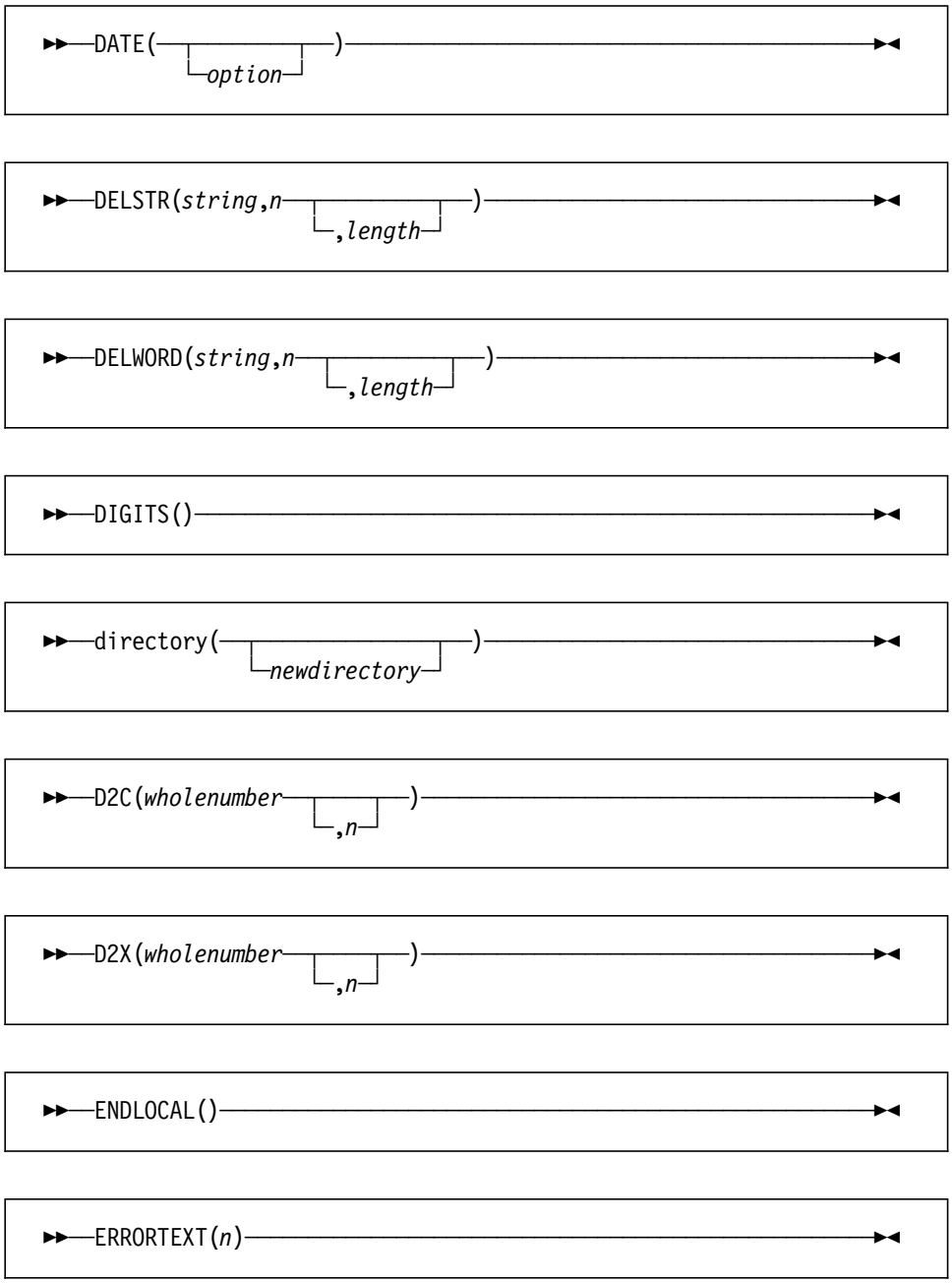

COPIES(string,n)

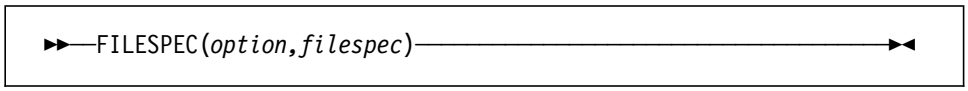

C2D(string
,n
)

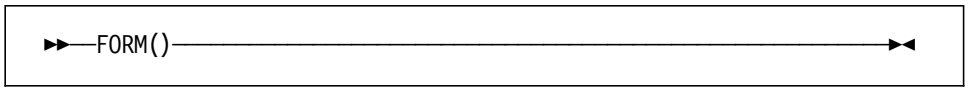

C2X(string)

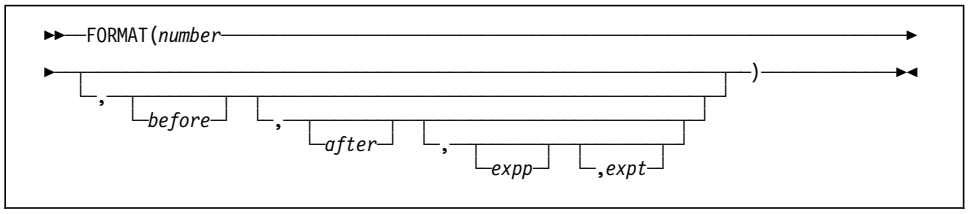

DATATYPE(string
,type
)

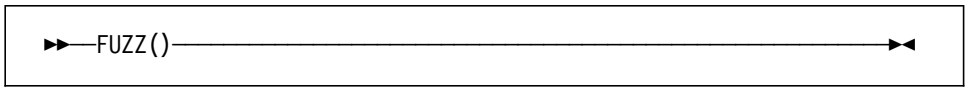

DATE(
option
)

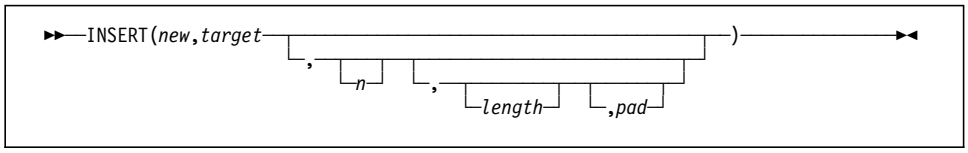

DATE(option)
DELSTR(string,n ,length)
DELWORD(string,n ,length)
DIGITS()
directory(newdirectory)
D2C(wholenumber ,n)
D2X(wholenumber ,n)
ENDLOCAL()
ERRORTEXT(n)

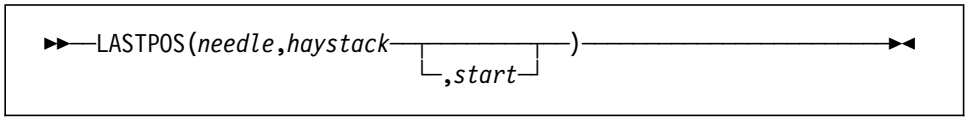

FILESPEC(option,filespec)

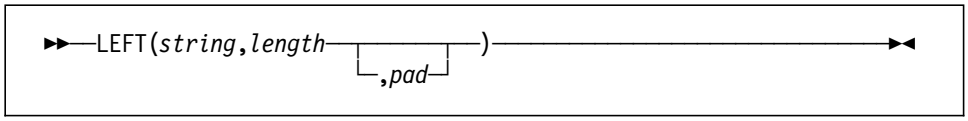

FORM()

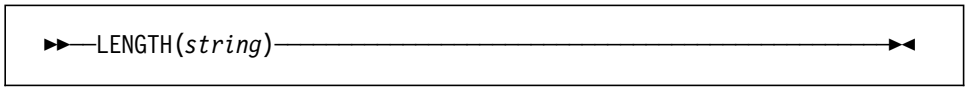

FORMAT(number
,
before
,
after
,
expp
,expt
)

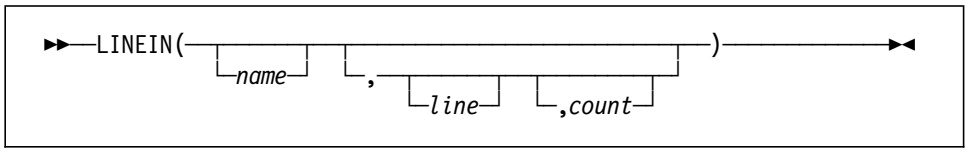

FUZZ()

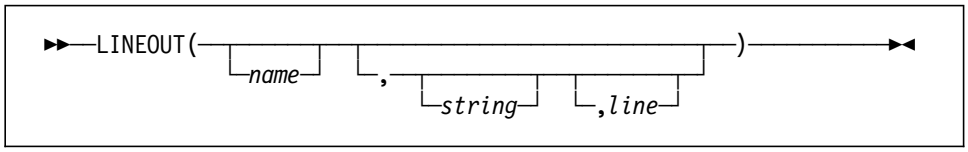

INSERT(new,target
,
n
,
length
,pad
)

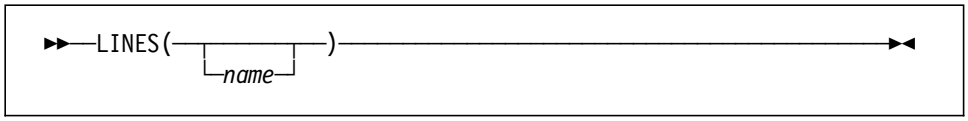

LASTPOS(needle,haystack
,start
)

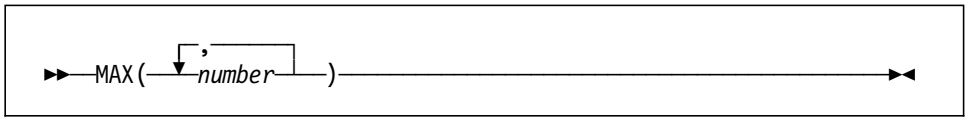

LEFT(string,length
,pad
)

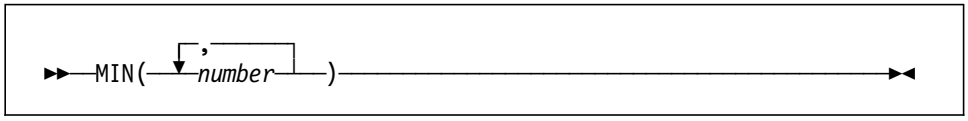

LENGTH(string)

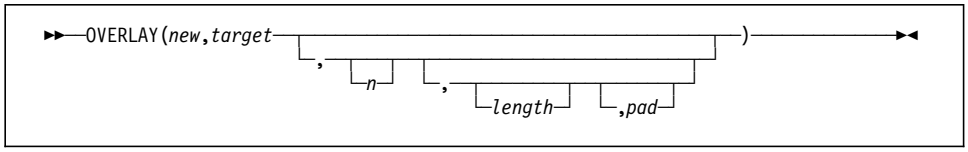

LINEIN(
name
,
line
,count
)

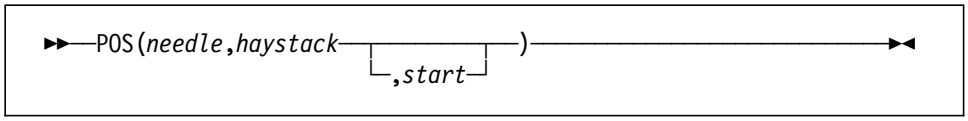

LINEOUT(
name
,
string
,line
)

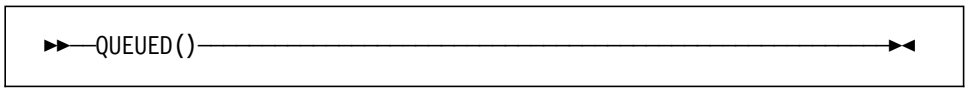

LINES(
name
)

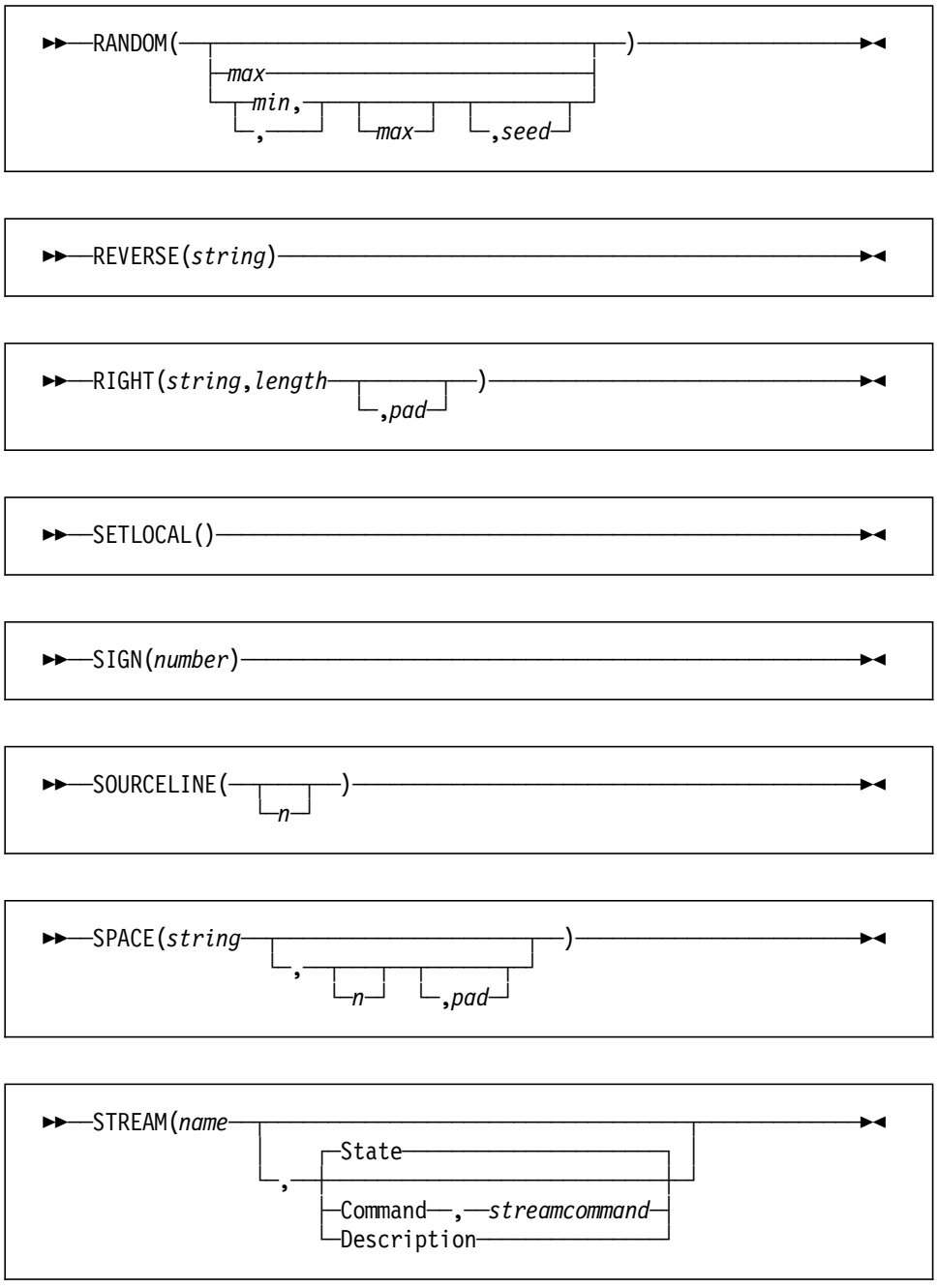

MAX(
,
number
)

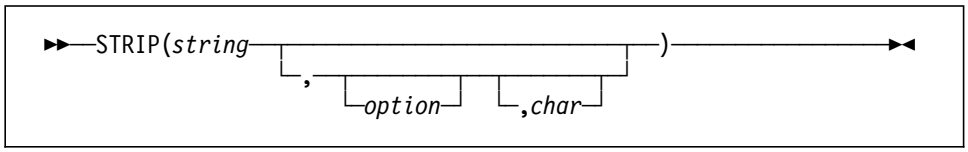

MIN(
,
number
)

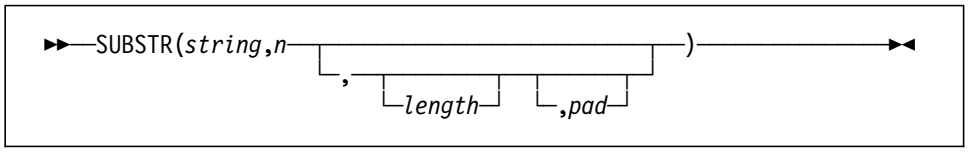

OVERLAY(new,target
,
n
,
length
,pad
)

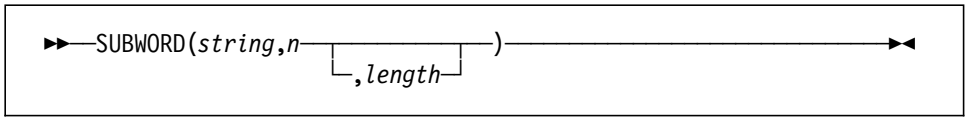

POS(needle,haystack
,start
)

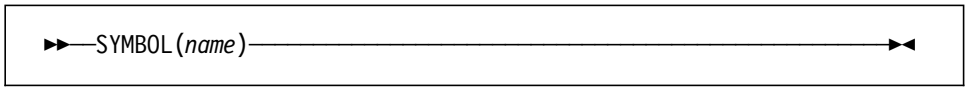

QUEUED()

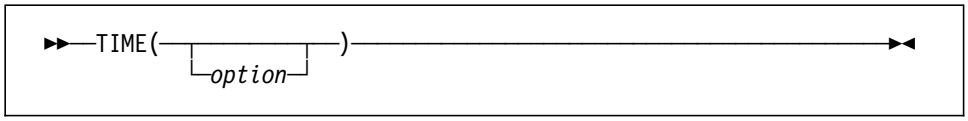

RANDOM(
max
min,
,
max
,seed
)
REVERSE(string)
RIGHT(string,length
,pad
)
SETLOCAL()
SIGN(number)
SOURCELINE(
n
)
SPACE(string
,
n
,pad
)
STREAM(name
,
State
Command
,streamcommand
Description

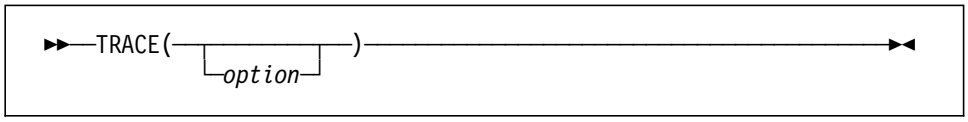

STRIP(string
,
option
,char

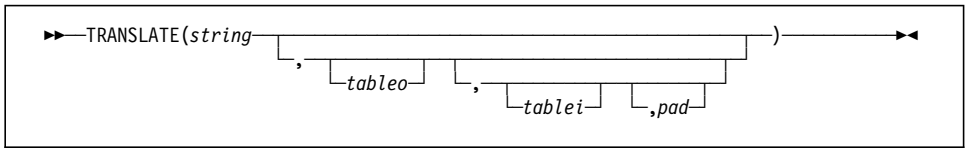

SUBSTR(string,n
,
length
,pad

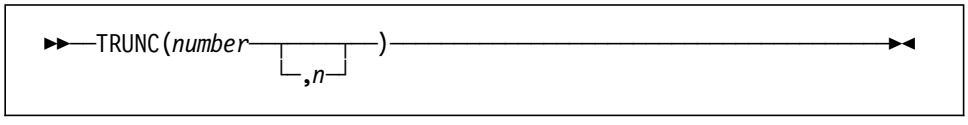

SUBWORD(string,n
,length

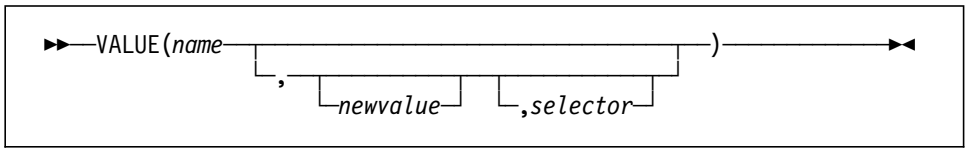

SYMBOL(name)

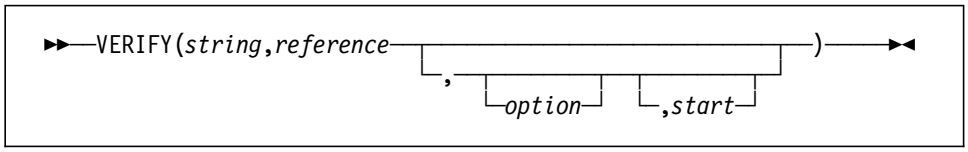

TIME(
option

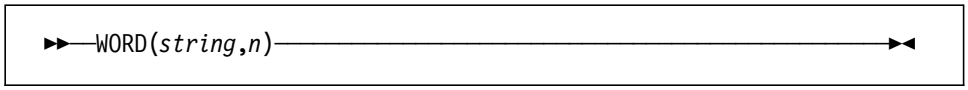

TRACE(
option

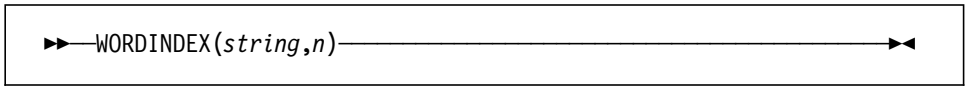

TRANSLATE(string
,
tableo
,
tablei
,pad

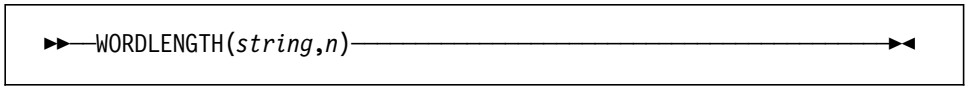

TRUNC(number
,n

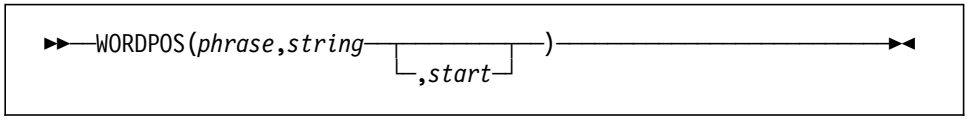

VALUE(name
,
newvalue
,selector
)

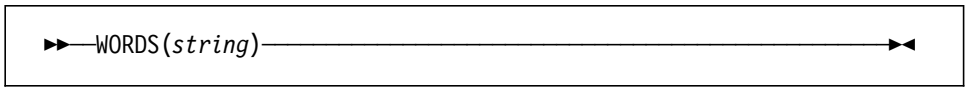

VERIFY(string,reference
,
option
,start
)

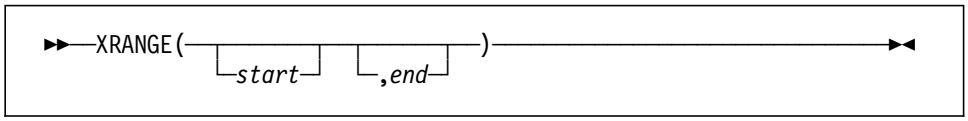

WORD(string,n)

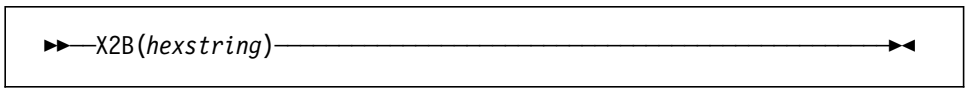

WORDINDEX(string,n)

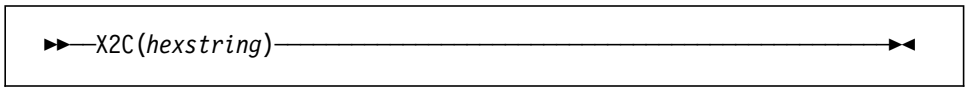

WORDLENGTH(string,n)

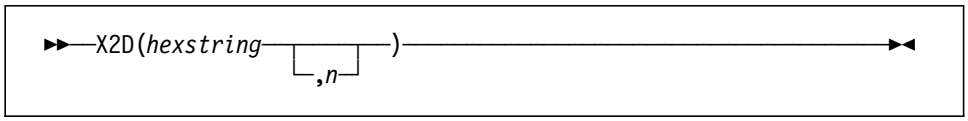

WORDPOS(phrase,string
,start
)


WORDS(string)

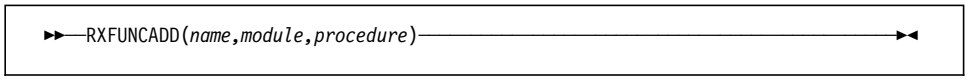

XRANGE(
start
,end
)

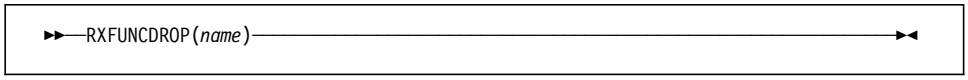

X2B(hexstring)

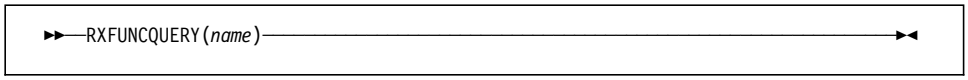


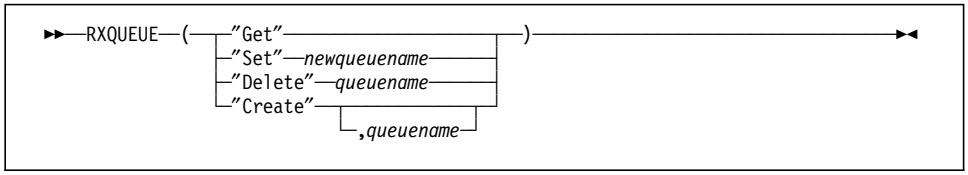


A.2.2 OS/2 API Functions







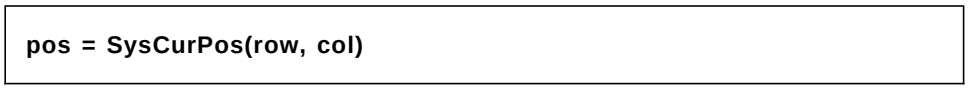


action = RxMessageBox(text, [title], [button], [icon])

A.2.3 REXX Utils Functions

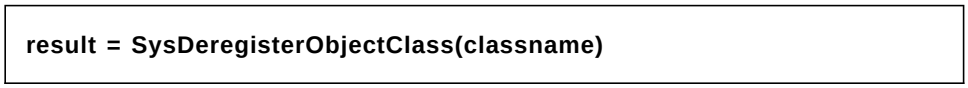
SysCls()

result = SysCreateObject(classname, title, location <,setup>,
<,duplicateflag>)

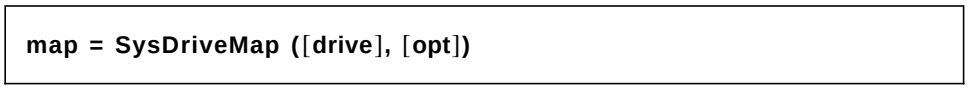

pos = SysCurPos(row, col)


SysCurState state


result = SysDestroyObject(name)

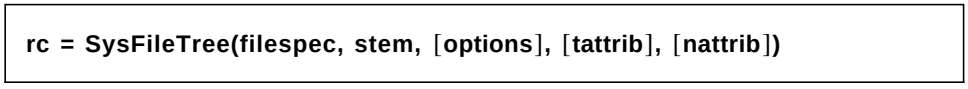

result = SysDeregisterObjectClass(classname)


info = SysDriveInfo (drive)


map = SysDriveMap ([drive], [opt])


call SysDropFuncs


rc = SysFileDelete(file)


rc = SysFileTree(filespec, stem, [options], [tattrib], [nattrib])

```
call SysFileSearch target, file, stem, [options]
```

```
result = SysGetEA(file, name, variable)
```

```
key = SysGetKey([opt])
```

```
msg = SysGetMessage(num, [file] [str1],...[str9])
```

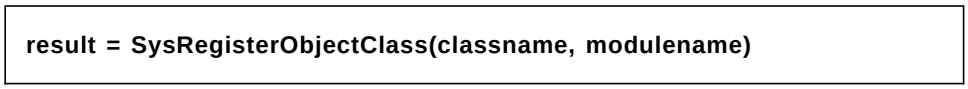
```
result = SysIni ([inifile], app, key, val, stem)
```

```
rc = SysMkDir(dirspec)
```

```
ver = SysOS2Ver()
```

```
result = SysPutEA(file, name, value)
```

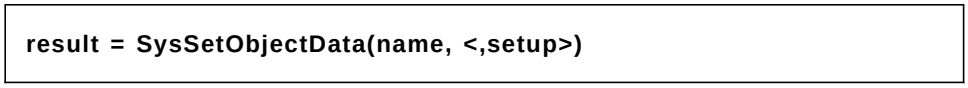
```
call  SysQueryClassList stem
```

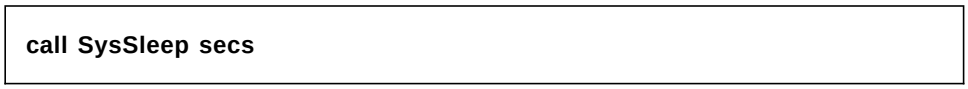

result = SysRegisterObjectClass(classname, modulename)


rc = SysRmDir(dirspec)

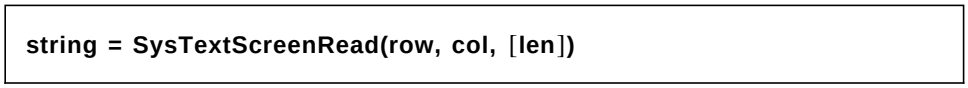

filespec = SysSearchPath(path, filename)

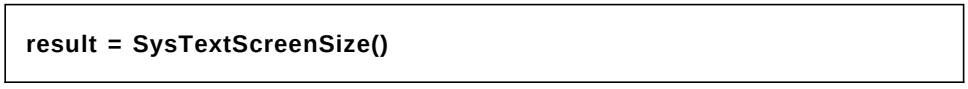

result = SysSetIcon(filename, iconfilename)


result = SysSetObjectData(name, <,setup>)


call SysSleep secs


file = SysTempFileName(template, [filter])


string = SysTextScreenRead(row, col, [len])


result = SysTextScreenSize()

```
result = SysWaitNamedPipe(name, [timeout])
```

Appendix B. OS/2 DB2/2 REXX Reference

We have included this appendix to give you a quick reference to the syntax of DB2/2 REXX APIs, SQL statements, and SQL Data Structures. For complete details of the parameters and usage of these statements, please refer to the *IBM Database 2 OS/2 Programming Reference* and the *IBM Database 2 OS/2 Structured Query Language(SQL) Reference*. The DB2/2 online documentation is another source for this information.

B.1 REXX DB2/2 API Syntax

Use the SQLDBS routine to call DB2/2 APIs with the following syntax:

```
call SQLDBS 'command string'
```

Enclose the command string in single quotes.

The following pages describe the correct syntax for calling DB2/2 APIs using the SQLDBS routine.

BACKUP DATABASE dbname [**{ALL|CHANGES}**] **TO drive**

BIND filename TO DATABASE dbname [**USING values**] [**MESSAGES msgfile**]

CATALOG DATABASE dbname [**AS alias**] **{ON drive|AT NODE nodename}** [[**IN codepage**] **WITH comment**]

CATALOG DCS DATABASE dbname [**AS tdbname**]
[**AR ar**]
[**PARMS "parms"**]
[[**IN codepage**] **WITH "comment"**]

```
CATALOG [APPC] NODE nodename [LOCAL locallu] REMOTE
 partnerlu[MODE mode] [ [IN codepage] WITH comment
 ]
```

```
CATALOG APPN NODE nodename [NETWORKID netid] REMOTE partnerlu
                    [LOCAL locallu]  [MODE mode]
                    [ [IN codepage] WITH comment ]
```

```
CATALOG NETBIOS NODE nodename REMOTE partnerlu ADAPTER
adapternum
                    [ [IN codepage] WITH comment ]
```

```
CHANGE DATABASE dbname COMMENT [ON drive] [IN
 codepage] WITH comment
```

```
CHANGE SQLISL TO {RR|CS|UR}
```

```
CLOSE DATABASE DIRECTORY scanid
```

```
CLOSE DCS DIRECTORY
```

```
CLOSE NODE DIRECTORY scanid
```

```
COLLECT { SYSTEM | DATABASE | ALL } STATUS
      [{ FOR DATABASE dbname | ON drive }] USING values
```

```
CREATE DATABASE dbname [ON drive] [ [IN
 codepage] WITH comment ]
        [ COLLATE {SYSTEM | NONE | USER udcs} ]
```

```
INVOKE progname
     [USING value] [INPUT DESCRIPTOR inda][OUTPUT
 DESCRIPTOR outda]
```

```
DROP DATABASE dbname
```

```
EXPORT stmt FROM dbname TO datafile OF filetype
   [MODIFIED BY filetmod] [USING dcoldata] MESSAGES
 msgfile:
CONTINUE EXPORT

STOP EXPORT
```

```
FREE STATUS RESOURCES
```

GET AUTHORIZATIONS value

GET DATABASE CONFIGURATION FOR dbname USING values

GET DATABASE MANAGER CONFIGURATION USING values

GET MESSAGE INTO msg [**LINEWIDTH width**]

GET DATABASE DIRECTORY ENTRY scanid [**USING value**]

GET DATABASE STATUS USING values

GET NODE DIRECTORY ENTRY scanid [**USING values**]

GET USER STATUS FOR DATABASE dbname USING values

IMPORT TO dbname FROM datafile OF filetype
[**MODIFIED BY filetmod**] [**METHOD {L|N|P} USING dcoldata**]
{INSERT|REPLACE|CREATE|INSERT_UPDATE|REPLACE_CREATE}
INTO tname [**(columns)**]
MESSAGES msgfile

CONTINUE IMPORT

STOP IMPORT

INSTALL SIGNAL HANDLER

INTERRUPT

MIGRATE DATABASE dbname

OPEN DCS DIRECTORY

OPEN DATABASE DIRECTORY ON drive USING value

OPEN NODE DIRECTORY USING value

REORG TABLE tablename IN dbname [**INDEX iname**] [**USE path**]

RESET DATABASE CONFIGURATION FOR dbname

RESET DATABASE MANAGER CONFIGURATION

RESTART DATABASE dbname

RESTORE DATABASE dbname FROM drive [**TO dbdrive**] [**WITHOUT ROLLING FORWARD**]

CONTINUE RESTORE

STOP RESTORE

ROLLFORWARD DATABASE dbname [{ **TO** {**isotime | END OF LOGS**} [**AND STOP**] } **|**
STOP | QUERY STATUS] [**USING values**]

RUNSTATS ON TABLE tname [**{AND | USING | FOR} INDEXES {ALL|USING values}**][**SHRLEVEL {REFERENCE|CHANGE}**]

START DATABASE MANAGER

STOP DATABASE MANAGER

UNCATALOG DATABASE dbname

UNCATALOG DCS DATABASE dbname [**USING values**]

UNCATALOG NODE nodename

UPDATE DATABASE CONFIGURATION FOR dbname USING values

UPDATE DATABASE MANAGER CONFIGURATION USING values

B.2 SQL Statements Syntax

This section contains syntax diagrams for SQL statements that can be invoked from REXX programs.

In this section, syntax is described using the structure defined below:

- Read the syntax diagrams from left to right, from top to bottom, following the path of the line.

  The ▶▶— symbol indicates the beginning of a statement.

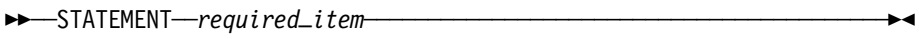
  The —▶ symbol indicates that the statement syntax is continued on the next line.

  The ▶— symbol indicates that a statement is continued from the previous line.

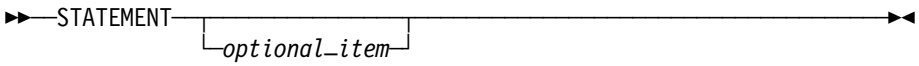
  The —▶◀ symbol indicates the end of a statement.

  Diagrams of syntactical units other than complete statements start with the ▶— symbol and end with the —▶ symbol.

- Required items appear on the horizontal line (the main path).

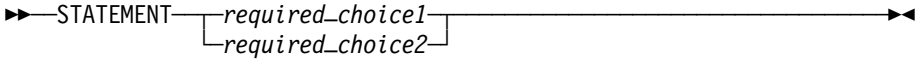


- Optional items appear below the main path.

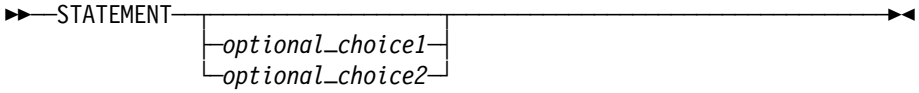


- If you can choose from two or more items, they appear vertically, in a stack.

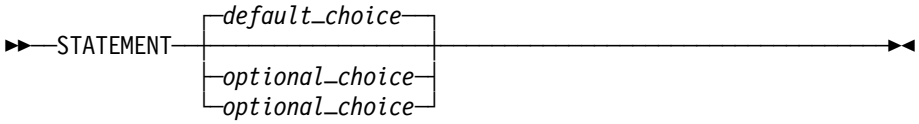
  If you *must* choose one of the items, one item of the stack appears on the main path.



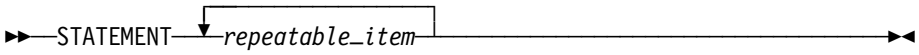
  If choosing one of the items is optional, the entire stack appears below the main path.



- If one of the items is the default, it will appear above the main path and the remaining choices will be shown below.



- An arrow returning to the left above the main line indicates an item that can be repeated.

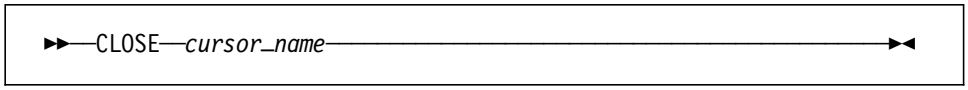


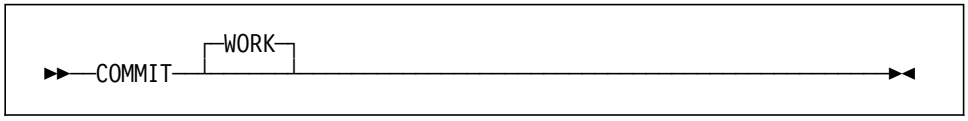
  A repeat arrow above a stack indicates that you can repeat the items in the stack.

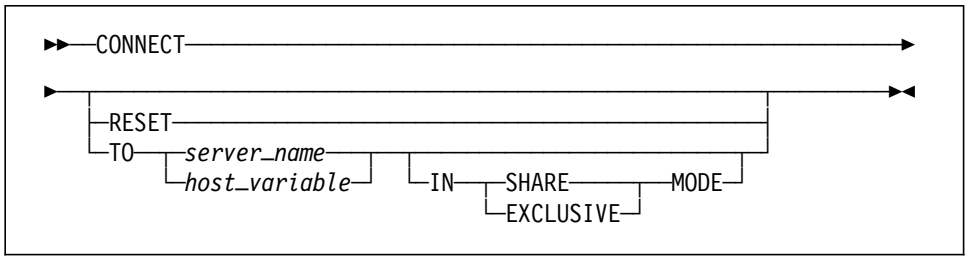
- Keywords appear in uppercase (for example, PARM1). They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *parmx*). They represent user-supplied names or values.

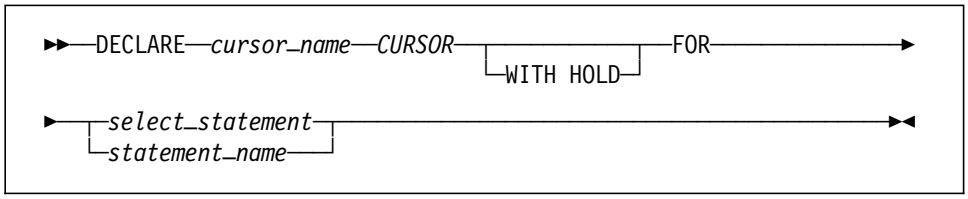
- If punctuation marks, parentheses, arithmetic operators, or such symbols are shown, you must enter them as part of the syntax.

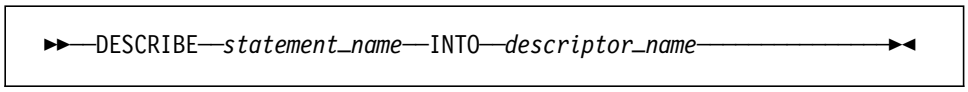
B.2.1 SQL Statements Passed Directly to SQLEXEC

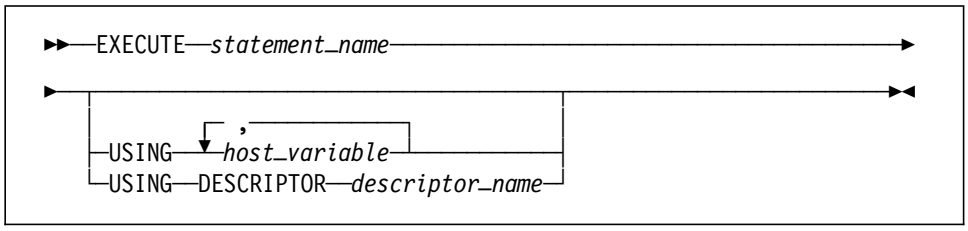


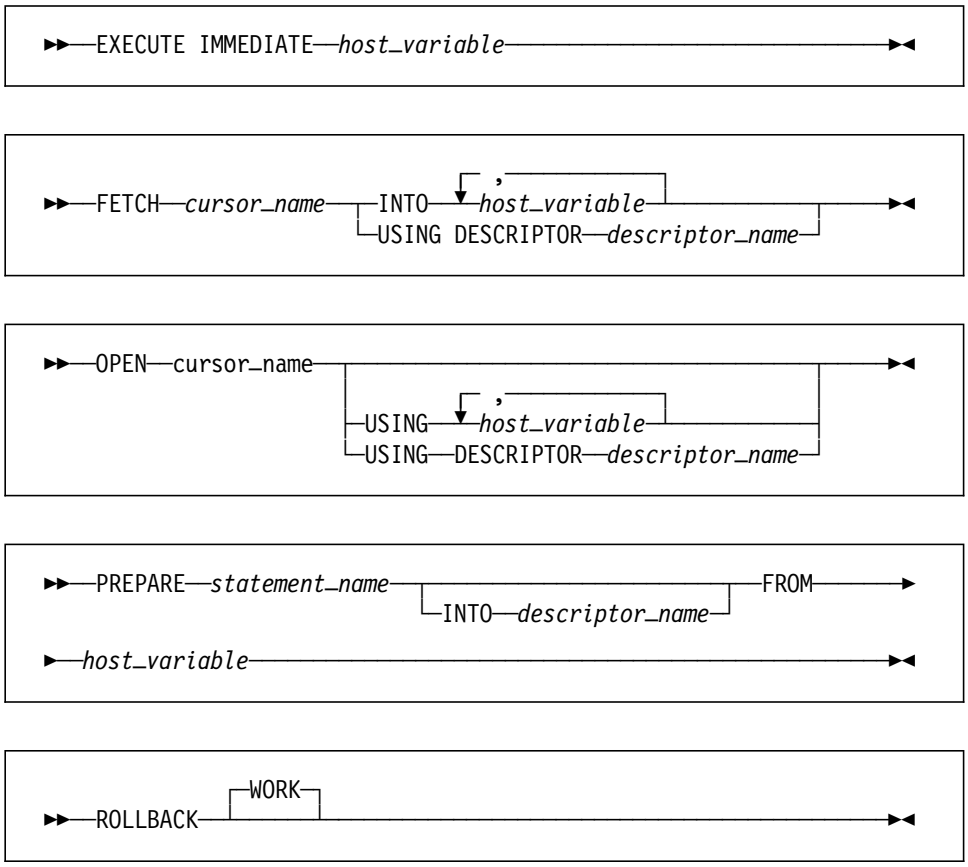








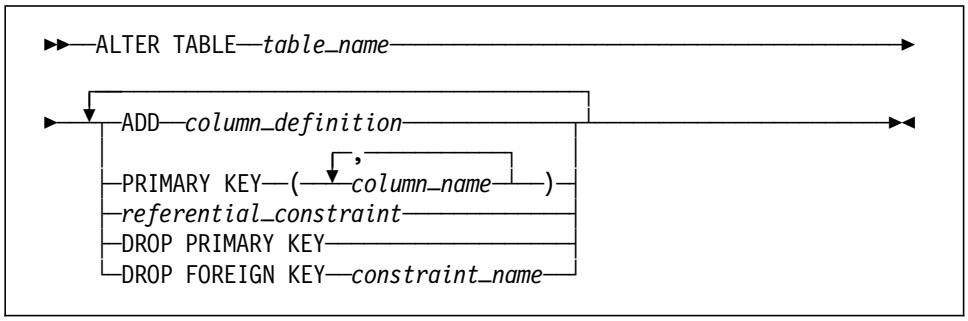


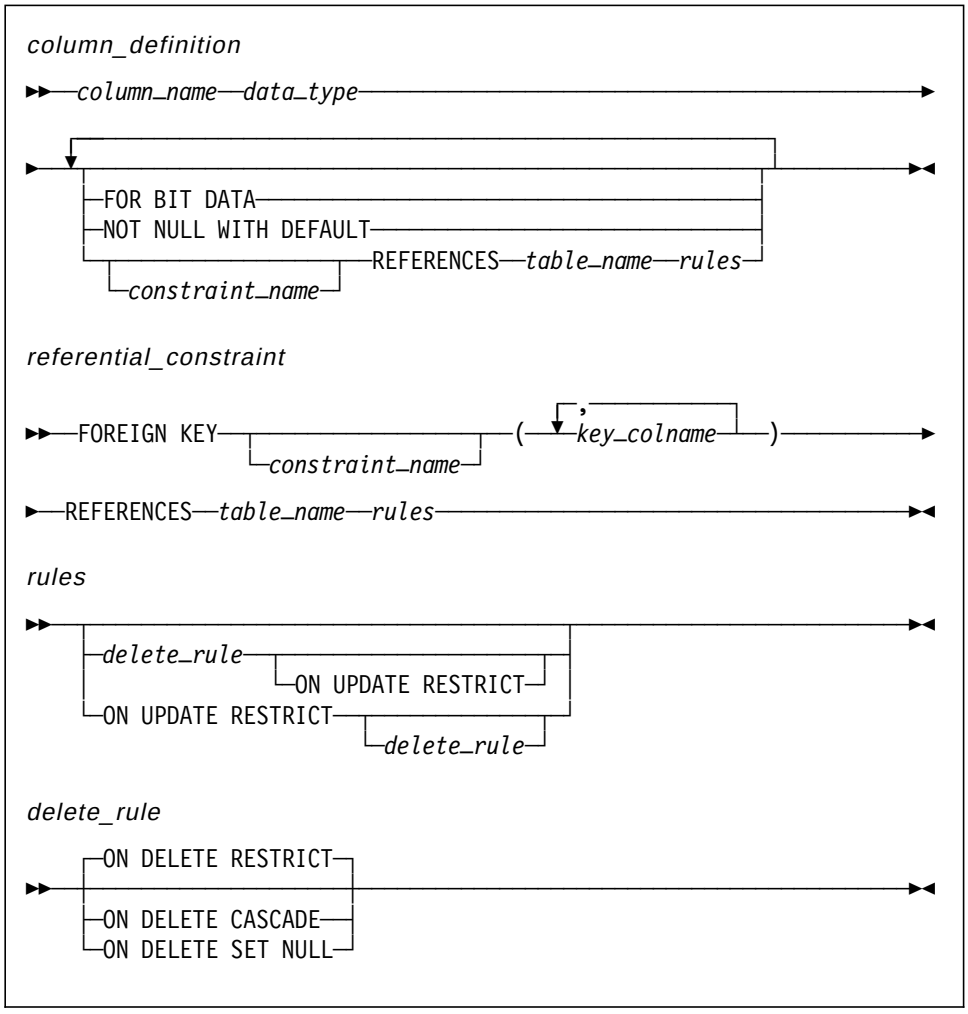


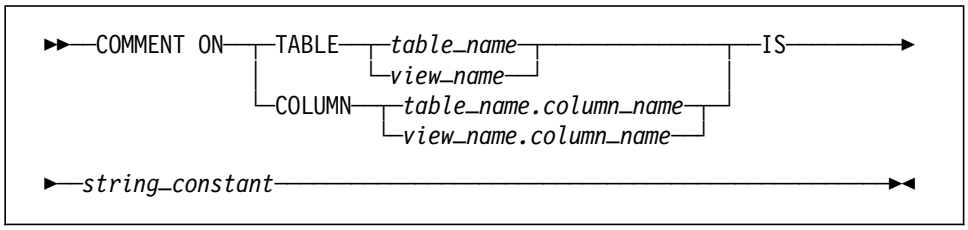
B.2.2 Dynamic REXX SQL Statements

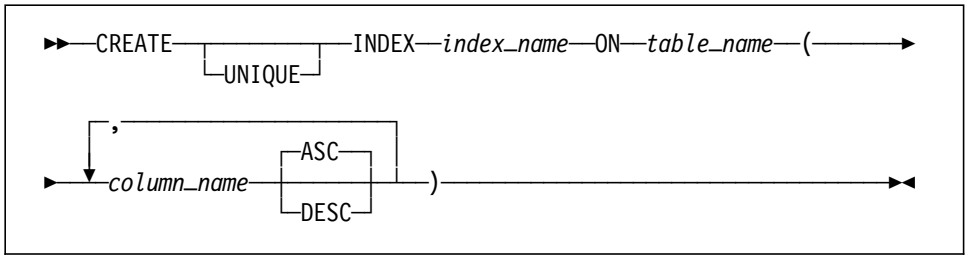
These statements cannot be used by the SQLEXEC REXX API directly. They must be dynamically prepared before they can be executed. This can be done in one of two ways:

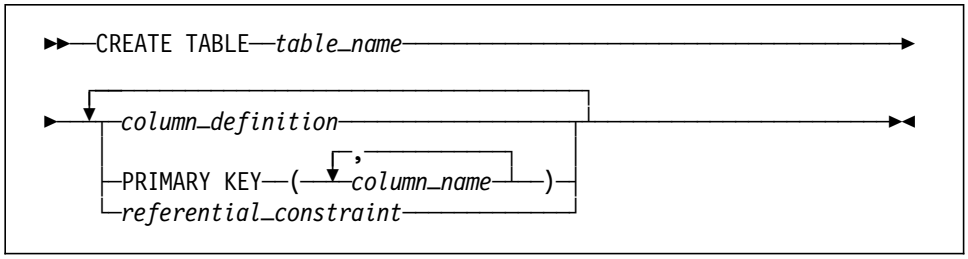
1. The statement string is used as the parameter in an EXECUTE IMMEDIATE statement.
2. The statement string is used as the parameter in a PREPARE statement and then an EXECUTE statement.


ALTER TABLE
table_name
ADD
column_definition
PRIMARY KEY
(
,
column_name
)
referential_constraint
DROP PRIMARY KEY
DROP FOREIGN KEY
constraint_name

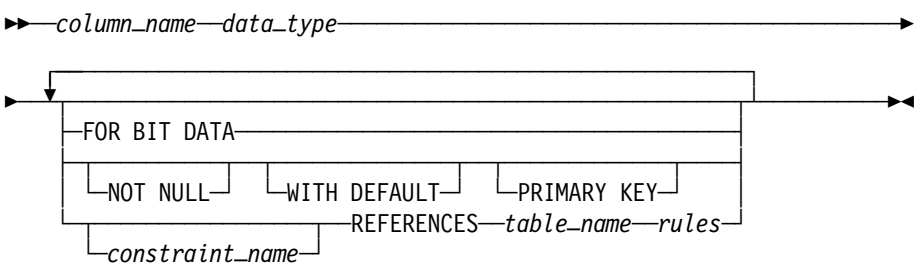

column_definition
column_name
data_type
FOR BIT DATA
NOT NULL WITH DEFAULT
REFERENCES
table_name
rules
constraint_name
referential_constraint
FOREIGN KEY
(
,
key_colname
)
constraint_name
REFERENCES
table_name
rules
rules
delete_rule
ON UPDATE RESTRICT
ON UPDATE RESTRICT
delete_rule
delete_rule
ON DELETE RESTRICT
ON DELETE CASCADE
ON DELETE SET NULL


COMMENT ON
TABLE
table_name
view_name
COLUMN
table_name.column_name
view_name.column_name
IS
string_constant

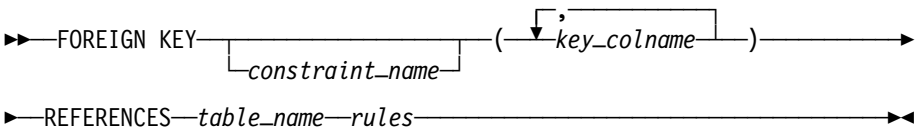

CREATE
UNIQUE
INDEX
index_name
ON
table_name
(
,
column_name
ASC
DESC
)


CREATE TABLE
table_name
,
column_definition
PRIMARY KEY
(
column_name
)
referential_constraint

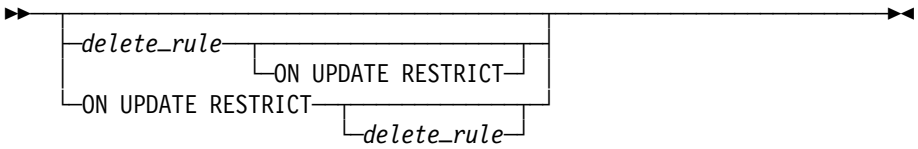
column_definition



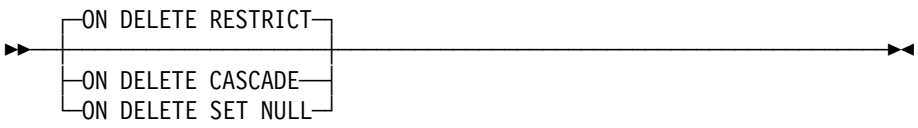
referential_constraint

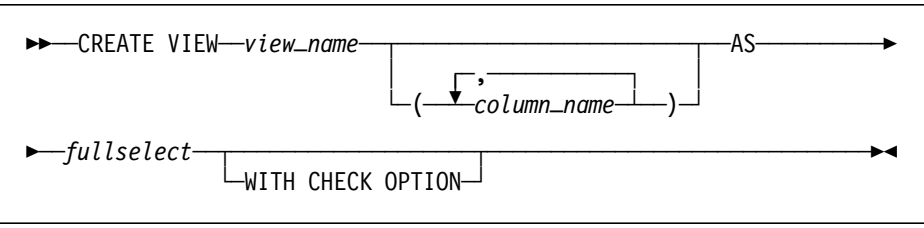


rules

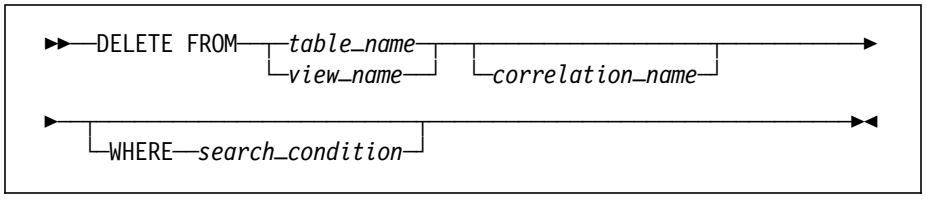


delete_rule

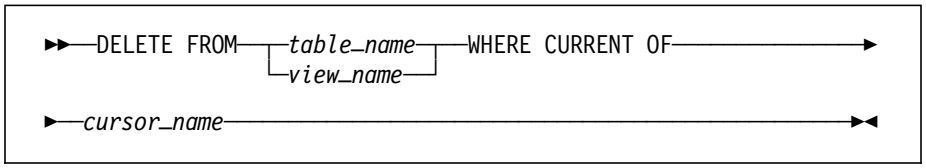


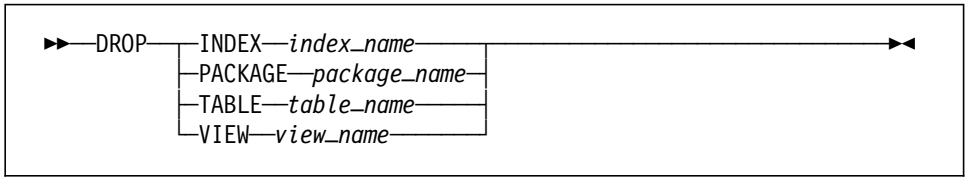


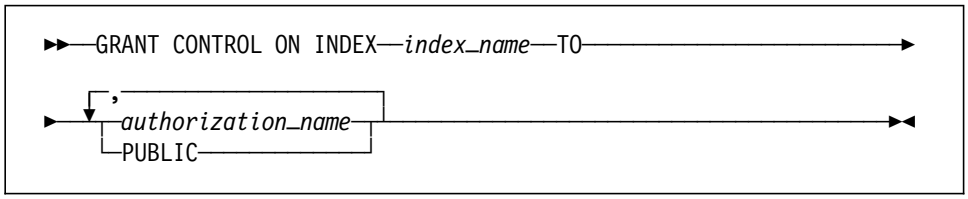
searched DELETE

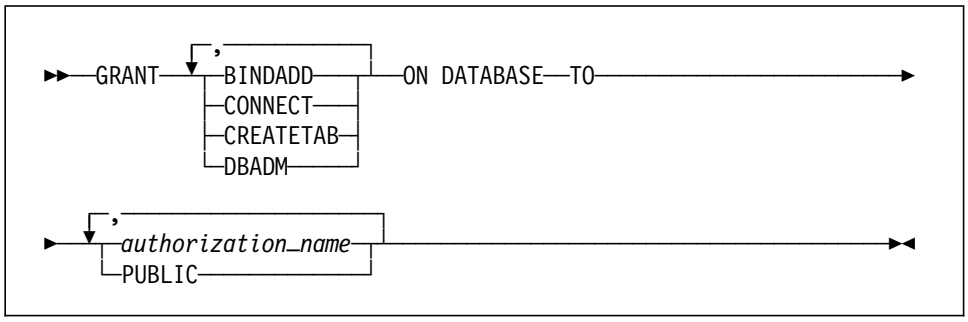


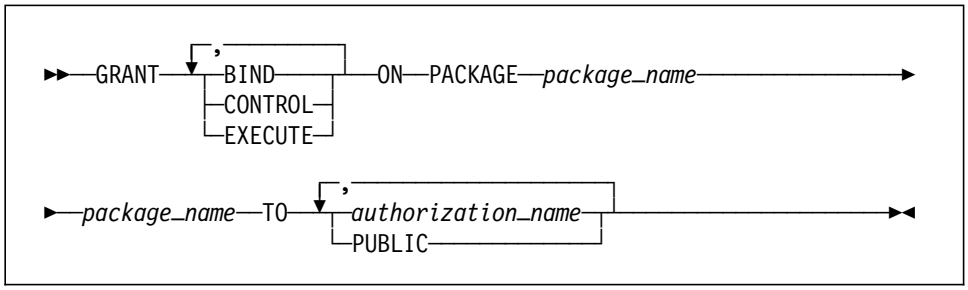
positioned DELETE

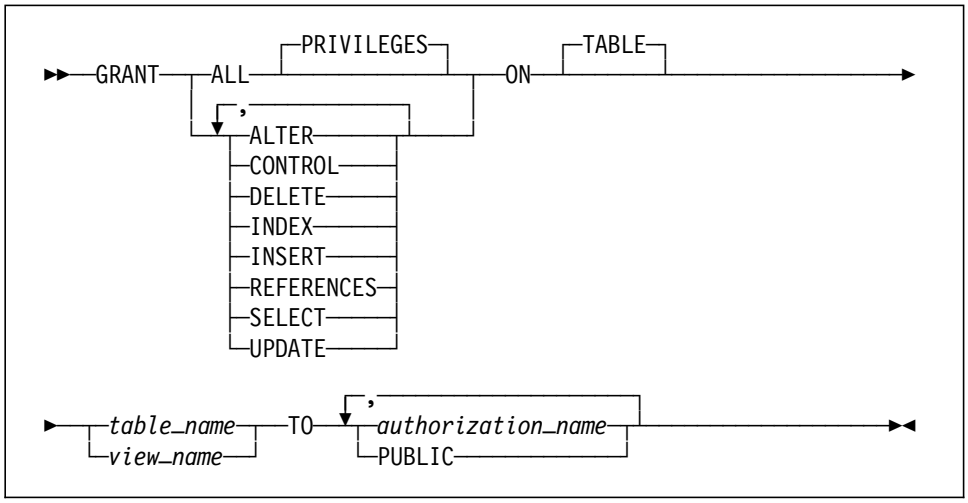


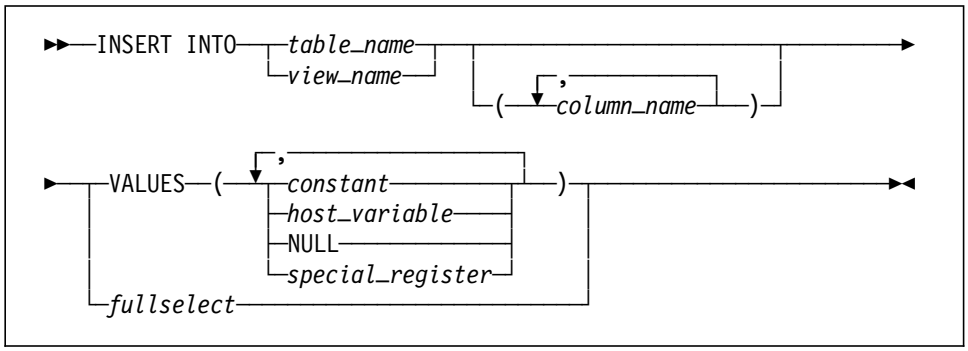


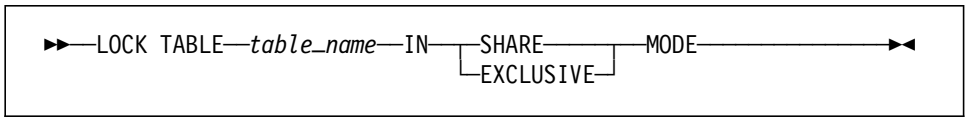


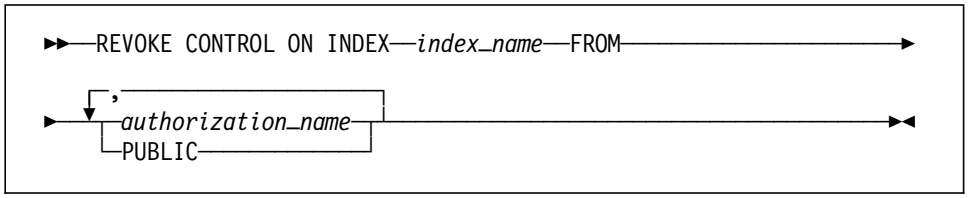

GRANT
BINDADD
CONNECT
CREATETAB
DBADM
ON DATABASE
TO
authorization_name
PUBLIC

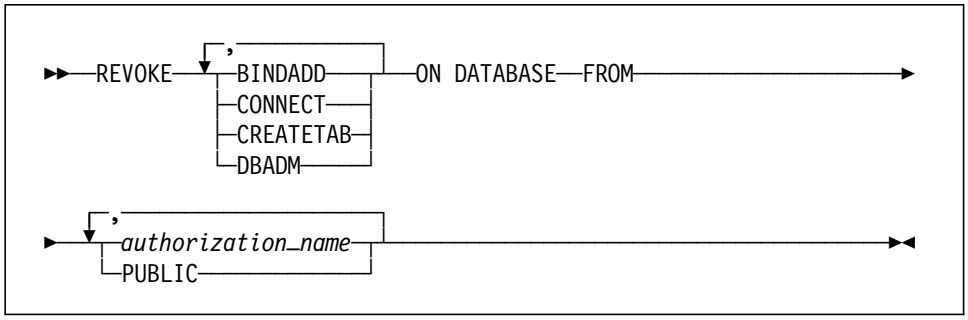

GRANT
BIND
CONTROL
EXECUTE
ON
PACKAGE
package_name
package_name
TO
authorization_name
PUBLIC

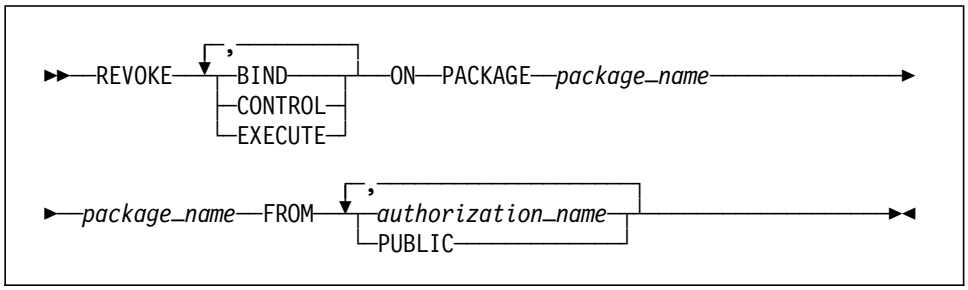

GRANT
ALL
PRIVILEGES
ALTER
CONTROL
DELETE
INDEX
INSERT
REFERENCES
SELECT
UPDATE
ON
TABLE
table_name
view_name
TO
authorization_name
PUBLIC

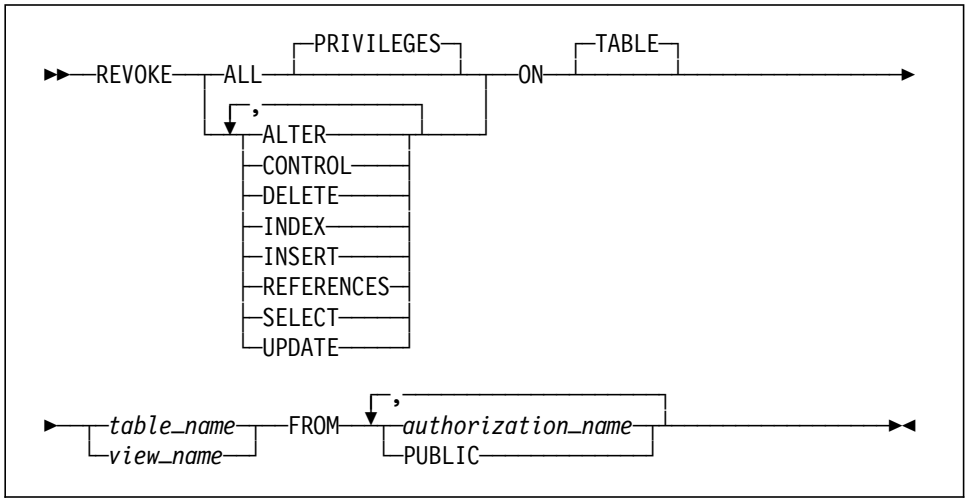

INSERT INTO
table_name
view_name
,
column_name
VALUES
constant
host_variable
NULL
special_register
fullselect

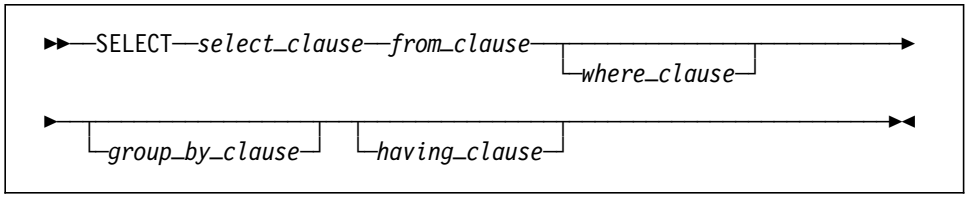

LOCK TABLE
table_name
IN
SHARE
EXCLUSIVE
MODE

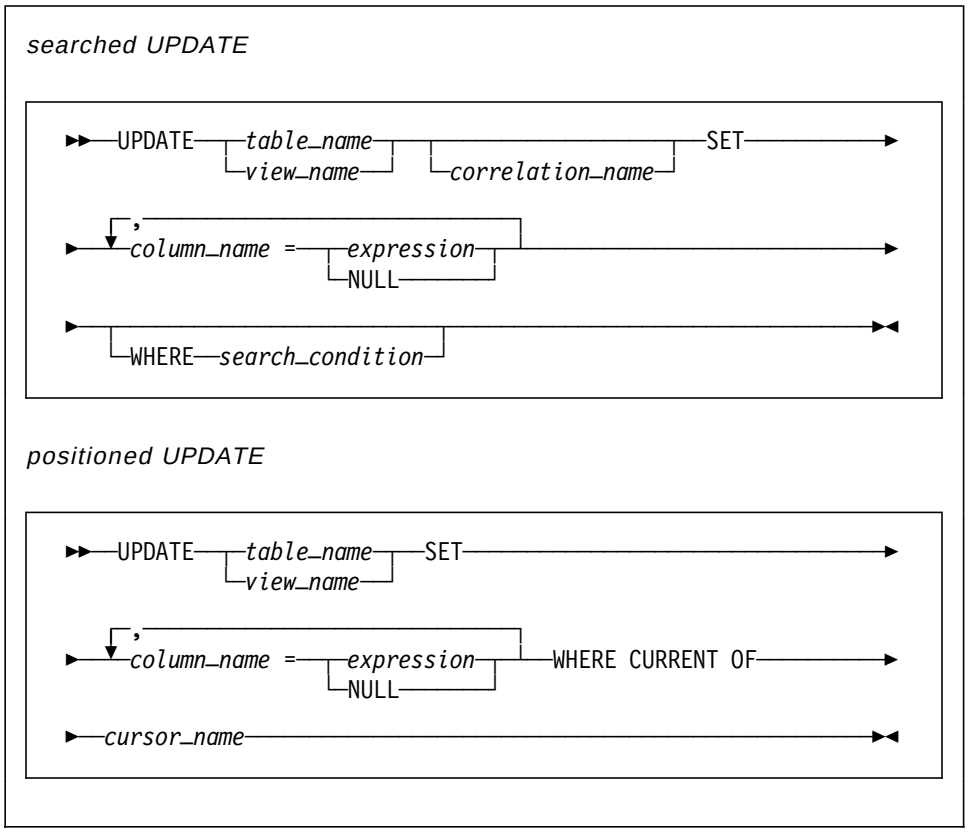

REVOKE CONTROL ON INDEX
index_name
FROM
,
authorization_name
PUBLIC


REVOKE
,
BINDADD
CONNECT
CREATETAB
DBADM
ON DATABASE
FROM
,
authorization_name
PUBLIC


REVOKE
,
BIND
CONTROL
EXECUTE
ON
PACKAGE
package_name
package_name
FROM
,
authorization_name
PUBLIC


REVOKE
ALL
PRIVILEGES
ON
TABLE
,
ALTER
CONTROL
DELETE
INDEX
INSERT
REFERENCES
SELECT
UPDATE
table_name
view_name
FROM
,
authorization_name
PUBLIC


SELECT
select_clause
from_clause
where_clause
group_by_clause
having_clause



B.3 SQL REXX Data Structures

This section lists REXX variable equivalents for data structures used to access DB2/2. In the examples, XXX represents any valid variable name, as defined by the application. In some cases, predefined occurrences of structures are supplied by DB2/2. The names of these occurrences are listed.

B.3.1 SQLCA

```
XXX.SQLCODE /* SQL return code */
XXX.SQLERRML /* Length for SQLERRMC */
XXX.SQLERRMC /* Error message tokens */
XXX.SQLERRP /* Diagnostic information */
XXX.SQLERRD.1 /* Diagnostic information */
XXX.SQLERRD.2
```

```
 .
 .
 .
XXX.SQLERRD.6
XXX.SQLWARN.0 /* Warning flags */
XXX.SQLWARN.1
 .
 .
 .
XXX.SQLWARN.10
XXX.SQLSTATE /* SQLSTATE */
```

One predefined occurrence of this structure is provided, called SQLCA. DB2/2 updates this occurrence after every executed SQL statement and DB2/2 API call.

B.3.2 SQLDA

```
XXX.SQLD /* Number of SQLVAR elements used */
XXX.1 - XXX.n /* SQLVAR element */
XXX.n.SQLTYPE /* Data type */
XXX.n.SQLLEN /* Data length */
XXX.n.SQLDATA /* Data value */
XXX.n.SQLIND /* Null indicator */
XXX.n.SQLNAME /* Column name */
```

Notes:

1. The SQLN field is not required or used.
2. If SQLTYPE is DECIMAL, SQLLEN is a compound symbol. The first number, XXX.n.SQLLEN.PRECISION, is the width of the decimal, or precision. The second, XXX.n.SQLLEN.SCALE, is the number of digits after the decimal point, or the scale.

Two predefined occurrences of this structure are provided when the Database Application Remote Interface is used. These occurrences are defined at the location of a server procedure as follows:

- SQLRIDA

  The input sqlda as passed from the client application.

- SQLRODA

  The output sqlda as passed from the client application.

B.3.3 SQLCHAR

```
XXX.LENGTH
XXX.DATA
```

One predefined occurrence of this structure is provided when the Database Application Remote Interface is used. This occurrence is defined at the location of a server procedure as follows:

- SQLRDAT

  The input data as passed from the client application.

B.3.4 SQLOPT

```
XXX.0 /* Number of options used (1 to 4) */
XXX.1 /* Date and time format (DEF, USA, EUR, ISO, JIS, LOC) */
XXX.2 /* Isolation level (RR, CS, UR) */
XXX.3 /* Blocking ( ALL, UNAMBIG, NO ) */
XXX.4 /* Authorization ID */
```

B.3.5 SQLEDINFO

```
XXX.0 /* Number of elements in the variable (always 9) */
XXX.1 /* Alias */
XXX.2 /* Database name */
XXX.3 /* Drive */
XXX.4 /* Database subdirectory */
XXX.5 /* Node name */
XXX.6 /* Release information */
XXX.7 /* Comment */
XXX.8 /* Codepage */
XXX.9 /* Entry type */
```

B.3.6 SQL_DIR_ENTRY

```
XXX.0 /* Number of fields returned (7) */
XXX.1 /* Release */
XXX.2 /* Local database name */
XXX.3 /* Target database name */
XXX.4 /* Application requester name */
XXX.5 /* Parameter string */
XXX.6 /* Comment */
XXX.7 /* Code page */
```

B.3.7 SQLENINFO

```
XXX.0 /* Number of elements (9) */
XXX.1 /* Node name */
XXX.2 /* Local LU */
XXX.3 /* Partner LU */
XXX.4 /* Mode */
XXX.5 /* Comment */
XXX.6 /* Codepage */
XXX.7 /* Protocol */
XXX.8 /* Adapter */
XXX.9 /* Network ID */
```

B.3.8 SQLESYSTAT and SQLEDBSTAT

```
XXX.0 /* Number of elements (7 or 16) */
XXX.1 /* Number of databases remaining */
XXX.2 /* Current time */
XXX.3 /* Time zone */
XXX.4 /* Product name */
XXX.5 /* Component ID */
XXX.6 /* Release level */
XXX.7 /* Last backup */
XXX.8 /* Last backup (database) */
XXX.9 /* Time zone (database) */
XXX.10 /* Number of current connects */
XXX.11 /* Database alias */
XXX.12 /* Database name */
XXX.13 /* Location */
XXX.14 /* Drive */
XXX.15 /* Node name */
XXX.16 /* Type */
```

Notes:

1. XXX.0 contains the number of fields returned. This is set to 7 for a system status request or if no databases exist. It is set to 16 if system status and database status are collected.
2. XXX.1 is set to 0 if system status only was requested. If database status was requested but no databases exist, system status is returned and this value is set to -1. If a database name was specified, system status and database status are returned and this value is set to 0. Otherwise, system status and database status are returned and this value contains the number of databases remaining that have not had their status information returned. The additional information is retrieved with the GET NEXT DATABASE STATUS BLOCK API.

B.3.9 SQLEUSRSTAT

```
XXX.0 /* Number of sets of user status returned (n) */
XXX.n.0 /* Number of elements for each user (10) */
XXX.n.1 /* Number of transactions since connect */
XXX.n.2 /* Number of requests since connect */
XXX.n.3 /* Number of requests for the current transaction */
XXX.n.4 /* Elapsed time since connect */
XXX.n.5 /* Elapsed time for the current transaction */
XXX.n.6 /* Authorization ID */
XXX.n.7 /* Node name */
XXX.n.8 /* Authority level */
XXX.n.9 /* Transaction state */
XXX.n.10 /* Lock state */
```

B.3.10 SQLDCOL

```
XXX.0 /* Number of entries in the variable (n) */
XXX.1 /* Starting location of column 1 */
XXX.2 /* Ending location of column 1 */
XXX.3 /* Starting location of column 2 */
XXX.4 /* Ending location of column 2 */
 .
 .
 .
XXX.n-1 /* Starting location of column n/2 */
XXX.n /* Ending location of column n/2 */
SQLFUPD
XXX.0 /* Number of entries in the variable (n) */
XXX.1 /* Token 1 */
XXX.2 /* Value corresponding to token 1 */
XXX.3 /* Token 2 */
XXX.4 /* Value corresponding to token 2 */
 .
 .
 .
XXX.n-1 /* Token n/2 */
XXX.n /* Value corresponding to token n/2 */
```

B.3.11 SQL_AUTHORIZATIONS

```
XXX.0 /* Number of elements (10) */
XXX.1 /* Direct SYSADM */
XXX.2 /* Direct DBADM */
XXX.3 /* Direct CREATETAB */
XXX.4 /* Direct BINDADD */
XXX.5 /* Direct SYSADM */
```

```
XXX.6 /* Indirect SYSADM */
XXX.7 /* Indirect DBADM */
XXX.8 /* Indirect CREATETAB */
XXX.9 /* Indirect BINDADD */
XXX.10 /* Indirect SYSADM */
```

Appendix C. OS/2 Workplace Shell Setup Strings and Color Definitions

We have included this appendix to give you a detailed look at the setup string parameters used in the SysCreateObject and SysSetObjectData REXXUTIL function calls for WPFolder and WPProgram objects. The MLAMBDLL.INF file on the diskette contains this information as well as sample code that utilizes REXXUTIL functions. In addition, the RGB color values for the 16 fixed colors of OS/2 2.1 are listed in this appendix.

C.1 WPFolder Setup String Parameters

WPFolder objects are a visual representation on the desktop of directories in the file system. WPFolder objects can be created with the REXXUTIL function SysCreateObject, and customized with the REXXUTIL function SysSetObjectData. Both of these function calls accept setup string parameters. This section contains a description of valid setup string parameters for WPFolder objects.

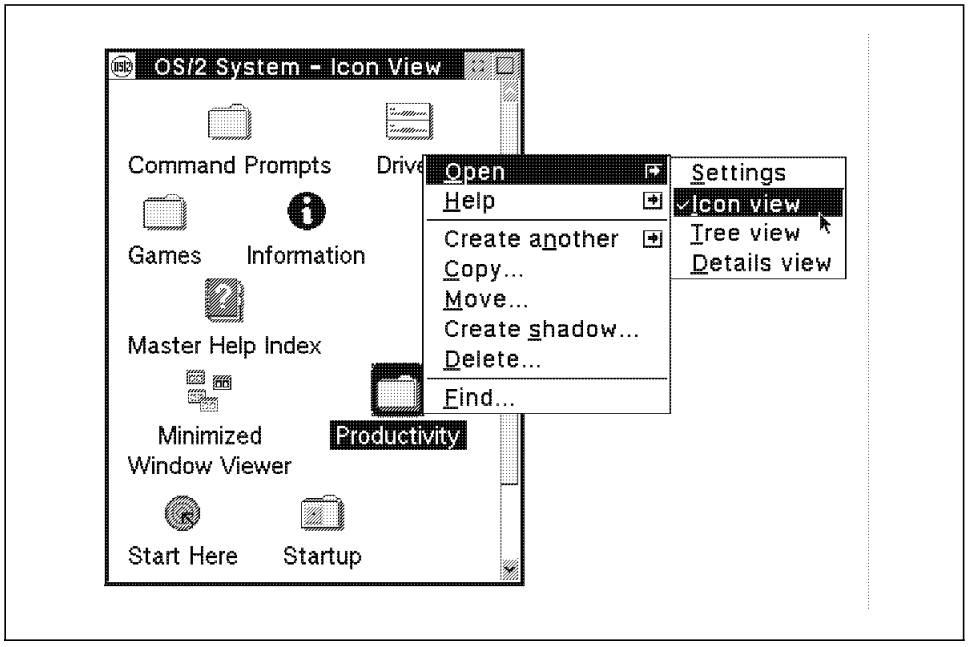


Figure 156. Open Options of a Folder

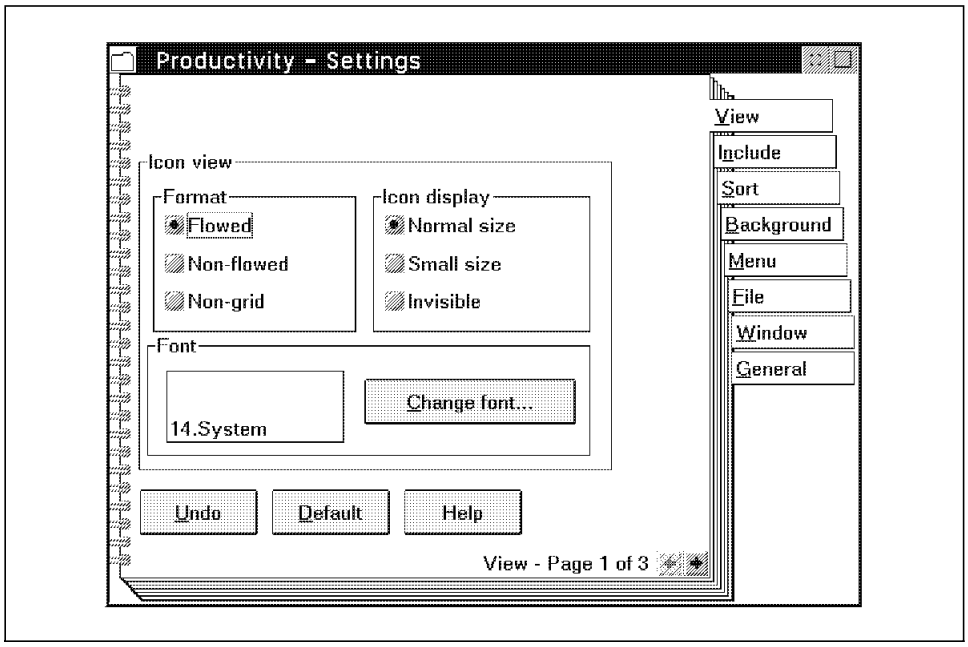


Figure 157. View Page of a Folder Settings Notebook

Table 1. WPFolder View Setup String Parameters

Keyname	Value	Description
OPEN	ICON	Open icon view when object is created/updated.
	TREE	Open tree view when object is created/updated.
	DETAILS	Open details view when object is created/updated.
ICONVIEW	s1[,s2,...sn]	Set icon view to specified style(s).
	Styles for ICONVIEW:	
	FLOWED	flowed list items
	NONFLOWED	non-flowed list items
	NONGRID	non-gridded icon view
	NORMAL	normal size icons
	MINI	small icons
	INVISIBLE	no icons
TREEVIEW	s1[,s2,...sn]	Set tree view to specified style(s).
	Styles for TREEVIEW:	
	NORMAL	normal size icons
	MINI	small icons
	INVISIBLE	no icons
	LINES	lines in tree view
	NOLINES	no lines in tree view
ICONFONT	value	Font size and facename. See Font Notes following.
TREEFONT	value	Font size and facename. See Font Notes following
DETAILSFONT	value	Font size and facename. See Font Notes following

Font Notes

The format for the value is:

```
size.facename fontstyle
```

For example, code to change the Information Folder icon view font:

```
"ICONFONT=8.Courier Bold'"
```

FaceName: Courier FontStyle: Normal

```
'ICONFONT=8.Courier'
```

FaceName: Courier FontStyle: Bold Italic

```
'TREEFONT=8.Courier Bold Italic'
```

Note that the " " are used in the .INI files but not in the .RC files.

Hint

To find out what the string should look like, create a folder, name it something simple like MYFOLD, then manually change the font size/name using the Open/Settings/Change font button. Close the settings, then from an OS/2 command line first determine the name of your desktop, for a typical 2.0 FAT file system it would be something like C:OS!2_2.0_D. (It gets easier for 2.1 where the desktop is normally named C:DESKTOP.) Then locate the folder you created C:OS!2_2.0_DMYFOLD. Then enter:

```
EAUTIL C:OS!2_2.0_DMYFOLD MYFOLD.EAS /S /P
```

This will create a MYFOLD.EAS file. Use a browse program to view this file and you'll see the values required.

C.1.1 WPFolder Background Setup String Parameters

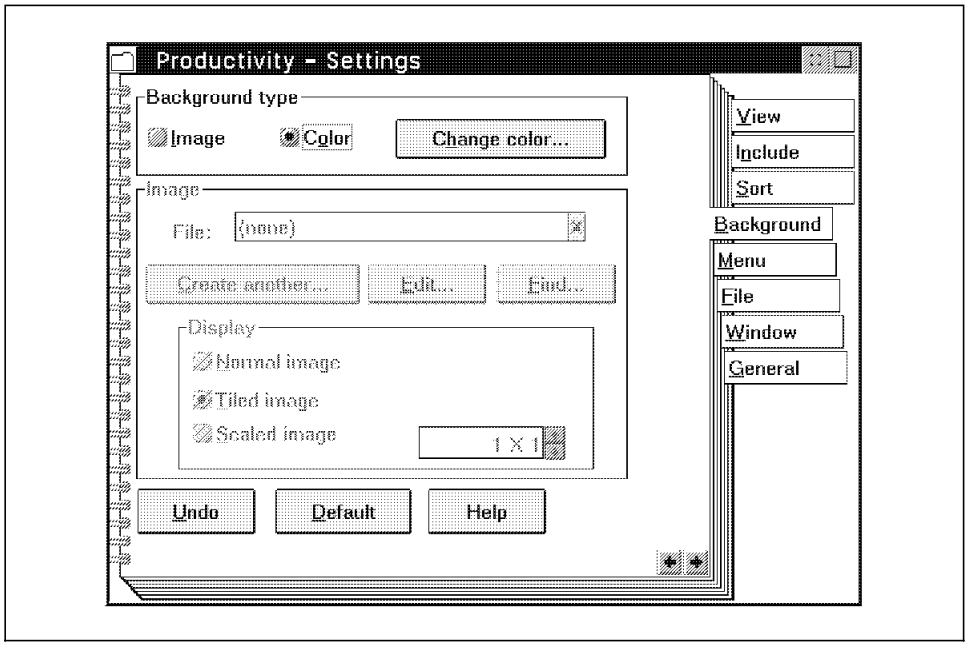


Figure 158. Background Page of a Folder Settings Notebook

Table 2. WPFolder Background Setup String Parameters

Keyname	Value	Description
BACKGROUND	filename	Sets the folder background. 'filename' is the name of a file in the OS2BITMAP directory of the boot drive.

C.1.2 WPFolder File Setup String Parameters

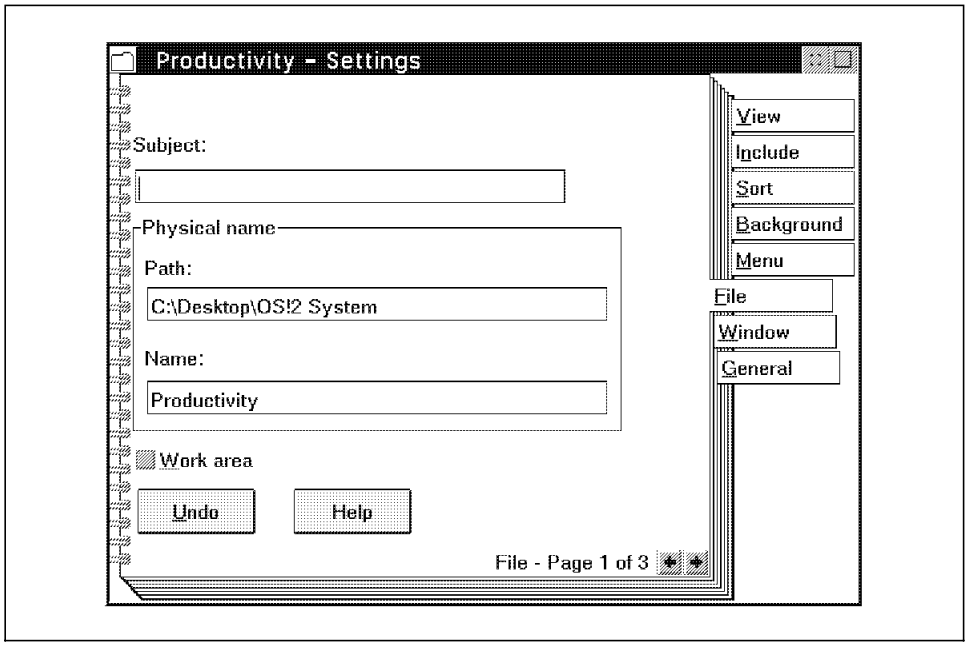


Figure 159. File Page of a Folder Settings Notebook

Table 3. WPFolder File Setup String Parameters

Keyname	Value	Description
WORKAREA	YES	Make the folder a Workarea folder
	NO	Make the folder a normal non-workarea folder.

C.1.3 WPFolder Window Setup String Parameters

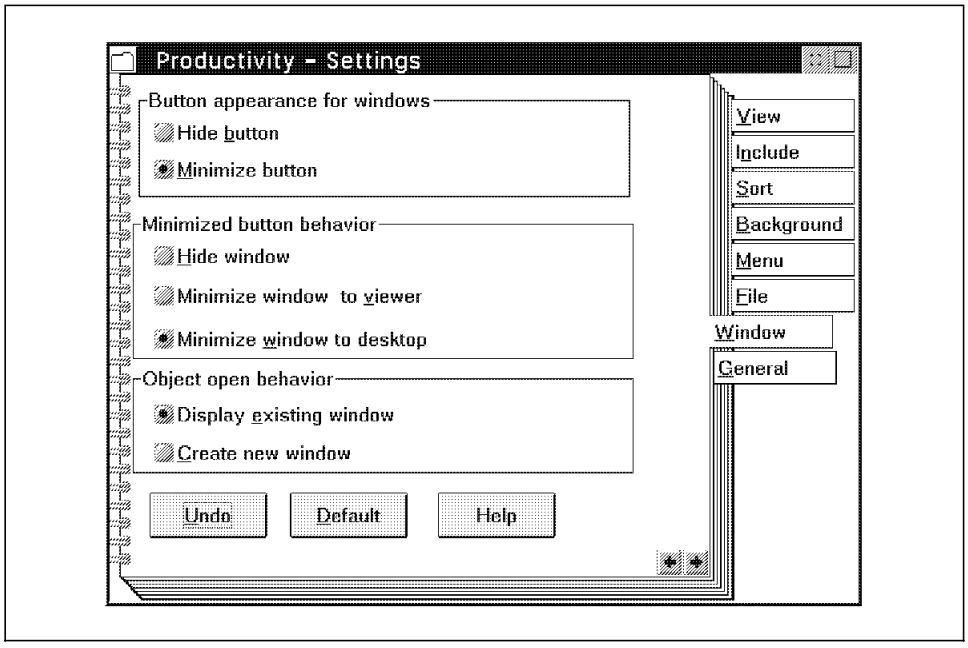


Figure 160. Window Page of a Folder Settings Notebook

Table 4. WPFolder Window Setup String Parameters

Keyname	Value	Description
MINWIN	HIDE	Views of this object will hide when their minimize button is selected.
	VIEWER	Views of this object will minimize to the minimized window viewer when their minimize button is selected.
	DESKTOP	Views of this object will minimize to the desktop when their minimize button is selected.
VIEWBUTTON	HIDE	Views of this object will have a hide button as opposed to a minimize button.
	MINIMIZE	Views of this object will have a minimize button as opposed to a hide button.
CCVIEW	YES	New views of this object will be created every time the user selects open.
	NO	Open views of this object will resurface when the user selects open.

C.1.4 WPFolder General Setup String Parameters

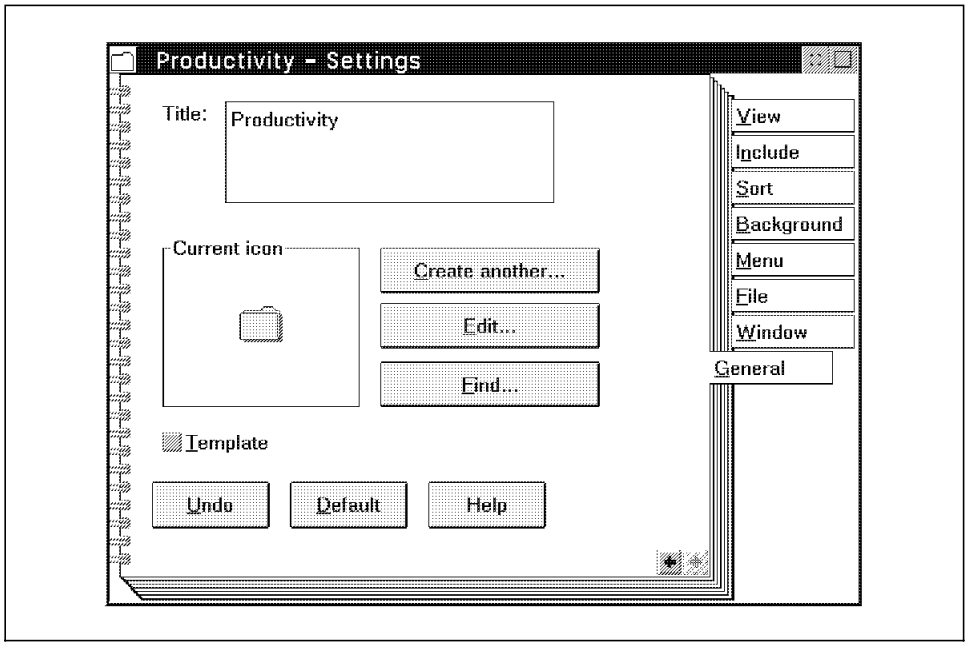


Figure 161. General Page of a Folder Settings Notebook

Table 5. WPFolder General Setup String Parameters

Keyname	Value	Description
TEMPLATE	YES	Creates object as a template.
	NO	Resets object's template property.
TITLE	value	Can be used to assign a name/title to an object.

C.1.5 WPFolder Icon Related Setup String Parameters

Table 6. WPFolder Icon Related Setup String Parameters

Keyname	Value	Description
ICONFILE	filename	This sets the object's icon.
ICONRESOURCE	id module	This sets the object's icon. 'id' is the identity of an icon resource in the 'module' dynamic link library (DLL). For example: `ICONRESOURCE=73 PMWP;` This would indicate resource 73 in PMWP.DLL.
ICONPOS	l b	This sets the object's initial icon position. The l and b values represent the position in the object's folder in percentage coordinates.
ICONVIEWPOS	l b w h	This sets the object's initial size. The values represent relative position in percentage coordinates. For example: ICONPOS=25 25 50 50 would create a folder whose bottom left corner is 25% from the left and 25% from the bottom and half the screen width/height.

C.1.6 WPFolder Miscellaneous Setup String Parameters

Table 7. WPFolder Miscellaneous Setup String Parameters

Keyname	Value	Description
OBJECTID	<name>	This sets the object's identity. The object ID will stay with the object even if it is moved or renamed. An object ID is any unique string preceded with a '<' and terminated with a '>'. This may also be a real name specified as a fully qualified path name. IMPORTANT: For any object you create you should use a unique OBJECTID! Do this for two reasons: 1. If you use an object ID it signifies a unique object that will not be recreated if you use the "FailIfExists" flag in your REXX call. Not using an object ID would cause multiple objects to be created if the same program was run over and over. 2. Should you need to later delete it or change your object, you can use this object ID in your REXX call to refer to it. Also one should not use an object ID that starts with "WP_" as many OS/2 objects use them; consider them reserved characters.
HELPPANEL	id	This sets the object's default help panel.
HELPLIBRARY	filename	This sets the help library.
OPEN	SETTINGS	Open settings view of object when created/updated.
	DEFAULT	Open default view of object when created/updated. Don't forget for folder objects you can use OPEN with these values: ICON, TREE, DETAILS

C.1.7 WPFolder Object Properties Setup String Parameters

Table 8. WPFolder Object Properties Setup String Parameters

Keyname	Value	Description
NODELETE	YES	Will not allow you to delete the object.
	NO	Resets the object's no delete property.
NOCOPY	YES	Will not allow you to make a copy.
	NO	Resets the object's no copy property.
NOMOVE	YES	Will not allow you to move the object to another folder and will create shadow on a move.
	NO	Resets the object's no move property.
NODRAG	YES	Will not allow you to drag the object.
	NO	Resets the object's no drag property.
NOLINK	YES	Will not allow you to create a shadow link.
	NO	Resets the object's no link property.
NOSHADOW	YES	Will not allow you to create a shadow link.
	NO	Resets the object's no shadow property.
NORENAME	YES	Will not allow you to rename the object.
	NO	Resets the object's no rename property.
NOPRINT	YES	Will not allow you to print it.
	NO	Resets the object's no print property.
NOTVISIBLE	YES	Will not display the object.
	NO	Resets the object's not visible property.

C.2 WPProgram Setup String Parameters

WPProgram objects are a visual representation on the desktop of program files in the file system. WPProgram objects can be created with the REXXUTIL function SysCreateObject, and customized with the REXXUTIL function SysSetObjectData. Both of these function calls accept setup string parameters. This section contains a description of valid setup string parameters for WPProgram objects.

C.2.1 WPProgram Setup String Parameters

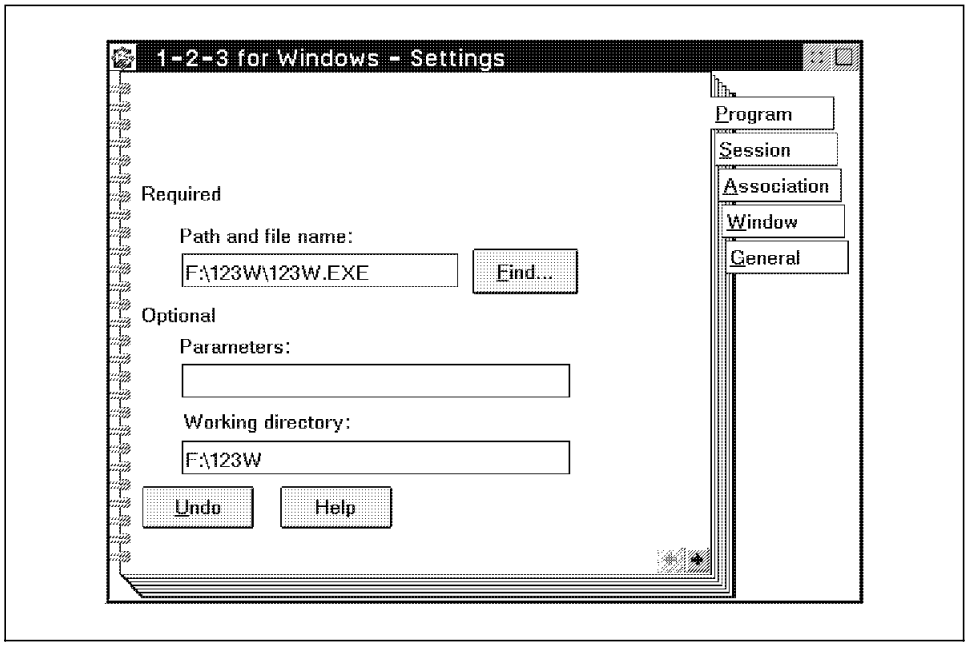


Figure 162. Program Page of a Program Settings Notebook

Table 9. WPProgram Program Setup String Parameters

Keyname	Value	Description
EXENAME	filename	Sets path and name of the program.
PARAMETERS	params	Sets the parameters list, which may include substitution characters. See the Programs Parameters Substitution characters table.
STARTUPDIR	pathname	Sets the working directory.

C.2.2 WPProgram Parameters Substitution Characters

Table 10. Program Parameters Substitution Characters

Parameter	Description
[ ]	(bracket blank bracket) You are prompted to type any parameters you want to use.
[text]	Characters placed inside of the brackets are displayed as the prompt string.
no parm	If the program object is started by clicking on it no parameters are passed. If you start the program object by dragging a file over it, the full file name is passed.
%	No parameters are passed. Useful for program objects. You may want to start from a folder′s pop-up menu.
%*	Enables you to open a data file object in one of two ways. You can drag the data file object to the program object and drop it. Or, you can open a data file object that you associated to a program.
%**P	Insert drive and path information without the last backslash ().
%**D	Insert drive with ′:′ or UNC name.
%**N	Insert file name without extension.
%**F	Insert file name with extension.
%**E	Insert extension without leading dot. In HPFS, the extension always comes after the last dot.

C.2.3 WPProgram Session Setup String Parameters

Table 11. WPProgram Session Setup String Parameters

Keyname	Value	Description
MINIMIZED	YES	Start program minimized.
MAXIMIZED	YES	Start program maximized.
NOAUTOCLOSE	YES	Leaves the window open upon program termination.
	NO	Closes the window when the program terminates.
PROGTYPE	value	Sets the session value. See Table 12 on page 285 for the possible values.
SET	XXX=VVV	XXX is any environment variable. VVV sets the value of the environment variable. When used will wipe out many variables you may assume were set. Check environment space closely when using. Also used to specify DOS settings for DOS and Windows programs. See Table 13 on page 286 for DOS and WIN-OS2 settings.

C.2.4 WPProgram Session Setup String Parameters for PROGTYPE

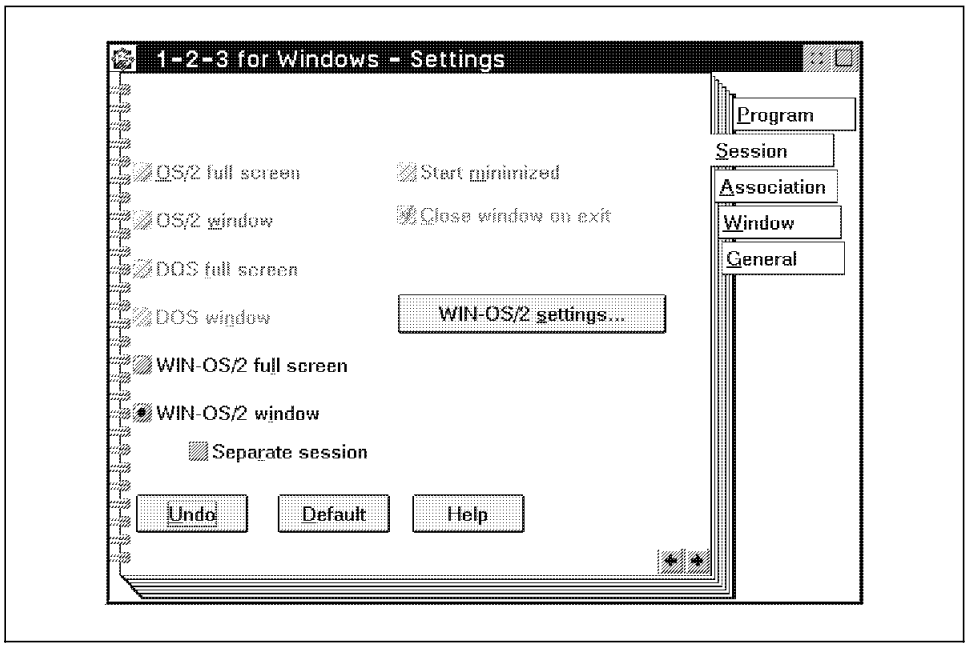


Figure 163. Session Page of a Program Settings Notebook

Table 12. WPProgram Session Setup String Parameters for PROGTYPE=	
Value	**Description**
OS/2 session values:	
PM	Sets the session type to PM
FULLSCREEN	Sets the session type to OS/2 full screen
WINDOWABLEVIO	Sets the session type to OS/2 windowed
DOS session values:	
VDM	Sets the session type to DOS full screen
WINDOWEDVDM	Sets the session type to DOS windowed
WIN-OS/2 session values:	
WIN	WIN-OS2 full screen
WINDOWEDWIN	WIN-OS2 windowed, NOT a separate VDM session
SEPARATEWIN	WIN-OS2 windowed, Separate VDM session
OS/2 2.1 systems session values:	
PROG_31_STD	WIN-OS2 full screen, Windows 3.1 Standard mode.
PROG_31_STDSEAMLESSVDM	WIN-OS2 windowed, Separate VDM session, 3.1 Standard mode
PROG_31_STDSEAMLESSCOMMON	WIN-OS2 windowed, NOT a separate VDM session, 3.1 Standard mode
PROG_31_ENH	WIN-OS/2 full screen, NOT a separate VDM session, 3.1 Enhanced Compatibility
PROG_31_ENHSEAMLESSVDM	WIN-OS2 windowed, Separate VDM session, 3.1 Enhanced Compatibility
PROG_31_ENHSEAMLESSCOMMON	WIN-OS2 windowed, NOT a separate VDM session, 3.1 Enhanced Compatibility

C.2.5 WPProgram DOS and WIN-OS2 Settings

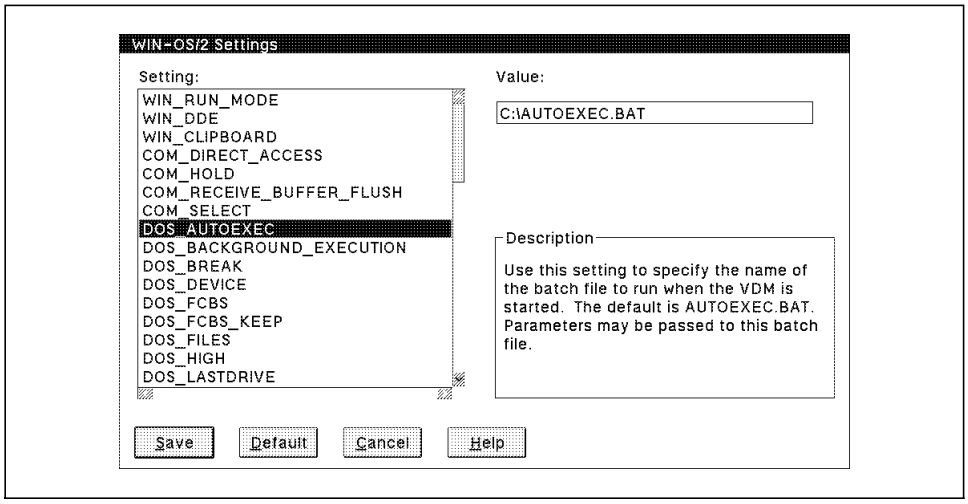


Figure 164. Settings Dialog on the Session Page of a Program Settings Notebook

Table 13 (Page 1 of 5). DOS and WIN-OS2 Settings Fields <default>

Keyname	**Value**
WIN_RUNMODE	Use the PROGTYPE parameter mentioned previously to define a enhanced mode WIN-OS2 program (Also see note 4 from above)
WIN_DDE	(See note 4 above)
WIN_CLIPBOARD	(See note 4 above)
AUDIO_ADAPTER_SHARING	1 0
COM_DIRECT_ACCESS	1 < 0 >
COM_HOLD	1 < 0 >
COM_RECEIVE_BUFFER_FLUSH	Valid settings: <NONE> ALL RECEIVE DATA INTERRUPT ENABLE SWITCH TO FOREGROUND
COM_SELECT	Valid settings: <ALL> COM1 COM2 COM3 COM4 NONE
DOS_AUTOEXEC	C:AUTOEXEC.BAT Use full BATch file name also you can pass parameters

Table 13 (Page 2 of 5). DOS and WIN-OS2 Settings Fields <default>	
Keyname	**Value**
DOS_BACKGROUND_EXECUTION	< 1 > 0
DOS_BREAK	1 < 0 >
DOS_DEVICE	Default: empty Remember to separate any used with "," for new line
DOS_FCBS	Limits: 0-255, default 16
DOS_FCBS_KEEP	Limits: 0-255, default 8
DOS_FILES	Limits: 20-255, default 20
DOS_HIGH	1 < 0 >
DOS_LASTDRIVE	Limits: last physical drive to Z, default Z
DOS_RMSIZE	Limits: 128-640, default 640, increments of 16
DOS_SHELL	Default: ″?:OS2MDOSCOMMAND.COM ″ ″?:OS2MDOS /P″ where ? is the boot drive
DOS_STARTUP_DRIVE	Default: empty Accepts text; like A: or C:DISKSDRDOS.IMG
DOS_UMB	1 < 0 >
DOS_VERSION	Default: DCJSS02.EXE,3,40,255 DFIA0MOD.SYS,3,40,255 DXMA0MOD.SYS,3,40,255 IBMCACHE.COM,3,40,255 IBMCACHE.SYS,3,40,255 ISAM.EXE,3,40,255 ISAM2.EXE,3,40,255 ISQL.EXE,3,40,255 NET3.COM,3,40,255 EXCEL.EXE,10,10,4 PSCPG.COM,3,40,255 SAF.EXE,3,40,255 WIN200.BIN,10,10,4 Remember what you put here will replace the existing list of items so be careful, also remember to use carets in front of any commas you need. Example: SET DOS_VERSION=IBMCACHE.SYS^,3^,40^,255

Table 13 (Page 3 of 5). DOS and WIN-OS2 Settings Fields <default>	
Keyname	**Value**
DPMI_DOS_API	Valid settings: <AUTO> ENABLED DISABLED
DPMI_MEMORY_LIMIT	Limits: 0-512, default 4
DPMI_NETWORK_BUFF_SIZE	Limits: 1-64, default 8
EMS_FRAME_LOCATION	Valid settings: <AUTO> NONE C000 C400 C800 CC00 D000 D400 D800 DC00 8000 8400 8800 8C00 9000
EMS_HIGH_OS_MAP_REGION	Limits: 0-96, default 32, note increments of 16
EMS_LOW_OS_MAP_REGION	Limits: 0-576, default 384, note increments of 16
EMS_MEMORY_LIMIT	Limits: 0-32768, default 2048, note increments of 16
HW_NOSOUND	1 < 0 >
HW_ROM_TO_RAM	1 < 0 >
HW_TIMER	1 < 0 >
IDLE_SECONDS	Limits: 0-60, default 0
IDLE_SENSITIVITY	Limits: 1-100, default 75
INT_DURING_IO	1 < 0 >
KBD_ALTHOME_BYPASS	1 < 0 >
KBD_BUFFER_EXTEND	< 1 > 0
KBD_CTRL_BYPASS	Valid settings: <NONE> ALT_ESC CTRL_ESC
KBD_RATE_LOCK	1 < 0 >
MEM_EXCLUDE_REGIONS	Initially empty, you can specify a range of memory to exclude or you can supply a single address for the beginning of a 4KB region, if you need several regions separate them with a comma (don't forget to use the caret since commas are special setup string parameters) Example: `SET MEM_EXCLUDE_REGIONS=C0000, D0000-D8000`

Table 13 (Page 4 of 5). DOS and WIN-OS2 Settings Fields <default>	
Keyname	**Value**
MEM_INCLUDE_REGIONS	Initially empty, you can specify a range of memory to include or you can supply a single address for the beginning of a 4KB region, if you need several regions separate them with a comma (don't forget to use the caret since commas are special setup string parameters) Example: 'SET MEM_INCLUDE_REGIONS=C0000, D0000-D7FFF NOTE: The include region D0000-D8000 will include the entire memory between D8000 and D8FFFF.
MOUSE_EXCLUSIVE_ACCESS	1 < 0 >
NETWARE_RESOURCES	Valid settings: NONE PRIVATE GLOBAL Special note, you use the words to change the value BUT the string MUST be 7 characters long! Example: SET NETWARE_RESOURCES=GLOBAL
PRINT_SEPARATE_OUTPUT	< 1 > 0
PRINT_TIMEOUT	Limits: 0-3600, default 15
TOUCH_EXCLUSIVE_ACCESS	1 0
VIDEO_8514A_XGA_IOTRAP	< 1 > 0
VIDEO_FASTPASTE	1 < 0 >
VIDEO_MODE_RESTRICTION	Valid settings: <NONE> CGA MONO Special note, you use the words to change the value BUT the string MUST be 15 characters long! Example: SET VIDEO_MODE_RESTRICTION=CGA
VIDEO_ONDEMAND_MEMORY	< 1 > 0
VIDEO_RETRACE_EMULATION	< 1 > 0
VIDEO_ROM_EMULATION	< 1 > 0
VIDEO_SWITCH_NOTIFICATION	1 < 0 >
VIDEO_WINDOW_REFRESH	Limits: 1-600, default 1
XMS_HANDLES	Limits: 0-128, default 32

Table 13 (Page 5 of 5). DOS and WIN-OS2 Settings Fields <default>	
Keyname	**Value**
XMS_MEMORY_LIMIT	Limits: 0-16384, default 2048, increment of 4
XMS_MINIMUM_HMA	Limits: 0-63, default 0

Important Notes on DOS and WIN-OS2 Settings

- To change these values you use: keyname=value. For example:

  ```
  SET DOS_FILES=45;SET DOS_HIGH=1;
  ```

  Also note that on the settings page you click on ON or OFF for some values. From an .RC file you use 1 for ON, 0 for OFF. For example:

  ```
  SET COM_HOLD=1;
  ```

  sets to ON to keep the communications ports open until the session ends.
- Some settings may already have default values, like DOS_VERSION. You must be careful since any action against that setting is treated as a replacement (even if you are using the updateifexist duplicate flag value). So if you wanted to add one item to DOS_VERSION, you should also include all of the existing values.
- Some settings are new once you've installed the OS/2 V2 Service Pack or upgraded to OS/2 V2.1. As well some may not be on your workstation due to your hardware configuration, for instance use of VIDEO_8514A_XGA_IOTRAP is only available on certain systems.
- WIN-OS2 Settings are new to V2 users and appear once the Service Pack is installed or you have upgraded to OS/2 V2.1.

C.2.6 WPProgram Association Setup String Parameters

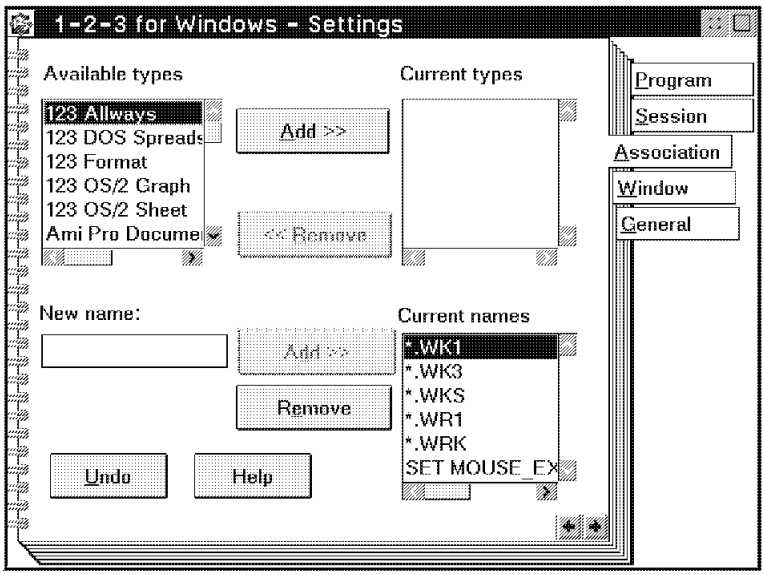


Figure 165. Association Page of a Program Settings Notebook

Table 14. WPProgram Association Setup String Parameters

Keyname	Value	Description
ASSOCFILTER	filters	Sets the filename filter for files associated to this program. Multiple filters are separated by commas. See notes about preserving existing filter values below.
ASSOCTYPE	type	Sets the type of files associated to this program. Multiple filters are separated by commas. See notes about preserving existing associate types below.

Preserving existing values

When using ASSOCFILTER and/or ASSOCTYPE include two commas at the end of your string to preserve any settings that are already applied. For example:

```
ASSOCTYPE=Metafile,PIF file,,;ASSOCFILTER=*.MET,*.PIF,,; ...
```

C.2.7 WPProgram Window Setup String Parameters

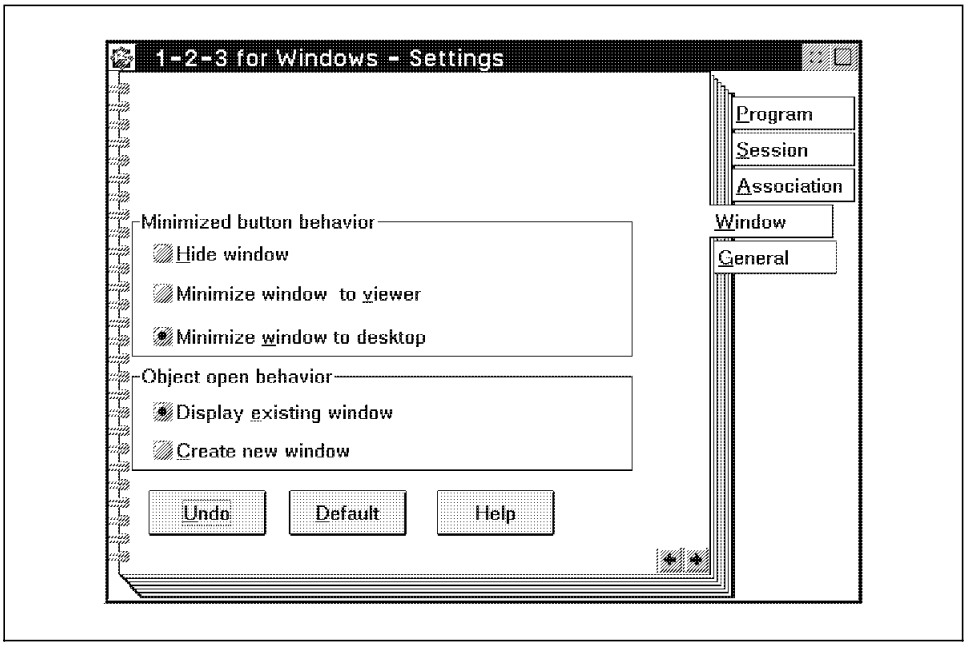


Figure 166. Window Page of a Program Settings Notebook

Table 15. WPProgram Window Setup String Parameters

Keyname	Value	Description
MINWIN	HIDE	Views of this object will hide when their minimize button is selected.
	VIEWER	Views of this object will minimize to the minimized window viewer when their minimize button is selected.
	DESKTOP	Views of this object will minimize to the Desktop when their minimize button is selected.
CCVIEW	YES	New views of this object will be created every time the user selects open.
	NO	Open views of this object will resurface when the user selects open.

C.3 RGB Values for Fixed Colors of OS/2 2.1

Table 16. RGB Values for the 16 Fixed Colors of OS/2 2.1

Color	R (Red)	G (Green)	B (Blue)
Black	0	0	0
Blue	0	0	255
Brown	128	128	0
Cyan	0	255	255
Dark Blue	0	0	128
Dark Cyan	0	128	128
Dark Gray	128	128	128
Dark Green	0	128	0
Dark Pink	128	0	128
Dark Red	128	0	0
Green	0	255	0
Intense White	255	255	255
Light Gray	204	204	204
Pink	255	0	255
Red	255	0	0
Yellow	255	255	0

Appendix D. CM/2 REXX EHLLAPI Reference

We have included this appendix to give you a quick reference to the syntax of the REXX EHLLAPI function calls and the EHLLAPI keyboard mnemonics. For more information on EHLLAPI, refer to *IBM Communications Manager/2 Version 1.0 EHLLAPI Programming Reference*.

D.1 REXX EHLLAPI Functions

This section contains the syntax for calling the EHLLAPI functions from REXX programs. It is assumed that the HLLAPISRV function in the SAAHLAPI.DLL is registered in the calling REXX program as HLLAPI. Some functions may require prerequisite function calls. For example, you need to issue a Connect function call before issuing a Search_PS function call.

```
HLLAPI('Change_Switch_Name', session_id, type [, new_name ])
```

```
HLLAPI('Change_window_name', session_id, type [, new_name ])
```

```
HLLAPI('Connect', session_id )
```

```
HLLAPI('Connect_PM', session_id )
```

```
HLLAPI('Convert_pos', session_id, column | position [, row ] )
```

```
HLLAPI('Copy_field_to_str', pos, length )
```

```
HLLAPI('Copy_OIA')
```

```
HLLAPI('Copy_PS')
```

```
HLLAPI('Copy_PS_to_str', pos, length)
```

```
HLLAPI('Copy_str_to_field', string, pos)
```

```
HLLAPI('Copy_str_to_PS', string, pos)
```

```
HLLAPI('Disconnect')
```

```
HLLAPI('Disconnect_PM', session_id)
```

```
HLLAPI('Find_field_len', search_option, pos)
```

```
HLLAPI('Find_field_pos', search_option, pos)
```

```
HLLAPI('Get_key', session_id )
```

```
HLLAPI('Get_window_status', session_id)
```

```
HLLAPI('Intercept_status', session_id, status )
```

```
HLLAPI('Lock_PMSVC', session_id, status, queue_option )
```

```
HLLAPI('Lock_PS', session_id, status, queue_option )
```

```
HLLAPI('Pause', n [, sessname] )
```

```
HLLAPI('Query_close_intercept', session_id )
```

```
HLLAPI('Query_cursor_pos' )
```

```
HLLAPI('Query_field_attr', pos )
```

```
HLLAPI('Query_host_update', session_id )
```

```
HLLAPI('Query_session_status', session_id )
```

```
HLLAPI('Query_sessions' )
```

```
HLLAPI('Query_system' )
```

```
HLLAPI('Query_window_coord', session_id )
```

```
HLLAPI('Receive_file', string )
```

```
HLLAPI('Release' )
```

```
HLLAPI('Reserve' )
```

```
HLLAPI('Reset_system' )
```

```
HLLAPI('Search_field', string, pos )
```

```
HLLAPI('Search_PS', string, pos )
```

```
HLLAPI('Send_file', string )
```

```
HLLAPI('Sendkey', string )
```

```
HLLAPI('Set_cursor_pos', pos )
```

```
HLLAPI('Set_session_parms', string )
```

```
HLLAPI('Set_window_status', session_id, option [, num1 | option1, num2 ] )
```

```
HLLAPI('Start_close_intercept', session_id )
```

```
HLLAPI('Start_host_notify', session_id, option )
```

```
HLLAPI('Start_keystroke_intercept', session_id, option )
```

```
HLLAPI('Stop_close_intercept', session_id )
```

```
HLLAPI('Stop_host_notify', session_id )
```

```
HLLAPI('Stop_keystroke_intercept', session_id )
```

```
HLLAPI('Wait' )
```

D.2 Keyboard Mnemonics

Keyboard mnemonics provide ASCII characters that represent the special function keys of the personal computer keyboard. The tables in this section contain the keyboard mnemonics for use with EHLLAPI.

Table 17. Mnemonics with Uppercase Alphabetic Characters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@B	Left Tab	X	X
@C	Clear	X	X
@D	Delete	X	X
@E	Enter	X	X
@F	Erase EOF	X	X
@H	Help		X
@I	Insert	X	X
@J	Jump (Set focus under PM)	X	X
@L	Cursor Left	X	X
@N	New Line	X	X
@O	Space		X
@P	Print	X	X
@R	Reset	X	X
@T	Right Tab	X	X
@U	Cursor Up	X	X
@V	Cursor Down	X	X
@X	DBCS (Reserved)	X	X
@Y	Caps Lock (No action)	X	X
@Z	Cursor Right	X	X

Table 18 (Page 1 of 2). Mnemonics with Lowercase Numbers or Letters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@0	Home	X	X
@1	PF1/F1	X	X
@2	PF2/F2	X	X
@3	PF3/F3	X	X
@4	PF4/F4	X	X
@5	PF5/F5	X	X

Table 18 (Page 2 of 2). Mnemonics with Lowercase Numbers or Letters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@6	PF6/F6	X	X
@7	PF7/F7	X	X
@8	PF8/F8	X	X
@9	PF9/F9	X	X
@a	PF10/F10	X	X
@b	PF11/F11	X	X
@c	PF12/F12	X	X
@d	PF13	X	X
@e	PF14	X	X
@f	PF15	X	X
@g	PF16	X	X
@h	PF17	X	X
@i	PF18	X	X
@j	PF19	X	X
@k	PF20	X	X
@l	PF21	X	X
@m	PF22	X	X
@n	PF23	X	X
@o	PF24	X	X
@p	Plus Key		X
@q	End	X	X
@s	Scrlk (No action)	X	X
@t	Num lock (No action)	X	X
@u	Page Up		X
@v	Page Down		X
@x	PA1	X	X
@y	PA2	X	X
@z	PA3	X	X

Table 19. Mnemonics with @A and @ Uppercase Alphabetic Characters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@A@C	Test		X
@A@D	Word Delete	X	X
@A@E	Field Exit	X	X
@A@F	Erase Input	X	X
@A@H	System Request	X	X
@A@I	Inset Toggle	X	X
@A@J	Cursor Select	X	X
@A@L	Cursor Left Fast	X	X
@A@N	Get Cursor	X	X
@A@O	Locate Cursor	X	X
@A@Q	Attention	X	X
@A@R	Device Cancel (Cancels Print Presentation Space)	X	X
@A@T	Print Presentation Space	X	X
@A@U	Cursor Up Fast	X	X
@A@V	Cursor Down Fast	X	X
@A@X	Hexadecimal		X
@A@Y	Cmd (Function) Key		X
@A@Z	Cursor Right Fast	X	X

Table 20. Mnemonics with @A and @ Lowercase Alphabetic Characters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@A@9	Reverse Video	X	X
@A@a	Destructive Backspace	X	X
@A@b	Underscore	X	X
@A@c	Reset Reverse Video	X	X
@A@d	Red	X	X
@A@e	Pink	X	X
@A@f	Green	X	X
@A@g	Yellow	X	X
@A@h	Blue	X	X
@A@i	Turqoise	X	X
@A@j	White	X	X
@A@l	Reset Host Colors	X	X
@A@n	Go directly to Session 1		
@A@o	Go directly to Session 2		
@A@p	Go directly to Session 3		
@A@q	Go directly to Session 4		
@A@r	Go directly to Session 5		
@A@t	Print (personal computer)	X	X
@A@y	Forward Word Tab	X	X
@A@z	Backward Word Tab	X	X

Table 21. Mnemonics with @A and @ Alphanumeric (Special) Characters

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@A@-	Field -		X
@A@+	Field +	X	X
@A@<	Record Backspace		X

Table 22. ASCII Mnemonics Using Data Keys and Combinations of Shift (@S) and @ Uppercase Alpha Keys

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@S@A	Erase EOL	X	X
@S@B	Field Advance	X	X
@S@C	Field Backspace	X	X
@S@D	Valid Character Backspace	X	X
@S@E	Print Presentation Space on Host		X
@S@J	ChgScr		
@S@K	Auto		
@S@P	POR	X	X
@S@T	Jump to Task Manager	X	X
@S@l	Shift Lock		
@S@x	Dup	X	X
@S@y	Field Mark	X	X
@r@s	Break		
@r@t	Pause		

Table 23. Alphabetic Keys

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
a - z	a - z	X	X
A - Z	A - Z	X	X
0 - 9	0 - 9	X	X
-	-	X	X
#	#	X	X
!	!	X	X
$	$	X	X
%	%	X	X
&	&	X	X
″	″	X	X
′	′	X	X
(	(	X	X
)	)	X	X
*	*	X	X
+	+	X	X
.	.	X	X
/	/	X	X
:	:	X	X
;	;	X	X
<	<	X	X
>	>	X	X
=	=	X	X
?	?	X	X
{	{	X	X
}	}	X	X
[	[	X	X
]	]	X	X
|	|	X	X

Table 24. Mnemonics with Special Character Keys

ASCII Mnemonic	Meaning	Supported by 3270 Emulation	Supported by 5250 Emulation
@@	@	X	X
@/	Overrun of queue (Only in the **GET_KEY** function)	X	X
@$	Alternate cursor (the Presentation Manager interface only)	X	X
@ <	Backspace	X	X

Appendix E. Published Books, Manuals, and Papers on REXX

This is a list of published (that is, generally available) books, manuals, and papers from referred journals that are closely associated with the REXX language or one of its implementations.

This list is in three parts, each ordered roughly chronologically by first edition:

1. IBM Manuals, etc., with IBM order numbers.
2. Other books and manuals.
3. Papers.

E.1 Books and IBM Manuals Available Through Usual IBM Channels

E.1.1 Cross-system books

ZB35-5100 The REXX Language, 2nd Ed. -- Cowlishaw

SC26-4358 SAA CPI:Procedures Language Reference (Level 1)

SC24-5549 SAA CPI: REXX Level 2 Reference

G511-1430 IBM REXX Compiler and Library/370:
Introducing the Next Step in REXX <CMS, MVS>

SH19-8160 IBM REXX Compiler and Library/370:
User's Guide and Reference <CMS, MVS>

LY19-6264 IBM REXX Compiler and Library/370:
Diagnosis Guide <CMS, MVS>

SB20-0020 The REXX Handbook - Ed. Goldberg & Smith

E.1.2 VM

SC24-5239 VM/SP: System Product Interpreter Reference

SC24-5238 VM/SP: System Product Interpreter User's Guide

SX24-5126 VM/SP: System Product Interpreter Reference Summary

SB09-1326 VM/SP: System Product Interpreter Reference (Chinese)

SB09-1325 VM/SP: System Product Interpreter User's Guide (Chinese)

SB09-9361 The System Product Interpreter (REXX) Examples and Techniques - Brodock

SC12-1599 VM/SP: System Product Interpreter Handbuch (German: SC24-5239, July 1984)

SC24-5357 VM/IS: Writing Simple Programs with REXX

SC23-0374 VM/XA: System Product Interpreter Reference

SC23-0375 VM/XA: System Product Interpreter User's Guide

GH19-8118 CMS REXX Compiler General Information

SH19-8120 CMS REXX Compiler User's Guide & Reference

LY19-6262 CMS REXX Compiler Diagnosis Guide

LN19-9048 CMS REXX Compiler Diagnosis Guide Technical Newsletter

SH19-8146 CMS REXX Compiler User's Guide and Reference - Supplement

GC24-5406 VM/SP: Program Update Info. -- REXX Language Enhancements

SC24-5465 VM/ESA: REXX/VM User's Guide

SC24-5466 VM/ESA: REXX/VM Reference

SC24-5598 VM/ESA R2: REXX/VM Primer

LYC0-9075 VM/ESA V1: REXX/370 LISTING

E.1.3 MVS

SC28-1882 TSO/E Version 2 REXX/MVS User's Guide

SC28-1883 TSO/E Version 2 REXX/MVS Reference

E.1.4 OS/2

S01F-0271 OS/2 Version 1.3 Procedures Language 2/REXX Reference

S01F-0272 OS/2 Version 1.3 Procedures Language 2/REXX User's Guide

S10G-6268 OS/2 (Version 2.0) Procedures Language 2/REXX Reference

S10G-6269 OS/2 (Version 2.0) Procedures Language 2/REXX User's Guide

SR28-5250 OS/2 (Version 2.1) REXX Handbook

E.1.5 AS/400

SC24-5512 AS/400 Procedures Language 400/REXX Reference

SC24-5513 AS/400 Procedures Language 400/REXX Programmer's Guide

SC24-5552 AS/400 Procedures Language 400/REXX Reference, Version 2

SC24-5553 AS/400 Procedures Language 400/REXX Programmer's Guide, Version 2

E.1.6 VSE

SC33-6528 VSE/ESA: REXX/VSE User's Guide

SC33-6529 VSE/ESA: REXX/VSE Reference

LY33-9144 VSE/ESA: REXX/VSE Diagnosis Reference

GC33-6533 VSE/ESA: REXX/VSE Licensed Program Specifications

E.1.7 Applications and Other REXX-related Books

GG24-1615 Using REXX in Practice: EXEC2 to REXX Conversion Experiences

GG24-3401 REXX/EXEC Migration To VM/XA SP

SC33-0478 GDDM REXX Guide

SR21-0864 SRA VM Using the CMS System Product Interpreter

SH20-7051 VM/SP System Product Interpreter:SQL/Data System Interface: Program Description/Operations Manual

GG66-3144 NetView Release 3: REXX Presentation Guide - Gibbons & Quigley

SC31-6135 NetView Customization - Writing Command Lists

GG66-3158 CMS Pipelines Tutorial - Hartmann, Kraines, and Lynn

GR28-2920 CUA 2001 VM Applications Core Functions Programmer's Reference Guide (CUA support for VM REXX applications)

E.2 Non-IBM Books and Manuals

Modern Programming Using REXX - Robert P. O'Hara and David R. Gomberg
In English: ISBN 0-13-597311-2 Prentice-Hall, 1985
ISBN 0-13-579329-5 (Second edition), 1988

The REXX Language - M. F. Cowlishaw
In English: ISBN 0-13-780735-X Prentice-Hall, 1985
ISBN 0-13-780651-5 (Second edition), 1990

    In German: ISBN 3-446-15195-8   Carl Hanser Verlag, 1988
                  ISBN 0-13-780784-8   P-H International, 1988
    In Japanese: ISBN 4-7649-0136-6   Kindai-kagaku-sha, 1988

Personal REXX User's Guide (PC-DOS and OS/2 REXX) version 2.0
    Mansfield Software Group, Inc., 1985-1990

ARexx User's Reference Manual (The REXX Language for the Amiga)
    William S. Hawes, 1987

uniREXX Reference Manual (REXX for a variety of Unix systems)
    The Workstation Group, 1990

Proceedings of the REXX Symposium for Developers and Users
    SLAC Report-368, 235pp, June 11, 1990

REXX In the TSO Environment - Gabriel F. Gargiulo
    ISBN 0-89435-354-3, QED Information Systems Inc.,
    Order #CC3543; 320pp, 1990
  Revised edition:
    ISBN 0-89435-418-3, QED Information Systems Inc.,
    471pp, 1993

Practical Usage of REXX - Anthony S. Rudd
    ISBN 0-13-682790-X, Ellis Horwood (Simon & Schuster), 1990

Personal REXX User's Guide (PC-DOS and OS/2 REXX) version 3.0
    Quercus Systems, 268pp, 1991

Portable/REXX for MS/DOS (Guide, Reference manual, Examples Reference,
    Reference Summary, and Learning to Program with Portable/REXX)
REXX/Windows (Product Guide and Reference)
    Keith Watts, Kilowatt Software, 1991

Proceedings of the REXX Symposium for Developers and Users
    SLAC Report-379, 244pp, May 8-9, 1991

Using ARexx on the Amiga - Chris Zamara and Nick Sullivan
    ISBN 1-55755-114-6, 424pp+diskette, Abacus, 1991

The REXX Handbook - Edited by Gabe Goldberg and Phil Smith III
    ISBN 0-07-023682-8, 672pp, McGraw Hill, 1991

Amiga Programmer's Guide to ARexx - Eric Giguere
    Commodore-Amiga, Inc., 1991

Programming in REXX - Charles Daney
    ISBN 0-07-015305-1, 300pp, McGraw Hill, 1992

Proceedings of the REXX Symposium for Developers and Users
    SLAC Report-401, 401pp, May 3-5, 1992

The ARexx Cookbook - Merrill Callaway
    ISBN 0-9632773-0-8, 221pp, Whitestone, 1992
    (Companion diskette: ISBN 0-9632773-1-6)

REXX--Advanced Techniques for Programmers - Peter C. Kiesel
    ISBN 0-07-034600-3, 239pp, McGraw Hill, 1993

REXX Tools and Techniques - Barry K. Nirmal
    ISBN 0-89435-417-5, 264pp, QED, 1993

REXX Reference Summary Handbook (OS/2) - C. F. S. Nevada, Inc
    C. F. S. Nevada, Inc, 20pp, 1993.

OS/2 2.1 REXX Handbook: Basics, Applications, and Tips - Hallett German
    ISBN 0442-01734-0, 459pp, Van Nostrand Reinhold, 1993

E.2.1 Applications and other REXX-related books

Command Language Cookbook - Hallett German
    ISBN 0-442-00801-5, 352pp, Van Nostrand Reinhold, 1992

Personal REXX User's Guide, Version 3.0 - OS/2 Supplement
    Quercus Systems, 94pp, 1992

VisPro/REXX (Visual programming with REXX)
    Hockware Inc, 196pp, 1993

REXX in der Praxis - Peter Kees
    ISBN 3-486-22666-5, 279pp, Oldenbourg, 1993

E.3 Papers

The Design of the REXX Language - M. F. Cowlishaw
    IBM Systems Journal, Vol 23, No. 4, 1984, pp326-335
    Offprints can be ordered from IBM, number: G321-5228

REXX on TSO/E - G. E. Hoernes
    IBM Systems Journal, Vol 28, No. 2, 1989, pp274-293
    Offprints can be ordered from IBM, number: G321-5359

Partial Compilation of REXX - R. Y. Pinter, P. Vortman, and Z. Weiss
IBM Systems Journal, Vol 30, No. 3, 1991, pp312-321
Offprints can be ordered from IBM, number: G321-5437

List of Abbreviations

AIX	Advanced Interactive Executive
APPC	advanced program to program communication
ANSI	American National Standards Institute
APA	all points addressable
API	application program interface
COBOL	Common Business Oriented Language
CPI-C	Common Programming Interface for Communications
CUA	Common User Access
CM/2	Communications Manager/2
CMS	Conversational Monitor System
DARI	Database Application Remote Interface
DB2/2	Database 2 for OS/2
DDCS/2	Distributed Database Connection Services/2
DLL	Dynamic Link Library
EHLLAPI	emulator high level language application programming interface
EPM	enhanced editor for PM
GUI	Graphical User Interface
IBM	International Business Machines Corporation
ISO	International Organizations for Standardization
ITSO	International Technical Support Organization
LAN	local area network
MMPM/2	Multimedia Presentation Manager
MVS	Multiple Virtual Storage
OS/2	Operating System/2
OS/400	Operating System/400
OIA	Operator Information Area
PAS/2	Personal Application System/2
PS/2	Personal System/2
PM	Presentation Manager
PROFS	Professional Office System
REXX	Restructured EXtended eXecutor language
SAA	Systems Application Architecture
VM/SP	Virtual Machine/System Product

Index

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Printed in U.S.A.

GG24-4199-00