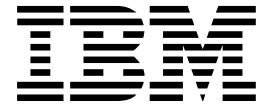AS/400 Advanced Series

REXX/400 Programmer's Guide

Version 4

AS/400 Advanced Series

REXX/400 Programmer's Guide

Version 4

```
┌─ Take Note! ──────────────────────────────────────────────────────────────────────┐
│                                                                                    │
│   Before using this information and the product it supports, be sure to read the general information under "Notices" on page vii. │
│                                                                                    │
└────────────────────────────────────────────────────────────────────────────────────┘
```

First Edition (August 1997)

This edition applies to the licensed program Operating System/400, (Program 5769-SS1), Version 4 Release 1 Modification 0, and to all subsequent releases and modifications until otherwise indicated in new editions.

Make sure that you are using the proper edition for the level of the product.

Order publications through your IBM representative or the IBM branch serving your locality. If you live in the United States, Puerto Rico, or Guam, you can order publications through the IBM Software Manufacturing Solutions at 800+879-2755. Publications are not stocked at the address given below.

IBM welcomes your comments. A form for readers' comments may be provided at the back of this publication. You can also mail your comments to the following address:

IBM Corporation
Attention Department 542
IDCLERK
3605 Highway 52 N
Rochester, MN 55901-7829 USA

or you can fax your comments to:

United States and Canada: 800+937-3430
Other countries: (+1)+507+253-5192

If you have access to Internet, you can send your comments electronically to IDCLERK@RCHVMW2.VNET.IBM.COM; IBMMAIL, to IBMMAIL(USIB56RZ).

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

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 that IBM product, program, or service may be used. Subject to IBM's valid intellectual property or other legally protectable rights, any functionally equivalent product, program, or service may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

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 Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Licensees of this program who wish to have information about it for the purpose of enabling: (i) the exchange of information between independently created programs and other programs (including this one) and (ii) the mutual use of the information which has been exchanged, should contact the software interoperability coordinator. Such information may be available, subject to appropriate terms and conditions, including in some cases, payment of a fee.

Address your questions to:

IBM Corporation
Software Interoperability Coordinator
3605 Highway 52 N
Rochester, MN 55901-7829 USA

This publication could contain technical inaccuracies or typographical errors.

This publication may refer to products that are announced but not currently available in your country. This publication may also refer to products that have not been announced in your country. IBM makes no commitment to make available any unannounced products referred to herein. The final decision to announce any product is based on IBM's business and technical judgment.

This publication contains examples of data and reports used in daily business operations. To illustrate them as completely as possible, the examples include the names of individuals, companies, brands, and products. All of these names are fictitious and any similarity to the names and addresses used by an actual business enterprise is entirely coincidental.

This publication contains small programs that are furnished by IBM as simple examples to provide an illustration. These examples have not been thoroughly tested under all conditions. IBM, therefore, cannot guarantee or imply reliability, serviceability, or function of these programs. All programs contained herein are provided to you "AS IS". THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE EXPRESSLY DISCLAIMED.

Programming Interface Information

This *REXX/400 Programmer's Guide* is intended to help you write programs using the AS/400 REXX interpreter. This *REXX/400 Programmer's Guide* documents General-Use Programming Interface and Associated Guidance Information provided by AS/400.

General-Use programming interfaces allow you to write programs that use the services of AS/400 REXX.

Trademarks

The following terms are trademarks of the IBM Corporation in the United States or other countries or both:

Application System/400	Operating System/400
AS/400	OS/2
BookManager	OS/400
DB2	Personal System/2
DB2/400	PS/2
IBMLink	SAA
Integrated Language Environment	SQL/400
Library Reader	Systems Application Architecture
Operating System/2	

Microsoft, Windows, and the Windows 95 logo are trademarks or registered trademarks of Microsoft Corporation.

PC Direct is a trademark of Ziff Communications Company and is used by IBM Corporation under license.

UNIX is a registered trademark in the United States and other countries licensed exclusively through X/Open Company Limited.

C-bus is a trademark of Corollary, Inc.

Java and HotJava are trademarks of Sun Microsystems, Inc.

Other company, product, and service names, which may be denoted by a double asterisk (**), may be trademarks or service marks of others.

About REXX/400 Programmer's Guide

This guide provides a wide-range discussion of programming with the IBM REXX for AS/400 system (also known as REXX/400). Its primary purpose is to provide useful programming information and examples to those who are new to REXX/400 and to provide those who have used REXX in other computing environments with information about the REXX/400 implementation.

This guide may refer to products that are announced, but are not yet available.

In the back of this book is a glossary and an index. Use the glossary to find the meaning of an unfamiliar term. Use the index to look up a topic and to see on which pages the topic is covered.

Who Should Read This Book

This guide is intended for the AS/400 system or application programmer, who wants to learn how to use REXX on the AS/400. While using the control language (CL) with REXX is discussed, much of the material in this guide applies to the system in general and may be used by programmers of all high-level languages supported by the AS/400 system.

What You Should Know before Reading This Book

Before using this guide, you should be familiar with general programming concepts and terminology, and have a general understanding of OS/400 and the AS/400 system. For more information about REXX, the *REXX/400 Reference* provides detail on all REXX instructions, functions, input and output, parsing, and application interfaces.

What This Book Contains

You will be introduced to the REstructured eXtended eXecutor (REXX) language. In addition, you will learn about the following:

- Contents of a REXX program, rules of syntax and substitution, and the use of variables

- How to write expressions, use conversations, enter AS/400 commands, control your program, and construct and design your REXX programs

- Examples of REXX programs.   .

Prerequisite and Related Information

For information about other AS/400 publications (except Advanced 36), see either of the following:

- The *Publications Reference* book, SC41-5003, in the AS/400 Softcopy Library.
- The *AS/400 Information Directory*, a unique, multimedia interface to a searchable database that contains descriptions of titles available from IBM or from selected other publishers. The *AS/400 Information Directory* is shipped with the OS/400 operating system at no charge.

Information Available on the World Wide Web

More AS/400 information is available on the World Wide Web. You can access this information from the AS/400 home page, which is at the following uniform resource locator (URL) address:

```
http://www.as400.ibm.com
```

Select the Information Desk, and you will be able to access a variety of AS/400 information topics from that page.

Chapter 1. Using REXX on the AS/400 System

Using REXX on the AS/400 System

As part of Operating System/400 (OS/400), REXX adds programming capabilities as a command processing language and an applications programming language. REXX, or the Restructured EXtended eXecutor language, is a procedural language for the Application System/400 (AS/400) system. REXX programs can reduce long, complex, or repetitious tasks to a single action.

REXX provides both an alternative to using Control Language (CL) programs and a way to expand CL.

- REXX provides a full set of structured programming instructions like DO...END and IF...THEN...ELSE. These instructions are discussed in Chapter 5, "Using REXX Instructions" on page 45.

- REXX can be used with other command environments it recognizes. These languages can be provided by the system, as CL is, or provided by the user within the rules which must be followed for REXX to find and recognize them. The interaction between REXX and CL is discussed in Chapter 7, "Understanding Commands and Command Environments" on page 79. User-defined interaction is discussed in the *REXX/400 Reference*.

Learning About ...

REXX is different from most of the programming languages currently available for the AS/400 system. Some of these differences, as well as some of the functional characteristics of REXX, are described here.

An Interpreted Language

The REXX language is an interpreted language. When a REXX program runs, the language processor directly interprets each language statement. Languages that are not interpreted must be compiled into a program object before they are run.

Free Format

REXX has only a few rules about programming format. This allows freedom and flexibility in program format. A single instruction can span many lines or multiple instructions can be entered on a single line. Instructions can begin in any column. Spaces or entire lines can be skipped. Instructions can be typed in uppercase, lowercase, or mixed case. REXX does not require line numbering.

Variables Without Type

REXX regards all data as character strings. REXX does not require that variables or arrays be declared as strings or numbers, where CL requires (*CHAR) or (*DEC). REXX will perform arithmetic on any string that represents a valid number, including those in exponential formats. REXX variables are discussed in Chapter 3, "Using Variables" on page 17.

Built-in Functions

REXX supplies built-in functions that perform various processing, searching, and comparison operations for both text and numbers. Other built-in functions provide formatting capabilities and arithmetic calculations. The REXX functions are discussed in "Using REXX Built-in Functions" on page 104.

Parsing

REXX includes extensive capabilities for working with character strings. Parsing input lets you easily assign variables from different input sources and manage the flow of information through your REXX program. REXX parsing is discussed in Chapter 6, "Using REXX Parsing Techniques" on page 61.

How to Determine and Correct Programming Errors

When a REXX program contains an error, messages with meaningful explanations are shown on the display. In addition, the TRACE instruction provides a powerful tool for determining problems in your REXX program. A complete description of these tools is provided in Chapter 10, "Determining Problems with REXX Programs" on page 123.

REXX and Systems Application Architecture

REXX/400 is one of the programming languages included in the IBM Systems Application Architecture (SAA). SAA is a framework of standards and definitions intended to promote consistency among different IBM products. Programs written in REXX according to SAA specifications are portable to all other SAA computing environments including the VM, TSO/E or IBM Operating System/2 (OS/2) computing environment, as long as system-specific instructions, functions or commands are not used.

To learn more about working with REXX in SAA supported computing environments, see the *SAA Common Programming Interface REXX Level 2 Reference*.

REXX/400, referred to in this book as REXX, is the SAA implementation on the AS/400 system. Occasionally, in this book, you will see the term REXX/400 used. If you are familiar with REXX in other computing environments, you should note that this term identifies an option available only to REXX on the AS/400 system.

Understanding the AS/400 System Security

Security of REXX programs is managed at a source file level. For more information on AS/400 system security, see the *Security – Reference* book.

Chapter 2. Writing and Running REXX Programs

Writing and running a REXX program occurs in two steps:

1. Enter the source for the REXX program,
2. Run the REXX interpreter against the source.

This chapter covers the following topics:

- Understanding the Parts of a REXX Program
- Understanding REXX Source Entry
- Running REXX Programs
- Using REXX Files
- Using the ILE Session Manager
- Using the SAY and PULL Keyword Instructions.

Understanding the Parts of a REXX Program

REXX programs are made up of clauses. There are many types of clauses, which are described here.

Using Clauses

The source statements which make up a REXX program, called *clauses*, can be:

- Null clauses
- Assignments
- Instructions
- Labels
- Commands.

Programs written in REXX are composed of clauses made up of tokens, which are character strings separated by blanks or by the nature of the tokens themselves. Conceptually, each clause is scanned from left to right before processing, and the tokens composing it are identified. Instruction keywords are recognized at this stage, comments are removed, and multiple blanks (except within literal strings) are converted to single blanks. Blanks adjacent to special characters are also removed.

REXX uses several types of delimiters:

- Tokens are delimited by blank spaces.

- Comments begin with /* and end with */.

- Clauses are delimited by a semicolon, which is implied by line ending, certain keywords or a colon if it follows a single symbol.

For more information on delimiters, see the *REXX/400 Reference*.

The following sections discuss using instructions, labels, and commands. The other two clause types, nulls and assignments are defined here.

Null Clauses   A clause that is empty (a blank line) or consists only of blanks or comments is called a *null clause.* REXX ignores all null clauses. Although not required by REXX/400, you should make the first line of your REXX program a comment. This identifies it as a REXX program in other SAA computing

5

environments. Comments can appear anywhere in a program, and are delimited by /* and */ as in the following:

```
/*    This is a comment                    */
```

Assignments   Assignments are clauses which assign values to variables. An assignment usually takes the form:

```
symbol = expression
```

Assignments are discussed in "Assigning Variables" on page 18.

Using Instructions

Clauses that begin with the keywords REXX recognizes are instructions. These keywords are recognized by REXX by their context. For example, when the word SAY is the first word in a clause, and is not followed by any equal sign or a colon, it is recognized as an instruction. The word SAY could appear anywhere in a literal string and not be interpreted as an instruction. This interpretation process is discussed in further detail in "Understanding REXX as an Interpreted Language" on page 10.

Certain keyword instructions, such as the IF instruction, may be made up of more than one clause. In this case, each clause begins with a REXX keyword, which becomes reserved. A complete list of the REXX reserved keyword instructions is contained in Appendix A, "REXX Keywords" on page 137. The keyword instructions are defined in the *REXX/400 Reference*.

Here is a simple REXX program which shows some of the types of source statements.

```
/* Name verification program */
SAY "Good morning.  Please enter your name."
PULL who
IF who = " " THEN SAY "Good morning stranger."
ELSE SAY "Good morning" who"."
```

This sample program consists of five source statements. It uses the SAY, PULL, and IF instructions which are detailed in "Using the SAY and PULL Keyword Instructions" on page 14 and "Using Structured Programming" on page 45. The types of clauses which make up this program are:

1. `/*...*/`

   The first clause of this program is a comment explaining the contents of the program. All REXX programs which will be used in SAA computing environments must start with a comment. First line comments are not required for REXX/400 programs, but are recommended as a REXX programming convention and a good programming standard. Comments are ignored as the program is interpreted, but are important in the documentation of the program.

2. `SAY`

   The second clause of this program is a *keyword instruction*, `SAY`, that shows text on the ILE Session Manager display.

3. `"Good morning.  Please enter your name."`

   Anything in quotation marks is shown just as it is. This is called a *literal string*.

4. PULL

   `PULL` is a keyword instruction which reads and holds the response entered by the program's user. This is the third clause.

5. who

   A *variable*, which gives a name to the place where the user's response is stored.

6. IF

   The fourth clause begins with the `IF` instruction to test a condition.

7. who = " "

   This is the condition to be tested. It asks the question "is the variable `who` empty?"

8. THEN

   This tells REXX to proceed with the instruction that follows, if the tested condition is true.

9. SAY "Good morning stranger."

   This shows `Good morning stranger.`, if the condition `who = " "` is true.

10. ELSE

    This final clause gives an alternative direction to run the instruction that follows, if the tested condition is not true.

11. SAY "Good morning" who"."

    This shows `Good morning`, followed by the value stored in `who`, if the tested condition is not true.

Using Functions and Subroutines

Internal functions, subroutines, and condition traps are indicated by clauses called *labels*. Labels are symbols that mark positions or portions of a program. They are distinguished by a trailing colon (for example, ERROR:). Other than their use with the CALL and SIGNAL instructions and for internal function calls, they are regarded as null clauses. Unlike null and instruction clauses, labels do not require trailing delimiters, such as semicolons or line endings, to separate them from other clauses.

The following program shows how internal functions and subroutines are used. The routine called SUM returns the sum of two numbers passed to it.

```
input1 = 2                      /* Set two numbers to use as input.*/
input2 = 3

CALL Sum input1, input2         /* Call SUM as a subroutine.      */
                                /* The parameters are not put     */
                                /* inside parentheses.            */
                                /* Subroutine calls assign the    */
                                /* return value to a variable     */
                                /* called result.                 */
                                /* Here the sum is written out the */
                                /* first time.                    */
```

```
SAY 'The sum of' input1 'and' input2 'is' result
                                /* Now call SUM as a function.    */
                                /* This time the parameters are   */
                                /* in parentheses, and the variable*/
                                /* result is not touched.         */
SAY 'The sum of' input1 'and' input2 'is' Sum(input1, input2)
EXIT

Sum:                            /* Here is the internal routine.  */
total = ARG(1) + ARG(2)         /* ARG is a built-in function     */
                                /* accesses the parameters passed */
                                /* to an internal routine, or to  */
                                /* the main REXX program.         */
RETURN total
```

Using Commands

Commands are clauses that are run by other programs. A command is simply an expression which is passed to the current command environment. By default, REXX/400 sends commands to the CL command environment for processing. Commands and command environments are described further in Chapter 7, "Understanding Commands and Command Environments" on page 79.

The following program shows how CL commands are issued from a REXX program. The program provides a value for the library parameter for the Display Library (DSPLIB) command, if one is not specified when the program is run.

```
ARG libname                          /* Get the parameter (if any). */
IF libname='' THEN libname="MYLIB"   /* If none provided, make the  */
                                     /* parameter "MYLIB".          */
"DSPLIB "libname                     /* Issue the DSPLIB command.    */
EXIT
```

Understanding REXX Source Entry

The source for a REXX program can be entered using the source entry tool you choose. The Source Entry Utility (SEU) supports the REXX source type. For more information about using SEU, see the *ADTS/400: Source Entry Utility*.

If you are familiar with SEU, you should be aware of three differences between entering REXX source and entering source for other languages. These are:

- Prompting assistance for REXX statements is not available.

- REXX syntax is not checked during source entry.

- Prompting for CL commands within members with the source type REXX is not available.

If you do not have a source file and library, use the Create Library (CRTLIB) command to create a library for your source files. Use the Create Source Physical File (CRTSRCPF) command to create a source file for REXX programs. QREXSRC is an IBM-supplied file in the QGPL library for REXX source.

Using REXX Source Type in Source Entry

The source type REXX can be used so source members that contain REXX programs are easily identified. The REXX source type is supported by SEU. Use of the REXX source type is not required. REXX will run with any source member regardless of source type.

Using the REXX source type, REXX provides efficiencies in running the REXX program. These efficiencies are not available with any other source type. When a REXX program is run, the source from the member is first converted into an internal form. The program is then run by using this internal form. When the source type is REXX, the internal form is saved in the associated space of the source member. Thus, it is available to use the next time the program is run. When the source type is not REXX, the internal form is also created but is not saved. In this case, each time the program is run, the internal form is created.

It should be noted that the internal form is not directly usable. REXX is responsible for maintaining the internal form and for ensuring that the internal form is kept up to date with the actual source. REXX will automatically rebuild the internal form after the source is updated. If the SEU is used, the internal form will be rebuilt as part of the SEU run, after the source member updates are saved.

When the source type is REXX, actions such as Delete File or Remove Member will not only effect the source member but also the internal form. When the source member is removed, the internal form associated with that member is also removed. When the source file is deleted, the internal form associated with any member of source type REXX will also be deleted. In addition, if the source type of an existing member is changed from REXX to some other type, the internal form is deleted.

When the source type is REXX and the REXX program is run, the internal form is used directly from the associated space by the REXX interpreter. As a result, the internal form does not increase the resources required to run the program. When the source type is not REXX, the internal form is created from the source dynamically when the REXX program is run. The internal form is saved for the duration of the run within the resources that are allocated to the REXX interpreter. As a result, in this case the internal form increases the amount of resources that are required to run the REXX program. Because there is a maximum amount of resources that can be allocated to the REXX interpreter, the space requirements to hold the internal form will reduce the space that will be available for other purposes, such as space to hold variables. If the source type is changed from REXX to something else, it is possible that a very large REXX program could now require resources beyond the maximum that is permitted. That is, changing the source type from REXX can cause a REXX program, that ran successfully before the source type change, to no longer run.

Using REXX Programs as Source File Members

REXX programs are not program objects. REXX programs are run from the source file. No compiling is necessary.

You should group REXX members by their security requirements, since security is established on the source file, not individual source members. A user with authority to the source file will have access to all of the members within it. Note that where line numbers are referred to in REXX programs, the line number used is the

relative line number. Sequence numbers are ignored by REXX. To make it easier to find lines, use the resequence option when you exit from SEU.

Understanding REXX as an Interpreted Language

The REXX interpreter works on your REXX program, clause by clause and token by token, doing what you have written.

How the tokens and clauses are interpreted depends on how they are used. This is how REXX interprets source statements.

- Tokens placed between /* and */ are comments. They are ignored by the interpreter.

- Tokens placed between quotation marks are literal strings. They are used exactly as written, without any further interpretation.

- A single token followed by an equal sign is interpreted to be an assignment. The token is assigned the value following the equal sign.

- A token immediately followed by a colon is interpreted to be a label, indicating a portion of the program which will include an internal function, subroutine, or condition trap.

- A clause which begins with a keyword, as listed in Appendix A, "REXX Keywords" on page 137, is interpreted as an instruction. REXX expects to perform the task indicated by the instruction.

- A completely blank line or a line consisting only of a semicolon is a null clause.

- Anything that is not an instruction, assignment, label, or null clause is interpreted as a command. Commands are passed to the current command environment unless otherwise indicated. See Chapter 7, "Understanding Commands and Command Environments" on page 79 for more information.

Running REXX Programs

REXX programs can be run, and the interpreter called, by:

- Using the Start REXX Procedure (STRREXPRC) command

- Using a REXX program as the command processing program (CPP) for a command

- Using the Work with Members option of the Program Development Manager

- Calling the QREXX application program interface (API).

Using the Start REXX Procedure Command

The Start REXX Procedure (STRREXPRC) command starts the REXX interpreter for a specific REXX program. The STRREXPRC command can be issued from anywhere a CL command can be issued. This is a way to run a REXX procedure from a CL program (the other way is CALL QREXX in the QSYS library, see page 12 for more information).

For information on the syntax for this command and where you can run it, see the *CL Reference*.

The *PARM* parameter of STRREXPRC passes an argument string to the REXX program. The REXX program can subdivide the string using the ARG or PARSE

ARG instructions within the program itself. You should make certain that the string is entered in a way that can be parsed by the receiving REXX program. You should also be aware that because the string is being entered by a CL command, CL rules for folding data will apply. To prevent folding, you should use quotation marks around the data. The quotation marks will not be passed to the REXX program. The REXX program will receive an exact image of the string entered for this parameter, unless the string is folded. For more information on using the ARG and PARSE ARG instructions, see "Using PARSE ARG" on page 62.

The *PARM* parameter lets you pass up to 3000 characters. Only the characters actually specified are passed.

Each time the STRREXPRC command is called, the REXX interpreter is restarted with the parameters specified.

Running REXX Programs by Using User-Defined Commands With REXX

REXX programs can be used as command processing programs (CPP) for CL commands. In this case you will see the following differences from the STRREXPRC command:

- The initial command environment and system exit programs can only be specified when the command is created using the Create Command (CRTCMD) command or changed using the Change Command (CHGCMD) command.

- The command may define separate parameters as needed. Each input is passed as one argument string.

- Even though each parameter is defined individually by the command, when the command is run all parameter values are concatenated together to form one argument to the REXX program. The order in which the values are concatenated is determined by the order of the parameters as defined by the command.

Most command definition functions are available. For more information on command definition and command definition objects (CDO), see the *CL Programming*.

The input argument string which is passed to the REXX program will be built from the user input and, potentially, the command itself. It is built in CL keyword format. The maximum length of the string is 5,989 characters including values, keywords, and punctuation. The ARG or PARSE ARG instructions may be used within the REXX program to parse the input string. For more information on keyword format and building the input string, see "Using Literal Patterns" on page 69.

Using the Program Development Manager (PDM) Work with Members Option

The Program Development Manager (PDM) Work with Members display includes an option to allow the REXX interpreter to be run against a member. Since REXX programs are not program objects, the REXX interpreter cannot be started from the PDM Work with Objects display.

Starting REXX from a Program

The REXX interpreter can be called by a program through the calling mechanism of the programming language that was used to write that program. The call is made to the QREXX program. The requirements for using this calling procedure are found in the *REXX/400 Reference*.

Using REXX Files

A REXX program has access to two files for input and output operations. One file is available for input, STDIN. One file is available for output, STDOUT. A second output file, STDERR, is used by the interpreter to write error messages and trace information. The following table shows the files, the instruction which uses each file, and the default settings for each of these files in interactive and batch mode.

File	Used by	Default in Interactive	Default in Batch
STDIN	PULL	Keyboard	QINLINE
STDOUT	SAY	Display	QPRINT
STDERR	TRACE	Display	QPRINT

The interpreter controls the opening and closing of STDIN, STDOUT, and STDERR.

You can redirect both STDIN and STDOUT, by using the CL override commands, in order to work with other than the defaults. If you use an override command for this purpose, the overridden file name must be specified as STDIN or STDOUT. Override commands must be issued before the REXX program uses these files. Overrides after a REXX program uses a file will not redirect STDIN or STDOUT. For more information, see "Overriding STDIN and STDOUT" on page 206.

Using the Integrated Language Environment (ILE) Session Manager

When STDIN is assigned to the keyboard and STDOUT is assigned to the display station, operations are controlled by the Integrated Language Environment (ILE) Session Manager and not through usual display file data management. This is the default for interactive jobs.

The ILE Session Manager supports:

- Paging backward and forward
- Input retrieval with F9
- Exit with F3
- End-of-file signalling with F4
- Print the scroller with F6
- Scroll to the top with F17
- Scroll to the bottom with F18
- Scroll to the left with F19
- Scroll to the right with F20
- Issue commands from the user window with F21
- Output of characters below '40'X.

Here is a simple program which routes output to the display station:

```
/* REXX example */
SAY 'This will be displayed on the Terminal Session'
```

This is what will appear on your terminal session:

```
    This will be displayed on the Terminal Session
    Press ENTER to end terminal session.




















    ==> _____
    _____
    3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
    18=Bottom  F19=Left   F20=Right F21=User Window
```

When the program finishes the session manager sends an implicit read to the display station. You will see the message, `Press ENTER to end terminal session.`

From this display, you can page through the session screens. When you press the Enter key, the session manager closes the session and sends the message, `End of terminal session.`

If you run multiple programs which all print to the display station interactively, all the session information is kept. If the program EXAMPLE is run again, the session is as follows:

```
    This will be displayed on the Terminal Session
    Press ENTER to end terminal session.
    This will be displayed on the Terminal Session
    Press ENTER to end terminal session.












    ==>  _____
    _____
    3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
    18=Bottom  F19=Left    F20=Right F21=User Window
```

As seen in the displays, input and output from all programs that use the ILE Session Manager is kept.

Using the SAY and PULL Keyword Instructions

The SAY and PULL instructions are used by REXX to allow for input from STDIN and output to STDOUT. These instructions allow you to create a dialogue between user input and REXX processing. In interactive mode, REXX uses line mode processing.

SAY   The SAY instruction writes data. Output from the SAY instruction is directed to STDOUT.

PULL   The PULL instruction reads from the REXX external data queue. If the queue is empty, PULL reads from STDIN. The maximum length of a line placed on the REXX external data queue is 32,767 bytes.

Note: Even if there are items on the REXX external data queue, PARSE LINEIN will read from STDIN without changing the external data queue.

Using Interactive Mode

A very simple example of using SAY and PULL for interactive user input and output was given earlier:

```
/* Name verification program */
SAY "Good morning.  Please enter your name."
PULL who
IF who = " " THEN SAY "Good morning stranger."
ELSE SAY "Good morning" who"."
```

When run with the defaults for STDIN and STDOUT, the following will be displayed:

```
 Good morning.  Please enter your name.




















 ==>  _____
      _____
 3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
 18=Bottom  F19=Left   F20=Right F21=User Window
```

The PULL instruction will cause the program to pause and wait for user input.

Display with response: When a response is given, the program will continue to run, and the terminal session will show the following:

```
 Good morning.  Please enter your name.
 Jesse
 Good morning JESSE.
 Press ENTER to end terminal session.

















 ==>  _____
      _____
 3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
 18=Bottom  F19=Left   F20=Right F21=User Window
```

Display without response: If no entry is made, and only the Enter key is pressed, the following will be shown:

```
    Good morning.  Please enter your name.

    Good morning stranger.
    Press ENTER to end terminal session.












    ==>  _____
         _____
    3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
    18=Bottom  F19=Left    F20=Right F21=User Window
```

Using Batch Mode

When running in batch mode, with the defaults for STDIN and STDOUT, REXX programs which use SAY instructions will produce a file using QPRINT. PULL instructions will read a record from the file QINLINE, which defaults to the inline data for the job.

Chapter 3. Using Variables

This chapter covers the information you need to work with variables, including:

- Understanding Variables and Constants
- Using Compound Symbols
- Using Variables in Programs, Functions, and Subroutines.

Understanding Variables and Constants

Variables and constants are types of symbols. A *symbol* is a group of up to 250 characters. These characters can be A-Z, a-z, 0-9, a period, an exclamation point, a question mark, an underscore, or any REXX/400 extension character identified as a NAME character for the coded character set identifier (CCSID) in which the REXX source file is written (see the *National Language Support* for more information).

Using Constants

A symbol that begins with a digit (0-9) or a period is a *constant*. You cannot change the value of a constant. Therefore, constants cannot be used as variables. One special form of constant symbol is a number in exponential notation. In REXX, numbers in exponential notation include the mantissa, the letter e or E, an optional + or - sign, and the whole number exponent. The following are a few examples of constants:

77   is a valid number

.0004   begins with a period (decimal point)

1.2e6   Exponential notation for 1,200,000

42nd   is not a valid number. Its value is always 42ND.

Using Variables

A *variable* is a symbol that represents a value. This value can be different each time the program is run or can change while the program is running. Although the value can change, the variable name stays the same.

A variable which has not been assigned a value is not initialized. It will contain the default value of the symbol, which is the symbol's own name in uppercase letters, as shown in the following examples:

```
/*  This displays unassigned variables.              */
SAY amount              /* This displays "AMOUNT".   */
SAY first               /* This displays "FIRST".    */
SAY price               /* This displays "PRICE".    */
SAY who                 /* This displays "WHO".      */
```

Naming Variables

Variable names may be up to 250 characters long. The other rules for naming variables are:

1. The first character must be either A-Z, a-z, 0-9, a period, an exclamation point, a question mark, an underscore, or any REXX/400 extension character identified as a NAME character for the coded character set identifier (CCSID) in which the REXX source file is written.

   REXX treats all letters in variable names as if they were in uppercase, so whether you write fred, Fred, or FRED as the name of a variable, REXX uses FRED as the variable name.

2. The remaining characters can be either A-Z, a-z, 0-9, a period, an exclamation point, a question mark, an underscore, or any REXX/400 extension character identified as a NAME character for the coded character set identifier (CCSID) in which the REXX source file is written.

   The period has a special meaning for REXX variables. It forms compound symbols. You should avoid using the period in variable names until you understand compound symbols, which are discussed in "Using Compound Symbols" on page 20.

Choosing variable names that are descriptive helps make your program more understandable. REXX helps you do that by allowing long names and allowing some punctuation and numbers in the names.

Assigning Variables

The process of giving a variable an initial value or changing its value is called an *assignment*.

The following is the syntax of an assignment:

```
symbol = expression
```

where:

symbol   is a valid REXX variable name

expression is the information to be stored. This can be a number, string, or a calculation performed by REXX.

REXX evaluates the *expression* and puts the result into the variable called *symbol*.

The following are examples of assigning values to variables:

- To give the variable *total* the value 0, use:

  ```
  total = 0
  ```

- To give another variable, called *price*, the same value as *total*, use:

  ```
  price = total
  ```

- To give the variable *total* a new value (the old value of *total* plus the value of *something*), use:

  ```
  total = total + something
  ```

The following are more examples of assigning variables:

```
data = 1                          /* This is an integer number.        */
data = 1+1                        /* This is an integer expression.    */
data = 3.14159                    /* This is a decimal number.         */
data = data * 2                   /* This is a decimal expression.     */
data = 2.03E+12                   /* This is an exponential notation.  */
data = data / 2                   /* This is an exponential expression.*/
data = "Hello, world"             /* This is a character string.       */
data = Substr("Hello, world",1,5) /* This is a character expression.   */
```

Assigning Variables from User Inputs: Variables can be assigned from input supplied by the user while the program is running. The PULL and ARG keyword instructions are commonly used for this purpose.

The PULL Instruction: The PULL instruction pauses the program to allow the user to type one or more items of data which are assigned to variables. The items are separated by spaces, as shown in the following example:

```
SAY "Type two numbers (leave a space between) and press Enter"
PULL first second
```

PULL reads the two numbers entered and assigns them, in order, to the list of variables that follow it. PULL is a shortened version of the PARSE PULL instruction which is discussed in "Using PARSE PULL" on page 62. The process of reading and breaking up information is called *parsing*, which is discussed in greater detail in Chapter 6, "Using REXX Parsing Techniques" on page 61.

The ARG Instruction: The ARG instruction performs the same operation as PULL, except that items are entered as arguments when the program is called.

```
/* Displays the sum of two numbers                */
/* entered this time at the command line.         */
ARG first second              /*collects entries */
SAY "The sum is" first + second
```

ARG is another form of the PARSE ARG instruction which is discussed in "Using PARSE ARG" on page 62.

If this program were put into the file QREXSRC in library QGPL in a member named SUM, the command to have it add one plus two would be:

```
STRREXPRC SRCFILE(QGPL/QREXSRC) SRCMBR(SUM) PARM('1 2')
```

The program would display The sum is 3.

Assigning an Expression Result: Variables can be assigned data that is the result of a calculation or other manipulation. This data is represented as an expression. The following are examples of assigning the result of a calculation to a variable:

```
area = 3 * 5              /* The area of a 3 by 5 in. rectangle */
SAY area "sq. in."       /* displays "15 sq. in.".             */

diameter = 5             /* The area of a 5 in. circle,        */
radius = diameter/2
area = 3.14 * radius **2  /* 3.14 times the radius squared      */
SAY area "sq. in."       /* displays "19.6250 sq. in.".        */
```

The previous examples are simple demonstrations of assigning expression results to a variable. But REXX expressions can have very complex forms, and they can work with all kinds of information. For further information on expressions, see Chapter 4, "Using REXX Expressions" on page 31.

Displaying a Variable's Value: To write the value of a variable to STDOUT, at any given point in a program, use the SAY instruction:

```
amount = 100               /* This assigns 100 to AMOUNT.     */
money = "dollars"          /* This assigns "dollars" to MONEY. */
SAY amount money           /* This displays "100 dollars".    */
amount = amount + 25       /* This adds 25 to AMOUNT.         */
SAY amount money           /* This displays "125 dollars".    */

SAY price                  /* This displays "PRICE".          */
```

Notice that when a reference is made to an unassigned variable, the default value is the variable's name in uppercase letters.

Another instruction which you can use to check the value of a variable while the program is running is the TRACE instruction. For further information, see "Using the TRACE Instruction and the TRACE Function" on page 123.

Using Compound Symbols

A variable name containing at least one period and at least one other character following the period is called a *compound symbol*. It cannot begin with a digit or a period, and if there is only one period, the period cannot be the last character. The following are a few examples of compound symbols:

Note: In these examples, the < and > are DBCS symbols representing '0E'X and '0F'X, respectively. For more information on DBCS character strings and symbols, see the *REXX/400 Reference*.

```
fred. = ''
fred.3
array.G.B.
PENS..six.6
<.F.R.E.D>.<.C.D>
```

Stems and Tails

The compound symbol consists of a *stem* and a *tail*. The stem contains the beginning of the name up to and including the first period. The following are the stems from the previous examples:

```
fred.
array.
PENS.
<.F.R.E.D>.
```

The number of elements in the stem are generally stored in STEM.0. If you use a DO loop, you can easily process all the elements.

The stem is followed by the tail, which consists of one or more valid symbols (constants or variables) that are separated by periods.

The following are tails from the compound symbol examples:

```
3
G.B.
.six.6
<.C.D>
```

The following is an example:

```
/* This program records account balances for customers of a pet store. */

last_name = 'Jones'
customer.last_name.first_name = 'Suzanne'
customer.last_name.address = '123 Main Street'
customer.last_name.balance = 115.23
customer.last_name.purchase = 'yellow parrot'

last_name = 'Smith'
customer.last_name.first_name = 'Adrian'
customer.last_name.address = '17 Cherry Lane'
customer.last_name.balance = 79.98
customer.last_name.purchase = 'brown hamster'

last_name = 'Jones'
Say 'Customer' customer.last_name.first_name last_name 'has a balance of',
customer.last_name.balance 'dollars.'
Say 'This customer''s last purchase was a' customer.last_name.purchase'.'

/* The output from this program is:

Customer Suzanne Jones has a balance of 115.23 dollars.
This customer's last purchase was a yellow parrot.
*/
```

Derived Names

You can use compound symbols to create an array of variables that can be processed by their derived names. For example, take the following collection:

```
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"
```

If you know which day of the week it is, then you know the name of that day. If j=6, then the instruction SAY day.j displays Friday.

This is the path that REXX takes to determine what is displayed:

1. REXX recognizes the symbol day.j as compound because it contains a period.

2. The characters following the period may be the name of a variable. In this case, the variable j.

3. The value of j is substituted with 6, producing a derived name of day.6.

4. The value of the variable day.6 is the literal string "Friday".

So, for example, if you want to display the days of the week forever:

```
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"

DO j = 1 by 1  /*This is a DO forever.  J gets incremented. */
   SAY day.j
   IF j = 7 THEN j = 0
END
```

This idea can be extended. By using the SELECT instruction, you can SAY the days for the month of January. The SELECT instruction performs the same operation as nested IF...THEN...ELSE statements, but is clearer than a large nested group.

Note: The following example will run correctly if Sunday is the first day of the month.

```
day.1 = "Sunday"
day.2 = "Monday"
day.3 = "Tuesday"
day.4 = "Wednesday"
day.5 = "Thursday"
day.6 = "Friday"
day.7 = "Saturday"

DO dayofmonth = 1 to 31 /* operation // divides and returns the remainder */
   dayofweek = (dayofmonth+6)//7 + 1

   SELECT
      WHEN dayofmonth = 1 THEN th = "st"
      WHEN dayofmonth = 2 THEN th = "nd"
      WHEN dayofmonth = 3 THEN th = "rd"
      WHEN dayofmonth = 21 THEN th = "st"
      WHEN dayofmonth = 22 THEN th = "nd"
      WHEN dayofmonth = 23 THEN th = "rd"
      WHEN dayofmonth = 31 THEN th = "st"
      OTHERWISE th = "th"
   END

   SAY day.dayofweek dayofmonth||th "January"
END
```

In this case, the SELECT instruction lets you choose a particular ending on the number of the day, depending on the day of the month. For example, the first line the program will display is:

```
Sunday 1st January
```

For more information on the SELECT instruction, see the *REXX/400 Reference*. Further information on the program controls of DO...END and IF...THEN...ELSE can be found in "Using Structured Programming" on page 45.

Arrays

Many programming languages provide a data structure called *arrays*, which allow you to access a sequentially numbered set of variables. In REXX, stemmed variables can be used in a similar way.

Example 1: If you wanted to read in ten values entered by the user, you could write:

```
DO i = 1 to 10
  SAY 'Enter value number' i
  PARSE PULL value.i
END
```

The subscripts on stemmed variables are constant or variable symbols, not expressions.

Example 2: Here is how you could insert a new value in the first entry in a stemmed variable, and shift all the old values down by one:

```
/* Assume count is the number of entries in stemmed variable array. */
DO i = count + 1 to 2 by -1
   previous = i - 1
   array.i = array.previous
END
count = count + 1                          /* Update the new count. */
array.1 = newvalue                         /* Store the new value.  */
```

Stemmed variables are more flexible than traditional arrays because the tails may have character values in addition to numeric values. You can store data of different types in the same stemmed variable. And when you have a *sparse array*, an array where the elements are not numbered consecutively, no storage is wasted for the unused entries.

Example 3: This is an example of using compound symbols to collect and process data. In the first part of the program, the first player's score is entered into SCORE.1, the second player's into SCORE.2, and so on. By using compound symbols, the array of SCOREs is processed to give the result in the required form.

```
/* This is a scoreboard for a game.  Any number of      */
/* players can play.  The rules for scoring are these:  */
/*                                                      */
/* Each player has one turn and can score any number of */
/* points;  fractions of a point are not allowed.  The  */
/* scores are entered into the computer and the program */
/* replies with                                         */
/*                                                      */
/*      the average score (to the nearest hundredth of  */
/*                          a point)                    */
/*      the highest score                               */
/*      the winner        (or, in the case of a tie,    */
/*                          the winners)                */

/*------------------------------------------------------*/
/* Obtain scores from players.                          */
/*------------------------------------------------------*/
SAY "Enter the score for each player in turn.  When all"
SAY "have been entered, enter a blank line."
SAY
```

```
n=1
DO forever
  SAY "Please enter the score for player "n
  PULL score.n
  SELECT
    WHEN DATATYPE(score.n,whole) THEN n=n+1
    WHEN score.n="" THEN leave
    OTHERWISE  SAY "The score must be a whole number."
  END
END

n = n - 1                  /* now n = number of players */
IF n = 0 THEN EXIT
/*-------------------------------------------------------*/
/* Calculate average score.                              */
/*-------------------------------------------------------*/
total = 0
DO player = 1 to n
   total = total + score.player
END

SAY "Average score is",
    FORMAT(total/n,,2,0) /* Format "total/n" with       */
                         /*   no leading blanks,        */
                         /*   round to 2 decimal places,*/
                         /*   and do not use            */
                         /*     exponential notation.   */

                                  /* continued ...   */

/*-------------------------------------------------------*/
/* Calculate highest score.                              */
/*-------------------------------------------------------*/
highest = 0
DO player = 1 to n
   highest = MAX(highest,score.player)
END
SAY "Highest score is" highest

/*-------------------------------------------------------*/
/* Now calculate:                                        */
/* * W, the total number of players that have a score    */
/*   equal to HIGHEST                                    */
/* * WINNER.1, WINNER.2 ... WINNER.W, the id-numbers     */
/*   of these players                                    */
/*-------------------------------------------------------*/
w = 0                    /* number of winners           */
DO player = 1 to n
   IF score.player = highest THEN DO
     w = w + 1
     winner.w = player
   END
END

/*-------------------------------------------------------*/
/* Announce winners.                                     */
/*-------------------------------------------------------*/
IF w = 1
```

```
        THEN SAY "The winner is Player #"winner.1
ELSE DO
    SAY "There is a draw for top place.  The winners are"
    DO p = 1 to w
        SAY "     Player #"winner.p
    END
END
SAY
```

Two-dimensional Arrays

There can be more than one period in a compound symbol. For example, the
following program simulates a board on which checkers can be played. The `BOARD`
is a two-dimensional array (8 squares by 8 squares) called `BOARD.ROW.COL`. There
is a total of 64 squares.

The picture shows how the playing pieces are set at the start of the game.
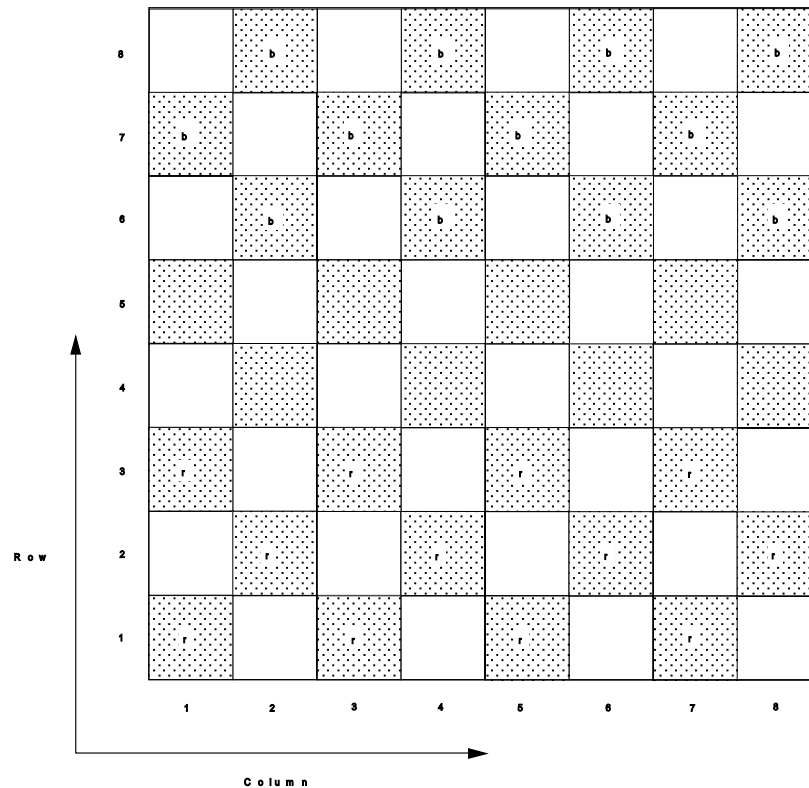
```
          /* In the internal representation, Red's "men" are      */
          /* represented by the character "r" and Red's "kings"    */
          /* by the character "R".  Similarly, Black's "men" and   */
          /* "kings" are represented by "b" and "B".               */
          /*-----------------------------------------------------*/
          /* Clear the board.                                      */
          /*-----------------------------------------------------*/
          board. = " "

          /*-----------------------------------------------------*/
          /* Set out the men.                                      */
          /*-----------------------------------------------------*/
          DO col = 1 by 2 to 7
             board.1.col = "r"
             board.3.col = "r"
          END
          DO col = 2 by 2 to 8
             board.2.col = "r"
          END
          DO col = 2 by 2 to 8
             board.6.col = "b"
             board.8.col = "b"
          END
          DO col = 1 by 2 to 7
             board.7.col = "b"
          END
```

Using Variables in Programs, Functions, and Subroutines

The following section discusses several considerations when using variables in REXX programs.

Using Special Variables

REXX has three special variables that are assigned values automatically as needed:

RC   The RC variable holds the return code for the last command run by the REXX program. Following SIGNAL events (SYNTAX, ERROR, and FAILURE), RC is set to either the syntax error number or the command return code. For more information on the RC special variable, see "Understanding Return Codes" on page 82.

RESULT   The RESULT variable holds the value set by a RETURN instruction from any subroutine or from an EXIT instruction in an external subroutine. When a subroutine is called by the CALL instruction, and the RETURN instruction which ends the subroutine specifies an expression, the value of that expression is put in the RESULT special variable. If the RETURN instruction does not specify an expression, RESULT is set to its default value, RESULT. For more information on the RESULT special variable, see "Understanding External Routines Written in REXX" on page 101.

SIGL   The SIGL variable holds the line number of the last call or branch to a label. For more information on the SIGL special variable and its use with conditions, see "Using Condition Trapping" on page 133.

Using the SYMBOL Function

Occasionally during the course of a program, it is useful to check if a symbol has already been used as a name of a variable. To do this, use the SYMBOL function:

►►—SYMBOL(*name*)—►◄

where *name* is the name of the symbol that you want to check.

This function returns the following codes:

- BAD, if *name* is not a valid symbol

- VAR, if *name* has already been used as a variable in the program

- LIT, if *name* is a valid variable that has not yet been assigned a value, or if it is a constant.

One use of SYMBOL is to ensure initialization of a variable. You may want to make certain that the variable is set to a proper starting value before it is used in an operation. For example:

```
IF SYMBOL("CASH") = "LIT" THEN cash = 0
cash = cash + payment
```

Notice what happens if the argument of SYMBOL is not in quotation marks:

```
cash = 100
SAY SYMBOL(CASH)      /* Says "LIT" because 100 is a literal. */
SAY SYMBOL("CASH")    /* Says "VAR" because CASH is the name  */
                      /* of a variable.                       */
```

Using the PROCEDURE Instruction

By default, variables created in a program are available to all internal routines and the main program. The PROCEDURE instruction limits the scope of variables.

To create a new generation of variables in an internal routine, use the PROCEDURE instruction as the first instruction of that routine. Then all the variables in the main program will be hidden from the routine. Local variables may be created in the routine (even using the same names as variables existing in the main program). When the routine ends, all of its local variables are deleted. The values of these variables cannot be accessed in the main program, and they cannot be accessed on future calls to the routine.

The following program shows the use of the PROCEDURE instruction. The program calculates the average of a list of values, assigned to the variable LIST. The value is calculated in a subroutine, which has the list passed to it as a parameter (the subroutine does not touch the variable LIST directly). The variable named COUNT is used in both the main program and the subroutine, but because of the PROCEDURE instruction, two separate variables are created.

```
/* Using the PROCEDURE Instruction                                    */

count = 999
list = 3 4 5 6 7

CALL average list

/* At this point:  COUNT = 999 (it is unchanged)                      */
/*                 list = 3 4 5 6 7 (it is unchanged)                 */
/*                 RESULT = 5 (it is set by the RETURN instruction.   */
Say 'The average is' result 'and COUNT is' count

EXIT

AVERAGE:
/* The argument must be a list of numbers, separated by blanks.    */
/* The average of the numbers is returned. */

PROCEDURE

/* At this point, all the variables in the main program are hidden */

ARG inputlist
sum = 0                                 /* SUM is now a local variable    */

DO count = 1 to words(inputlist)  /* COUNT is now a local variable */
    sum = sum + word(inputlist,count)
END

SAY 'In the subroutine (after the loop) COUNT is' count

RETURN sum/words(inputlist)

/* The output from the program is
in the subroutine (after the loop) COUNT is 6.
The average is 5 and COUNT is 999.
*/
```

The PROCEDURE instruction can only be used within an internal routine, and it must be the first instruction in the routine.

If an external routine is called, the PROCEDURE instruction is implied. The variables of the calling program are hidden from the external routine, and vice versa.

Using the PROCEDURE EXPOSE Instruction

To share a limited set of variables between a main routine and an internal subroutine, use:

►►──PROCEDURE──EXPOSE──▼─*name*─┘──►◄

where *name* is the name of a variable to be shared.

Any variables not named by the PROCEDURE EXPOSE instruction are protected just as with the PROCEDURE instruction. The ones listed are available to the subroutine. The list of variables to be exposed can also be assigned to a single

variable, and that single variable can be used to identify the whole list. You can also share the variables in an array by specifying the stem of that array.

Example 1: The following program calculates the total interest earned from the checking, savings, and certificate accounts. The program uses two internal subroutines to obtain the same value in two different ways.

These two subroutines are identical except for the manner in which they expose the same variables, defined in the beginning. The first subroutine, INTEREST1, exposes the three variables by listing them. The second subroutine, INTEREST2, exposes the variables by using one variable, enclosed in parentheses, which denotes the variable list.

```
/* Assign three variables.                                   */
check_rate = .055
save_rate = .065
cert_rate = .075

/* Remember the list of variables.                           */
/* The variables names are put in quotation marks            */
/* because the names are required,                           */
/* not the values.  The names may be uppercase or lowercase. */
rate_list = "check_rate SAVE_RATE CERT_RATE"

checking_balance = 1000
saving_balance = 5000
cert_balance = 10000

/* Call the first routine to calculate total interest        */
earning1 = interest1(checking_balance, saving_balance, cert_balance)


/* Call the second routine to calculate total interest       */
earning2 = interest2(checking_balance, saving_balance, cert_balance)

/* At this point, earning1 = earning2, because the two functions */
/* return the same result.                                   */

EXIT

INTEREST1:
PROCEDURE EXPOSE check_rate save_rate cert_rate
ARG balance1, balance2, balance3

total = balance1 * check_rate + balance2 * save_rate + balance3 * cert_rate

RETURN total

INTEREST2:
PROCEDURE EXPOSE (rate_list)
ARG balance1, balance2, balance3

total = balance1 * check_rate + balance2 * save_rate + balance3 * cert_rate

RETURN total
```

Example 2: The PROCEDURE EXPOSE instruction will expose all variables with a given stem when the stem is exposed. Here is the interest calculation example again, using stems.

```
/* Assign three variables                                          */
rate.check = .055
rate.save = .065
rate.cert = .075

checking_balance = 1000
saving_balance = 5000
cert_balance = 10000

/* Call the routine to calculate total interest                    */
earning = interest(checking_balance, saving_balance, cert_balance)



/* At this point, earning is the same as earning1 and earning2 were */
/* in the previous example.                                         */

EXIT

interest: Procedure expose rate.
ARG balance1, balance2, balance3

total = balance1 * rate.check + balance2 * rate.save + balance3 * rate.cert

RETURN total
```

Chapter 4. Using REXX Expressions

An *expression* is a collection of tokens that is evaluated. Expressions are used by variable assignments, instructions, and function calls. In any of the following instructions, you can put an expression where the word *expression* appears:

```
symbol = expression
SAY expression
IF expression THEN...
```

Here are some simple expressions, with the evaluated results shown in the comment:

```
2 + 2           /* Its value is '4'.                          */
"D" "E" "F"     /* Its value is 'D E F'.                      */
5 < 7           /* Its value is '1', because the comparison is true.*/
```

This chapter covers the information you need to know in order to write expressions that REXX can evaluate. This includes:

- Using Terms and Operators
- Using Function Calls
- Using Expressions in Instructions
- Using Expressions as Commands.

Using Terms and Operators

Expressions contain the data to be evaluated and the operations to be performed.

Terms   The data the expression evaluates is called a *term*. An expression can contain many different terms. The kinds of terms that can be included in expressions are:

Numbers   Numbers are strings that REXX can use in calculations. REXX recognizes them as constant values. The following are examples of valid numbers:

```
25          3.14159          1990
```

Literal strings   Literal strings are strings enclosed within matched quotation marks. The string is treated as a constant value. The following are examples:

```
"Wednesday"
"09 June 90"
'LIBNAME/FILENAME'
```

Variables   Variables are symbols that stand for changeable data. REXX places the value of the variable in the expression. The following are examples:

```
date=30      /* This places the number 30      */
             /* in the variable DATE.          */

month="March"  /* This places the literal string */
             /* March in the variable MONTH.   */

SAY month date /* This displays "March 30".     */
```

31

Function calls Function calls are any call to a function in an expression. REXX performs the function call, then uses the returned result as the expression term. The following are examples:

```
SAY TIME()      /* This displays the current time.*/
SAY SUBSTR("REXX",2,1) /* This displays "E",        */
                       /* because E is 1 character  */
                       /* from the string "REXX",   */
                       /* starting with the         */
                       /* 2nd character.            */
```

Operators The computation to be performed on the terms is indicated by an operator. An expression can contain many operators, depending on its complexity. The operators can perform arithmetic, string, comparison, and logical operations on the terms included in the expression.

You can control how the expression works by understanding the order in which expressions are evaluated and how to control that evaluation process using parentheses.

Order of Operation Expressions are evaluated from left to right, with the operations performed on adjacent terms. Some operators have higher priority than others, which changes the left to right evaluation. The complete order of precedence of operators is shown in Table 5 on page 145.

Parentheses You can also control how expressions are evaluated by using parentheses. Expressions within parentheses are evaluated first. The following are examples:

```
SAY 6-4+1      /* This displays 3.       */
SAY 6-(4+1)    /* This displays 1.       */
SAY 3+2||2+3   /* This displays 55.      */
SAY 3+(2||2)+3 /* This displays 28.      */
```

Appendix D, "Operators and Order of Operations" on page 143 provides a complete list of all of the operators recognized by REXX.

Using Arithmetic Operators

Arithmetic operations can be performed on valid REXX numbers. The following are examples of numbers:

12   This is an *integer.*

0.5   This is a *decimal fraction*.

3.5E6   This is a *floating point* number using *exponential notation*.

-5   This is a *signed* number.

REXX provides extensive arithmetic capabilities. The following table shows some of the arithmetic operators available, and an example of each.

Operator	Operation	Example
+ (plus sign)	Addition	SAY 7+2 /* This displays '9'. */
- (minus sign)	Subtraction	SAY 7-2 /* This displays '5'. */
* (asterisk)	Multiplication	SAY 7*2 /* This displays '14'. */
/ (one slash)	Division	SAY 7/2 /* This displays '3.5'.*/
% (percent sign)	Integer division. The result is a whole number with the remainder ignored.	SAY 7%2 /* This displays '3'. */
// (two slashes)	Remainder after integer division	SAY 7//2 /* This displays '1'. */
** (two asterisks)	Exponentiation	SAY 7**2 /* This displays '49'.*/

Note: A valid REXX number can be either a constant or a variable which contains a number.

By default, REXX calculates to nine significant digits, not counting the zeros that come just after the decimal point in very small decimal fractions.

```
SAY 7/30000000000                /* This displays '0.000000000233333333'.*/

SAY 1*2*3*4*5*6*7*8*9*10*11*12 /* This displays '479001600'.            */
```

You can change the number of significant digits by using the NUMERIC DIGITS instruction. See "Using the NUMERIC DIGITS Instruction" on page 35.

Using the DATATYPE Function

Arithmetic operations can only be performed on valid numbers. Before performing arithmetic operations on data, you may want to check that the data is valid numeric data. To do this, use the DATATYPE function.

In its simplest form, this function returns the word NUM, if the argument is a valid number that could be used in arithmetical operations. Otherwise, it returns the word CHAR.

The following are examples of using the DATATYPE function:

```
DATATYPE(99)           /* This returns NUM.     */
DATATYPE(6.6)          /* This returns NUM.     */
DATATYPE(5.5.5)        /* This returns CHAR.    */
DATATYPE('5,000')      /* This returns CHAR.    */
DATATYPE(5 4 3 2)      /* This returns CHAR.    */
```

If you wanted a user to keep entering values until a valid number is entered, you could write:

```
DO UNTIL DATATYPE(howmuch) = "NUM"
  SAY "Enter a number"
  PULL howmuch
  IF DATATYPE(howmuch) = "CHAR"
  THEN SAY "That was not a number.  Try again."
END

SAY "The number you entered was" howmuch
```

There is an additional form of the DATATYPE function which provides more information. This form requires two arguments. For example, if you were interested only in whole numbers, you would use the following format:

►►──DATATYPE(number,whole)──►◄

where:

number
   Is the data to be tested

whole
   Is the type of data to be tested for (in this case, a whole number). Only the first character is inspected. Thus, to test for whole numbers, it would be sufficient to write W or w.

This form of the function returns 1 (true) if *number* is a whole number or 0 (false) if not, as shown in the following example:

```
DO UNTIL DATATYPE(howmany,whole)
```

⋮

```
  PULL howmany
```

⋮

```
END
```

If you also wanted to restrict the input to numbers greater than zero, you would use:

```
DO UNTIL DATATYPE(howmany,whole) & howmany > 0
```

⋮

```
  PULL howmany
```

⋮

```
END
```

The DATATYPE function can test for other types of data, as well. See the DATATYPE function in the *REXX/400 Reference* for more information.

Using Exponential Notation

Since it is easy to make a mistake counting the zeros in numbers, it is useful to use exponential notation.

Numbers written in exponential notation, like 1.5E9, are sometimes called *floating point numbers*. Conversely, ordinary numbers, like 3.14, are sometimes called *fixed point numbers*.

Exponential notation is indicated by a fixed point number followed by the letter E followed by a whole number. The whole number is the exponent. When the exponent is positive, it indicates how many places to the right the decimal point of the fixed point number moves to obtain the same value as an ordinary number. When it is negative, it indicates the number of places the decimal point moves to the left. The following are examples:

   4.5E6 is the same as 4500000

   23E6 is the same as 23000000

1E12 is the same as 1000000000000

4.5E-3 is the same as 0.0045

1E-6 is the same as 0.000001

Exponential notation can be used in expressions and when entering numeric data. REXX will use this notation when displaying results that are too big or too small to be expressed conveniently as ordinary numbers or decimals. When REXX uses this notation, the mantissa will usually be a number between 1 and 9.999999999, as shown in the following example:

```
j = 1
DO UNTIL j > 1E12
   SAY j
   j = j * 11
END
                    /* This displays '1'.                       */
                    /* This displays '11'.                      */
                    /* This displays '121'.                     */
                    /* This displays '1331'.                    */
                    /* This displays '14641'.                   */
                    /* This displays '161051'.                  */
                    /* This displays '1771561'.                 */
                    /* This displays '19487171'.                */
                    /* This displays '214358881'.               */
                    /* This displays '2.35794768E+9'.           */
                    /* This displays '2.59374246E+10'.          */
                    /* This displays '2.85311671E+11'.          */
```

Using the NUMERIC DIGITS Instruction: If you do not want to use exponential notation, or simply want to increase the accuracy of your calculations, you can use the NUMERIC DIGITS instruction to change the number of significant digits from its default of 9.

The following are examples:

```
/*   Examples of numbers with unusually high precision   */

NUMERIC DIGITS 10
SAY "A large signed number is" 2**31-1
                    /* Displays 'A large signed number is 2147483647'.*/
NUMERIC DIGITS 48
SAY "1/7=" 1/7
    /* Displays 1/7=0.142857142857142857142857142857142857142857.*/
```

You can check the NUMERIC DIGITS setting at any time by using the DIGITS function, as shown in the following example:

```
SAY 'The current precision is' DIGITS() 'digits.'
```

Controlling Rounding and Truncation: Because of the way arithmetic operations are carried out, the accuracy of the final results may be influenced by the rounding operations.

The following is an example:

```
NUMERIC DIGITS 3
SAY 100.3 + 100.3            /* Displays '201',           */
                             /* which is 200.6 rounded.   */
```

REXX uses conventional rounding, and rounds as each operation within the expression is completed. In the following example, the expression is evaluated from left to right and rounding occurs after each addition:

```
SAY 100.2 + 100.2 + 100.2    /* Displays '300', which is  */
                             /* 200.4, rounded to 200,    */
                             /* plus 100.2, which is 300.2 */
                             /* rounded to 300.           */
```

When your program performs a series of arithmetic operations, rounding can affect the accuracy of the final result.

You can use the FORMAT function to control rounding of numbers at the point in your calculations where you want rounding to occur. The TRUNCATE function is used to round numbers down and override conventional rounding. The following is an example:

```
/*  An example of rounding  */
qty.1 = 500
qty.2 = 500

unitprice.1 = 400/12
unitprice.2 = 200/12

SAY                    /* Leave a blank line.                   */
SAY "Quantity  Unit price  Total price  Remarks"
SAY copies("-",58)     /* Places 58 hyphens across the display. */
DO item = 1 to 2    /* For each item, determine the total price, */
                    /* one time rounding conventionally, and     */
                    /* one time rounding down.                   */
   unitprice = FORMAT(unitprice.item,9,2)
   SAY FORMAT(qty.item,6,0),
       FORMAT(unitprice,7,2),
       FORMAT(qty.item * unitprice,10,2),
       "   Rounding conventionally"

   unitprice = TRUNC(unitprice.item,2)
   SAY FORMAT(qty.item,6,0),
       FORMAT(unitprice,7,2),
       FORMAT(qty.item * unitprice,10,2),
       "   Rounding down"
END
```

When this program is run, the following will be written to STDOUT.

```
Quantity  Unit price  Total price  Remarks
----------------------------------------------------------
   500       33.33      16665.00    Rounding conventionally
   500       33.33      16665.00    Rounding down
   500       16.67       8335.00    Rounding conventionally
   500       16.66       8330.00    Rounding down
```

Notice that the TRUNCATE function changed the unit price of the second item from 16.67 to 16.66, with the resulting $5 difference in total price.

For more information on the DO...END instruction, see Chapter 5, "Using REXX Instructions" on page 45.

Using String Operators

String operators perform concatenation. Concatenation simply means joining strings together. Whether a space is placed between concatenated terms depends on the operator you use.

- If you leave one or more blanks between the terms of an expression, REXX concatenates them with a single blank.

- If you put the terms together with no intervening blanks, this is called an *abuttal*. This operation simply joins the terms with no blanks.

- If you use the concatenation operator, ||, you also join two strings without a blank. This operator allows you to concatenate strings where abuttal will not work, such as joining variables. It also can be used in place of abuttal when you want to be explicit about how strings are to be joined.

In summary, the concatenation operators are:

Operator	Operation
Blank(s)	Concatenate terms with one blank in between
||	Concatenate terms without a blank (force abuttal)
Abuttal	Concatenate without an intervening blank

Concatenation examples:

```
SAY "slow"||"coach"    /* This displays 'slowcoach'. */

SAY "slow"     "coach" /* This displays 'slow coach'.*/

adjective = "slow"
SAY adjective"coach"   /* This displays 'slowcoach'. */
                       /* (using abuttal).          */

SAY adjective  "coach" /* This displays 'slow coach'.*/
SAY "("adjective")"    /* This displays '(slow)'.    */

SAY 4||5               /* This displays '45'.        */
SAY 4 5                /* This displays '4 5'.       */
tens = 4
units = 5
SAY tens||units        /* This displays '45'.        */
SAY tensunits          /* This displays 'TENSUNITS'  */
                       /* (because abuttal           */
                       /*  produces a new            */
                       /*  symbol).                  */

SAY (4||5) / 3         /* This displays '15'.        */
```

Concatenation works with numeric and nonnumeric strings.

Using Comparison Operators

Comparison operators test data, rather than change or control it directly. An earlier example in this book included the comparison operation:

```
IF who = " " THEN SAY...
```

The result of this comparison controls what happens next.

Operator	Operation
= =	True if terms are strictly equal (identical)
=	True if the terms are equal (numerically or when padded)
\ = =	True if the terms are NOT strictly equal
¬ = =	True if the terms are NOT strictly equal
\ =	Not equal (inverse of =)
¬ =	Not equal (inverse of =)
>	Greater than
<	Less than
> >	Strictly greater than
< <	Strictly less than
> <	Greater than or less than (same as not equal)
< >	Less than or greater than (same as not equal)
> =	Greater than or equal to
\ <	Not less than
¬ <	Not less than
> > =	Strictly greater than or equal to
\ < <	Strictly NOT less than
¬ < <	Strictly NOT less than
< =	Less than or equal to
\ >	Not greater than
¬ >	Not greater than
< < =	Strictly less than or equal to
\ > >	Strictly NOT greater than
¬ > >	Strictly NOT greater than

Note: The symbols \ and ¬ are synonymous. Either may be used as a <u>not</u> symbol. Usage is a matter of availability or personal preference.

Primarily, the comparison operators let you test if a term is equal to, not equal to, greater than, or less than some other term of the expression. By combining comparison operators you can test if terms are greater than or equal to, not greater than or equal to, and so forth. You can combine comparison operators to test terms.

The equal sign (=) can have different meanings in REXX depending on its position in a clause. The following is an example:

```
amount = 5              /* The variable AMOUNT is assigned the value 5.*/

SAY amount = 5          /* Compare the value of AMOUNT with 5.         */
                        /* If they are the same, say '1';              */
                        /* otherwise, say '0'.                         */
```

The result of a comparison expression is either 1 (indicating the comparison is true) or 0 (indicating the comparison is not true). The following are examples:

```
SAY 5 = 5               /* This displays "1".            */
SAY 5 <> 5              /* This displays "0".            */
SAY 5 = 4               /* This displays "0".            */
SAY 2 + 2 = 4           /* This displays "1".            */
SAY 2 + 2 = 5           /* This displays "0".            */

howmuch = 2 + 3         /* This assigns the sum of 2 and 3  */
                        /* to the variable HOWMUCH.         */

SAY "apples" = "oranges"  /* This displays "0".            */

fruit = "oranges"       /* This assigns the string "oranges" */
                        /* to the variable FRUIT.            */

SAY howmuch fruit       /* This displays "5 oranges".        */

SAY howmuch fruit = "4 oranges"   /* This displays "0".        */
SAY howmuch fruit = "5 plums"     /* This displays "0".        */
SAY howmuch fruit = "5 oranges"   /* This displays "1".        */
```

If you compare a number with a character string which is not a number, REXX will compare them as character values.

```
SAY 5 = 'Five'                     /* This displays "0".        */
SAY '00000F00' = '00000E00'        /* This displays "0".        */
SAY '0' =         '00000E00'       /* This displays "1".        */
SAY '0E000000' = '000000E0'        /* This displays "1".        */
```

Note that the last two examples use numbers in exponential notation.

Using the NUMERIC FUZZ Instruction: The NUMERIC FUZZ instruction can be used to control the precision used to make comparisons. Sometimes comparisons are too precise, as shown in the following examples:

```
/*  Expressions evaluated using default REXX precision.       */
SAY 1 + 1/3                 /* This displays '1.333333333'.   */
SAY 1 + 1/3 + 1/3 + 1/3     /* This displays '1.999999999'.   */
SAY 1 + 1/3 + 1/3 + 1/3 = 2 /* This displays '0'.             */
```

To make comparisons less precise than the default REXX arithmetic, you can use the NUMERIC FUZZ instruction.

The following are examples:

```
/*  These are expressions evaluated without approximation allowed. */
SAY 1 + 1/3 + 1/3 + 1/3 = 2        /* This displays '0'.        */
SAY 1 + 1/3 + 1/3 + 1/3 < 2        /* This displays '1'.        */
/*  These are expressions evaluated with approximation allowed.   */
NUMERIC FUZZ 1
```

```
SAY 1 + 1/3 + 1/3 + 1/3 = 2        /* This displays '1'.      */
SAY 1 + 1/3 + 1/3 + 1/3 < 2        /* This displays '0'.      */
```

Using the NUMERIC FUZZ Built-in Function: You can check the current setting
of NUMERIC FUZZ by using the FUZZ function. FUZZ will return 0, by default.
This means that 0 digits will be ignored during the comparison operation.

```
/*  Display the current NUMERIC FUZZ setting                        */
SAY 'The current FUZZ setting is' FUZZ()  /* Displays '0', by default. */
```

Using Strict Comparison Operators

By using strict comparison operators you can specify character-by-character
comparison, with no padding of either of the strings. The operators do not try to
perform numeric comparisons because they test for an exact match between the
two strings.

To find out if two strings are identical, use the strictly equal operator (==).

```
/* Set the value of variables Y and Z, and make comparisons    */
x="2"
z="+2"

SAY y = z                          /* This displays '1' true.   */
SAY y ¬= z                         /* This displays '0' false.  */
SAY y == z                         /* This displays '0' false.  */
SAY y ¬== z                        /* This displays '1' true.   */
```

You can also determine whether two strings are strictly greater than or strictly less
than each other by using the strictly greater than (>>) and strictly less than (<<)
operators. The following are examples:

```
SAY "cookies" >> "carrots"         /* This displays '1' true.   */
SAY "$10" >> "nine"                /* This displays '0' false.  */
SAY "steak" << "fish"              /* This displays '0' false.  */
SAY " steak" << "steak"            /* This displays '1' true.   */
```

The last comparison shows that " steak" is strictly less than "steak" since the
blank is lower in the sequence of characters.

The strict comparison operators are especially useful if you are interested in leading
blanks, trailing blanks, and zeros that are not significant zeros.

Using Logical Operators

The logical operators change and combine expressions. To change an expression
by reversing its evaluated result, use the NOT operator. To combine comparisons
to get the overall true or false value of more than one condition, use the logical
operators AND and OR.

The logical operators are shown in the following table:

Operator	Operation	Result
&	AND	Returns 1 if both terms are true
|	Inclusive OR	Returns 1 if either them is true
&&	Exclusive OR	Returns 1 if either, but not both, is true
**Prefix **	Logical NOT	1 becomes 0 and 0 becomes 1
Prefix ¬	Logical NOT	1 becomes 0 and 0 becomes 1

Using the AND Operator

To write an expression that is true when every one of a set of comparisons is true, use the AND (&) operator. The following is an example:

```
IF ready = "YES" & steady = "RIGHT"
THEN SAY "GO"
```

To result in 1 and continue with the THEN clause, both conditions must be met. Otherwise, nothing will happen.

Here are some other examples:

```
SAY (3=3) & (5=5)          /* This displays '1'.        */
SAY (3=4) & (5=5)          /* This displays '0'.        */
SAY (3=3) & (4=5)          /* This displays '0'.        */
SAY (3=4) & (4=5)          /* This displays '0'.        */
```

Using the OR Operator

To write an expression that is true when at least one of a set of comparisons is true, use the inclusive OR (|) operator. The following is an example:

```
IF ready = "YES" | steady = "RIGHT"
THEN SAY "GO"
```

In this case, if either or both expressions are true, GO will be displayed.

Here are some other examples:

```
SAY (3=3) | (5=5)          /* This displays '1'.        */
SAY (3=4) | (5=5)          /* This displays '1'.        */
SAY (3=3) | (4=5)          /* This displays '1'.        */
SAY (3=4) | (4=5)          /* This displays '0'.        */
```

Using Exclusive OR: The exclusive OR (&&) operator evaluates to 1 when one and only one of the expressions in the comparison is true. The following is an example:

```
city = 'NEW YORK'
state = 'NJ'
local = 'NO'
IF city = 'NEW YORK' && state = 'NJ'
THEN local = 'YES'
SAY local                  /* This displays 'NO'.       */
```

Using the NOT Operator

The NOT operator is placed in front of a term and changes the value from true (1) to false (0) or false to true. REXX uses two different characters as the NOT operator: ¬ and \. These have identical meanings; you can use either one. REXX defines two different characters for the same meaning to accommodate the differences between the many systems and keyboards REXX supports.

```
SAY ¬ 0                  /* This displays '1'.        */
SAY ¬ 1                  /* This displays '0'.        */
SAY ¬ 2                  /* This gives a syntax error. */

SAY ¬ (3 = 3)            /* This displays '0'.        */

/*                                                   */
fruit = "oranges"        /* This assigns "oranges" to */
                         /* the variable FRUIT.       */

SAY fruit = "oranges"    /* This displays '1'.        */
SAY fruit = "apples"     /* This displays '0'.        */
SAY ¬(fruit = "apples")  /* This displays '1'.        */
SAY ¬(fruit = "oranges") /* This displays '0'.        */
```

The NOT operator reverses the result of any comparison it precedes. An expression that REXX otherwise evaluates as 1 is negated (its evaluation changed to 0) when you put the NOT operator in front of it. And, similarly, an expression that REXX would otherwise evaluate as 0 is evaluated as 1 when preceded by a NOT operator.

Combining Comparisons

Long expressions can be formed using combinations of operators, as shown in the following example:

```
IF ((savings_balance > 10000 | checking_balance > 5000),
   & customer_years > 5) && customer_status = 'SPECIAL' THEN
   SAY 'This customer is a candidate for changing status'
```

The logical expression be TRUE, evaluate as 1, if:

- Either of the two balances is over the limit, and the customer_years variable is over 5, while the customer_status variable is not SPECIAL

- Both balances are below the limits

- The customers_years variable is not over 5, while the customer_status variable is SPECIAL.

Using Function Calls as Expressions

REXX provides many built-in functions which can be used in expressions. A list of these functions is in Appendix B, "REXX Built-in Functions" on page 139. For a complete description of all of the REXX built-in functions, see the *REXX/400 Reference*. It provides additional information on using built-in and other types of functions.

A built-in function will always return a value.

A *function call* can be written anywhere in an expression. The interpreter performs the computation named by the function and returns a result, which is then used in the expression in place of the function call. If REXX finds

symbol (*expression*...)

in an expression, with no space between the last character of the symbol and the left parenthesis, the interpreter assumes that *symbol* is the name of a function and that this is a call to the SYMBOL function.

The result returned by a function depends on what is inside the parenthesis. When the value of the function has been calculated, the result is put back into the expression in place of the function call.

Using Expressions in Instructions

The *REXX/400 Reference* contains an alphabetic list of all REXX instructions and their syntax. You can use an expression if *expression* is specified.

Using Expressions as Commands

REXX processes your program one clause at a time. The interpreter examines each clause to determine if the clause is a keyword instruction, a variable assignment, a label, or a null clause. If it is none of these, then REXX evaluates the entire clause as an expression and passes the result on to the current command environment. It is left to the current command environment to process the command.

Chapter 7, "Understanding Commands and Command Environments" on page 79 discusses this in more detail.

Chapter 5. Using REXX Instructions

REXX provides an extensive set of instructions which control your program. This chapter provides you with information about using REXX instructions, including:

- Learning About Keyword Instructions
- Using Structured Programming
- Understanding Programming Style
- Using the INTERPRET Instruction
- Using a REXX Program Instead of a CL Program.

Learning About Keyword Instructions

REXX instructions are recognized as instructions by keywords. When a REXX keyword begins a clause, and it is not an assignment, REXX interprets it as an instruction. Keywords themselves are not reserved, because they can be used in other places within a clause. The complete set of keywords is listed in Appendix A, "REXX Keywords" on page 137 and are defined in the *REXX/400 Reference*.

Instructions like IF, SELECT, and DO also have subkeywords. For example, the IF instruction can begin a clause which also might include the subkeyword THEN or ELSE. THEN or ELSE at the beginning of a clause will not be recognized as part of an instruction, but when used after the IF keyword they will be recognized as part of the complete instruction.

Using Structured Programming

REXX programs can contain many different statements. Primarily, they will be either a single list of instructions or a number of lists of instructions connected by instructions indicating which list should run next.

The instructions that direct how the program runs are called *control instructions*. The direction a program can take by using control instructions can be any of the following:

Branches
Branches allow you to select one of several lists of instructions to run. Branches are controlled by the IF and SELECT instructions.

Loops
Loops allow you to repeat a list of instructions for a specified number of times or until a stated condition is satisfied while a condition is true. Loops are controlled by the DO instruction.

Calls
Calls can be made to subroutines to perform a separate, well-defined task that might be necessary within a larger task. You use the CALL instruction to tell REXX to run a subroutine, then return and run the next sequential instruction. Calls to subroutines are discussed in Chapter 8, "Using REXX Functions and Subroutines" on page 99.

Transfers of Control Transfers of control are made in order to continue the program from a different point within the program itself. Transfers of control are controlled by the SIGNAL instruction.

This type of control is discussed in "Using Condition Trapping" on page 133.

Exits   Exits leave your program unconditionally and stop all processing. The EXIT instruction exits the program and can also pass back a value when written in the form `EXIT` *expression*. The valid values for *expression* are determined by how the program was called. For more information on exits and the EXIT instruction, see the *REXX/400 Reference*.

A single program can contain one or all of the control instructions. Controls can also be nested within controls.

In the sections that follow, the control instructions are discussed along with ways to combine them.

Using Branches

Branches allow you to control how your program runs by setting conditions and indicating what set of instructions should run under those conditions. Branches can be built using the IF instruction along with the THEN and ELSE keyword. The SELECT instruction can also be used.

Using the IF Instruction

The following example shows a very simple branch:

```
/* This is an example of a simple IF...THEN...ELSE instruction.    */

IF a = c THEN
     SAY "It's true; a does equal c."
ELSE
     SAY "No, a does NOT equal c."
```

If `a = c` the `SAY "It's true; a does equal c."` instruction is run. If `a = c` is not true, the `SAY "No, a does NOT equal c."` instruction is run.

Using IF...THEN adds a branch of instructions to run when the controlling expression is true.

```
IF expression
THEN instruction
```

By adding the ELSE subkeyword, you can indicate two branches. One set of instructions will run when the condition is true. One set of instructions will run when the condition is not true.

```
IF expression
THEN instruction1
ELSE instruction2
```

To put a list of instructions, rather than a single instruction, after the THEN subkeyword or ELSE subkeyword, you must group the instructions using the DO instruction with the END subkeyword.

```
DO
   instruction1
   instruction2
   instruction3
END
```

A DO...END instruction tells REXX to treat the instructions between them as a single instruction. Without a DO...END grouping, only the first instruction following the THEN or ELSE subkeyword will run.

```
/* This is an example of an IF-THEN-ELSE with multiple instructions  */
/* in both the true part and the false part.                         */

IF a = c
THEN
  DO
   SAY "It's true; a does equal c."
   SAY "This message is in the true part of the instruction"
  END
ELSE
  DO
   SAY "No, a does NOT equal c."
   SAY "This message is in the false part of the instruction"
  END
```

It is possible to use multiple IF instructions to control program flow without DO...END instruction groups. This program shows how the IF...THEN...ELSE instruction may be used together without DO and END. In REXX, an ELSE subkeyword belongs to the first IF instruction if there is no other ELSE subkeyword associated with it.

```
/* Suppose the variables weather and distance are set to some    */
/* correct values.  This program will use the two variables to   */
/* suggest a plan for the day.                                   */

IF weather = "FINE"
THEN
   IF distance = "NEAR"
   THEN SAY "We will walk to our customer's office"
   ELSE SAY "We will drive to our customer's office"
ELSE SAY "We will stay inside and do paperwork today"
```

Using the SELECT Instruction
A simple IF...THEN...ELSE instruction allows two sets of instructions to be run depending on the condition. The SELECT instruction expands this to many choices depending on the condition.

```
SELECT
   WHEN expression1 THEN instruction1
   WHEN expression2 THEN instruction2
   WHEN expression3 THEN instruction3

⋮

   OTHERWISE
      instruction
      instruction
      instruction

⋮

END
```

The SELECT instruction performs the following:

- If *expression1* is true, *instruction1* is run. After this, processing continues with the instruction following the END.

- If *expression1* is not true, then *expression2* is tested. If it evaluates as true, then *instruction2* is run and processing continues with the instruction following the END.

- If none of the expressions are true, then processing continues with the instruction following the OTHERWISE subkeyword.

The OTHERWISE subkeyword must be used if there is any possibility that all of the WHEN expressions could be evaluated as not true.

As with the IF instruction, to tell REXX to run a list of instructions which follow a THEN subkeyword, you must enclose those instructions within a DO...END group. A DO...END group is not required after the OTHERWISE subkeyword.

Example 1: Here is a simple example of the SELECT instruction.

```
/*  This program requests the user to enter a whole      */
/*  number from 1 through 12 and replies with the        */
/*  number of days in that month.                        */
/* ------------------------------------------------------ */
/*  Get input from user.                                 */
/* ------------------------------------------------------ */
DO UNTIL DATATYPE(Month,WHOLE),
     & month >= 1 & month <= 12
   SAY "Enter the month as a number from 1 through 12"
   PULL month
END
/* ------------------------------------------------------ */
/*  Calculate the days in month.                         */
/* ------------------------------------------------------ */
SELECT
   WHEN month =  9 THEN days = 30
   WHEN month =  4 THEN days = 30
   WHEN month =  6 THEN days = 30
   WHEN month = 11 THEN days = 30
   WHEN month =  2 THEN days = "28 or 29"
   OTHERWISE
   days = 31
END

SAY "There are" days "days in Month" month
```

Example 2: This program uses the SELECT instruction, along with other controls. It requests the user to provide the age and gender of a person. As a reply, it displays the status of that person. People under the age of 5 are BABIES. Those aged 5 through 12 are BOYS or GIRLS. Those aged 13 through 19 are TEENAGERS. All the remaining are either a MAN or a WOMAN.

```
/*-------------------------------------------------------- */
/* Get input from user.                                    */
/*-------------------------------------------------------- */
DO UNTIL DATATYPE(age,NUMBER) & age >= 0
   SAY "What is the person's age?"
   PULL age
END

DO UNTIL gender = "M" | gender = "F"
   SAY "What is the person's gender (M or F)?"
   PULL gender
END
/*-------------------------------------------------------- */
/* DETERMINE STATUS                                        */
/*                                                         */
/* Input:                                                  */
/* AGE         Assumed to be 0 or a positive number.       */
/* GENDER        "M" is male.  Anything else is female.    */
/*                                                         */
/* Result:                                                 */
/* STATUS      Possible values:  BABY, BOY, GIRL, TEENAGER, */
/*             MAN, WOMAN.                                  */
/*-------------------------------------------------------- */
SELECT
  WHEN age < 5 THEN status = "BABY"
  WHEN age < 13 THEN
    DO
      IF gender = "M"
      THEN status = "BOY"
      ELSE status = "GIRL"
    END
  WHEN age < 20 THEN status = "TEENAGER"
  OTHERWISE
    IF gender = "M"
    THEN status = "MAN"
    ELSE status = "WOMAN"
END

SAY "This person should be counted as a" status
```

Example 3: The following two programs show the same operation. The first uses
the SELECT instruction, and the second uses IF...THEN...ELSE instructions.

```
/* This program requests the user to enter two words and  */
/* says which one is higher, using the SELECT instruction.*/

SAY "Enter two words"
PULL word1 word2 .
SELECT
   WHEN word1 = word2
   THEN SAY "The words are the same",
            "or numerically equal"
   WHEN word1 > word2
   THEN SAY "The first word is higher"
   OTHERWISE
   SAY "The second word is higher"
END
```

Example 4: The same result can be achieved using the IF instruction.

```
/* This program requests the user to supply two words and */
/* says which is higher, using the IF instruction.       */

SAY "Enter two words"
PULL word1 word2 .
IF word1 = word2
THEN SAY "The words are the same",
         "or numerically equal"
ELSE DO
   IF word1 > word2
   THEN SAY "The first word is higher"
   ELSE SAY "The second word is higher"
END
```

The choice is up to you, however, SELECT will generally be the easier instruction to use where more than two conditions are needed.

Using the NOP Instruction

A THEN or ELSE subkeyword must be followed by an instruction. In those cases where you intend for nothing to run, you can use the NOP instruction. The following two examples show how to use NOP with the SELECT instruction and the IF instruction.

Using NOP with SELECT:

```
SAY "Where is the harbor?"
PULL where
SELECT
   WHEN where = "AHEAD" then NOP
   WHEN where = "PORT BOW" THEN SAY "Turn left"
   WHEN where = "STARBOARD BOW" THEN SAY "Turn right"
   OTHERWISE SAY "Not understood"
END
```

Using NOP with IF:

```
IF gas = "FULL" & oil = "SAFE" & window = "CLEAN"
THEN NOP
ELSE SAY "Find a gas station."
```

Using Loops

The DO...END instruction lets you group instructions so the list of instructions runs once. The DO instruction used with other subkeywords lets your instructions run repeatedly. When a program repeats a list of instructions, it is looping. A loop can occur a specific number of times, as long as a particular condition is true, until some condition is satisfied or until the user wants it to stop.

Looping with Counters

To repeat a loop a specific number of times use:

```
DO exprr
   instruction1
   instruction2
   instruction3

   ⋮
END
```

where

exprr

 is the **expr**ession for **r**epetitor which must evaluate to a whole number. This is the number of times the loop will be repeated.

The following is an example:

```
/* This is the simplest form of a repetitive DO in REXX.              */

DO 5
   "SNDUSRMSG MSG('You will see this message five times')"
END
```

Using Indexes: Each pass through a loop can be numbered. That number can be used as a variable in your program.

```
DO name = expri TO exprt
   instruction1
   instruction2
   instruction3

   ⋮
END
```

where

name

 is the control variable called a *counter*. You can use it in the body of the loop. Its value is changed each time you pass through the loop, and may be changed by the program itself.

expri

 is the **expr**ession for the **i**nitial value. It is the value you want the counter to have the first time through the loop.

exprt

 is the **expr**ession for the **t**o value. It is the value you want the counter to have the last time through the loop. The loop will end if the next iteration will put the counter above the *exprt* value.

Example 1: The following program shows how REXX programs can perform a loop using an integer loop counter.

```
DO i = 1 to 5
   "SNDUSRMSG MSG('This is message" i "of five')"
END
```

Example 2: The following example shows how the iterations can be affected by the program itself, in this case by reassigning the value of i.

```
DO i = 1 to 3
   i = 5
END              /* This program will only run one time. */
```

Example 3: You can use the counter to calculate something different each time through the loop. In the following example, the counter is called COUNT and calculates the width of each row of stars.

```
/* This program displays a triangle.            */
/* The user is asked to specify the height of the*/
/* triangle.                                     */

SAY "Enter the height of the triangle",
    " (a whole number between 3 and 15)."
PULL height
SELECT
   WHEN \DATATYPE(height,WHOLE) THEN SAY "Try again"
   WHEN height < 3 THEN SAY "Too small"
   WHEN height > 15 then say "Too big"
   OTHERWISE

   DO count = 1 to height by 2      /* Draw a triangle.*/
   SAY CENTER(COPIES('*',count),height)
   END

   SAY "This is a triangle with a width of "height'.'
END
```

After you leave the loop, you can still refer to the counter. It will always exceed the value of the TO expression, *exprt.*

The counter can be incremented by values other than 1, which is the default. To specify some other value, use the following:

```
DO name = expri TO exprt BY exprb
⋮
END
```

where

exprb

   is the **expr**ession for **b**y and gives the number that is to be added to name at the bottom of the loop.

The following is an example of a DO FOREVER loop:

```
DO name = expri BY exprb
⋮
END
```

Example 4: REXX allows decimal numbers, as well as integers, as loop indexes.
Also, the index can be stepped in the negative direction. The following example
will loop ten times, in reverse:

```
DO i = 8.3 to 7.4 by -.1
   SAY 'The value of the loop index is' i
END
```

Looping Using Conditions

Loops can run based on conditions you set. DO WHILE loops run as long as a
condition is true. DO UNTIL loops run until a given condition is true. DO
FOREVER loops end with a LEAVE, RETURN, or EXIT instruction.

Using DO WHILE: To create a loop that repeats its list of instructions as long as
a given condition is true, use DO WHILE.

```
DO WHILE exprw
   instruction1
   instruction2
   instruction3

   ⋮
END
```

where

exprw
> is the **expr**ession for **w**hile and is an expression that must result in 0 or 1.

Example 1: The condition is tested at the top of the loop, before the instruction list
is run. If the condition is false, the instructions will not be run. If it is true, the
instructions will run.

```
/* This program uses a binary search to locate the zero of an      */
/* arbitrary increasing function, f.                               */

SAY "Enter the minimum and maximum ends of the range"
PULL min max
SAY "Enter the maximum allowable error"
PULL maxerr

DO WHILE max - min > maxerr
   guess = (max + min) / 2
   IF F(guess) > 0 then max = guess
      ELSE min = guess
   END
SAY "The final guess was" guess
```

Using DO UNTIL: To create a loop which runs until a condition is true, use DO
UNTIL.

```
DO UNTIL expru
   instruction1
   instruction2
   instruction3

   ⋮
END
```

where

expru

    is the **expr** for **u**ntil and is an expression that, when evaluated, must give a
    result of 0 or 1.

Example 1: When you use a DO UNTIL instruction, the test occurs at the bottom
of the loop. This means that the instruction list enclosed by the DO UNTIL will
always run at least once.

```
/* This program uses a binary search to locate the zero of an     */
/* arbitrary increasing function, f.                              */

SAY "Enter the minimum and maximum ends of the range"
PULL min max
SAY "Enter the allowable distance from 0 for the final function value"
PULL maxerr

guess = (max + min) / 2
DO UNTIL ABS(F(guess)) < maxerr
   IF F(guess) > 0
      THEN max = guess
   ELSE min = guess
guess = (max + min) / 2
END

SAY "The final guess was" guess "which produced a function value of" F(guess)
```

Example 2: DO UNTIL can be used to check input. The following example uses
the DATATYPE function to make certain that the user only enters a number:

```
/*  An example with numbers only.         */
DO UNTIL DATATYPE(entry,NUM)
   SAY "Enter a number"
   SAY "(or press Enter alone to quit):"
   PULL entry
   IF entry = "" THEN EXIT
END
```

This loop will always run at least once, even if the variable ENTRY is already a
number. It will continue to run until you either enter a number or press the Enter
key alone.

Using DO FOREVER: In some cases, the DO instruction may not be the place
where you want to set conditions. DO FOREVER is provided for such instances.
In the following example, the loop is ended by an EXIT instruction, which ends the
entire program.

```
SAY 'This program will add up a series of numbers which you give it'
SAY 'Enter a null line to end the program'

sum = 0
```

```
DO FOREVER
   SAY 'Enter a number'
   PULL number
   IF number = '' THEN EXIT
   sum = sum + number
   SAY 'Total so far is' sum
END
```

The LEAVE instruction immediately ends a loop, and will move processing to the instructions following the END subkeyword.

```
SAY 'This program will add up a series of numbers which you give it'
SAY 'Enter a null line to end the program'

sum = 0

DO FOREVER
   SAY 'Enter a number'
   PULL number
   IF number = '' THEN LEAVE
   sum = sum + number
END

   SAY 'The final total is' sum
```

Combining Iterative and Conditional Loops

REXX lets you combine repetitive and conditional loop controls on a single DO instruction. For example, suppose an array called `data` has been set up with elements 1 through 10, and we want to search for an entry that may be present.

```
DO i = 1 to 10 WHILE data.i <> target
END

IF i = 11 THEN SAY 'Sorry, target was not found'
   ELSE SAY 'Target was found in entry number' i
```

In an iterative loop, the loop will run until the loop counter variable passes the value of the **to** expression. When combining iterative and conditional loops, as in the example, the iterative part of the loop works the same way. So, in the example, if the item you are searching for is not found, the variable `i` will have the value 11 when the loop stops. If the item is found, the variable `i` will have the value of the item number containing the target.

Using the ITERATE Instruction

The ITERATE instruction bypasses all remaining instructions in the loop and test the ending conditions.

ITERATE can be introduced by a THEN or ELSE subkeyword, and REXX will proceed with the operations usually done at the bottom of the loop. If an UNTIL condition has been specified, it is tested. If a counter has been specified, it is incremented and tested. If a WHILE condition has been specified, it is tested.

If tests indicate that the loop is still active, usual processing then continues from the top of the loop.

Example 1: The following program shows how the ITERATE instruction is normally used with an IF or SELECT instruction.

```
DO j = 1 to limit by delta
   instruction1
   instruction2
   IF condition
   THEN DO
      instruction3
      instruction4
      ITERATE j
   END
   instruction5
   instruction6
END
```

Example 2: The following program asks for a list of words, which will be put into an array. If a number is entered, the number is not put into the array.

```
count = 0
DO FOREVER
   SAY 'Enter a word (no numbers allowed), or a null line to end.'
   PULL word
   IF DATATYPE(word,NUMBER) THEN ITERATE
   IF word = '' THEN LEAVE

   count = count + 1
   list.count = word
END

SAY count 'words were entered.  Here is the list.'
DO i = 1 to count
   SAY i list.i
END
```

Using the LEAVE Instruction

Conditional loops will continue to run as long as the condition is satisfied. The LEAVE instruction can be used to end processing of a loop, as shown in "Using DO FOREVER" on page 54. However, a program can be made up of loops within loops. You may want to be able to specify when control should pass from a particular loop. To do this, you will need to give the loop a name and use the LEAVE instruction. When the LEAVE instruction is run, control will move to the instructions which follow the END subkeyword, as shown in the following example:

```
DO outer = 1   /* OUTER is the name of the loop.  */
⋮
END
```

To specify this as the loop to leave, put the name of its counter after the LEAVE instruction. The following is an example:

```
DO outer = 1
⋮
     /* This is a DO FOREVER because BY 1 is the default.  */
   DO UNTIL DATATYPE(answer,WHOLE)
     SAY "Enter a number.",
         "When you have no more data, enter a blank line."
     PULL answer
     IF answer = "" THEN LEAVE outer
   END

 ⋮
   /* Process answer. */
END
/*  Come here when there is no more data.*/
```

Understanding Programming Style

REXX is a free format language. It has very few requirements for how source code is entered. The following programs illustrate how the REXX rules for formatting a program are very flexible. It shows the same IF...THEN...ELSE in four different ways.

REXX clauses must end with a semicolon, unless the clause ends at the end of the line. The semicolon is implied when each clause begins on a separate line.

Example 1: This example explicitly ends each clause with a semicolon. The entire program can be all on one line.

```
IF a=c THEN;DO; SAY 'a=c'; SAY 'This is hard to read';END;ELSE;DO;
SAY 'a \= c'; SAY 'This is hard to read'; END
```

Example 2: This example shows how the same instructions can be spread out over many lines. In this example, each line is a complete clause.

```
IF a=c
THEN
DO
   SAY 'a=c'
   SAY 'This is hard to read'
END
ELSE
DO
   SAY 'a\=c'
   SAY 'This is hard to read'
END
```

Example 3: This example shows each clause spread out over more than one line. To tell REXX to read to the end of a clause when it is spread over more than one line, use the continuation character, as shown.

```
IF,
a,
=,
c
THEN
DO
   SAY,
   'a=c'
   SAY,
   'This is harder to read'
END
ELSE
DO
   SAY,
   'a\=c'
   SAY,
   'This is harder to read'
END
```

Example 4: This example shows a compromise in style, using indentation to make the program easier to read.

```
IF a=c
   THEN DO
      SAY 'a=c'
      SAY 'This is easier to read'
   END
ELSE
   DO
      SAY 'a\=c'
      SAY 'This is easier to read'
   END
```

You can choose whatever programming style you prefer. It is a good idea to use indentation to make it easier to see the beginning and end of branches or loops.

Using the INTERPRET Instruction

The INTERPRET instruction evaluates an expression and processes it as if it were part of the text of the program.

Example 1: This example shows how interpret can be used to write a simple calculator program.

```
/* Simple calculator program.                              */
/* For instance, if the user enters  (1+2) * 3, the answer will be 9.*/

SAY 'Enter the expression you want to have evaluated'
PULL expr
INTERPRET "answer =" expr
SAY 'The answer is' answer
```

Example 2: In many cases where the interpret instruction is used, there is another way to reach the same result. For example, this program uses SELECT to choose which of several programs to call, based on the day of the week.

```
today = DATE(Weekday)
SELECT
   WHEN today = 'Sunday'    THEN CALL Sunday
   WHEN today = 'Monday'    THEN CALL Monday
   WHEN today = 'Tuesday'   THEN CALL Tuesday
   WHEN today = 'Wednesday' THEN CALL Wednesday
   WHEN today = 'Thursday'  THEN CALL Thursday
   WHEN today = 'Friday'    THEN CALL Friday
   WHEN today = 'Saturday'  THEN CALL Saturday
END
```

The same operation can be performed using the INTERPRET instruction, which makes it shorter.

```
today = DATE(Weekday)
INTERPRET "Call" today
```

Example 3: Since INTERPRET lets you create instructions dynamically, you can sometimes combine two instructions by using it. The same action taken in "Example 2" on page 23 can be accomplished by using the INTERPRET instruction within the loop.

```
DO i = count + 1 to 2 by -1
   INTERPRET "ARRAY."i "=ARRAY."i-1
END
count = count + 1          /* Update the new count.    */
array.1 = newvalue         /* Store the new value.     */
```

Using a REXX Program Instead of a CL Program

In many cases, REXX can perform tasks which might otherwise be done by CL programs. The first example below shows a simple CL program. Following it is a REXX program which performs the same operation.

```
PGM       PARM(&LIBRARY)
DCL       &LIBRARY *CHAR 10
  IF (&LIBRARY *EQ '          ') THEN(CHGVAR &LIBRARY 'MYLIB')
  DSPLIB &LIBRARY
ENDPGM
```

Here is the same operation written in REXX:

```
ARG libname                          /* Get the parameter (if any).*/
IF libname = '' THEN libname = "MYLIB" /* If none is provided, make  */
                                     /* the parameter "MYLIB".     */
"DSPLIB" libname                     /* Issue the DSPLIB command.  */
EXIT
```

This example uses commands which are discussed in Chapter 7, "Understanding Commands and Command Environments" on page 79.

Chapter 6. Using REXX Parsing Techniques

Parsing is a way of taking data from various input sources and assigning it to variables. This chapter covers the information you need to use parsing within a REXX program, including:

- Understanding Parsing
- Parsing with Patterns
- Using String Functions.

Understanding Parsing

Parsing lets you split an input string into component substrings which, in turn, are assigned to one or more variables.

The input string can come from any of the following sources:

- An argument passed to the program
- A command line argument
- An item on the REXX external data queue
- Input from the file STDIN
- The contents of a variable
- The result of an expression
- System information.

This variety of input sources, along with the many ways that the input can be analyzed and controlled, makes parsing one of the most powerful features of REXX.

Parsing is performed by using a REXX clause made up of two parts:

1. The **PARSE instruction**, in any of its various forms, denotes the source of the input text.

2. A **parsing template** is a list of symbols that describe how the input text is to be broken up. In its simplest form, the template is a list of variables to which words of input text are assigned.

Using the PARSE Instruction

The form of PARSE instruction that you use depends on the source of input. There are seven PARSE forms which correspond to the potential sources of input. The optional UPPER subkeyword will convert the input text into uppercase. UPPER can be used with any form of PARSE.

The ways the PARSE instruction can be used are shown below.

Using PARSE ARG: The PARSE ARG instruction takes as input an argument string passed to the current REXX program, subroutine, or function in which the PARSE ARG instruction is used. The following is an example:

```
/* The argument(s) passed to a program can be parsed.    */
PARSE ARG arg1
SAY 'The first argument passed to this program is' arg1
```

The instruction PARSE UPPER ARG can also be given as simply, ARG. The following is an example:

```
    /* The ARG instruction is a short version of PARSE UPPER ARG.    */

PARSE UPPER ARG input1
ARG input2

SAY 'Parse Upper Arg got' input1
SAY 'Arg got' input2

IF input1 == input2 THEN
   SAY 'The two different instructions work the same'
ELSE
   SAY 'You will never see this message.'

PARSE ARG input3
SAY 'PARSE ARG got' input3                 /* Displays the mixed case  */
                                           /* version of the input.    */
```

Using PARSE PULL: The PARSE PULL instruction takes as input the next item on the REXX external data queue. If the queue is empty, PARSE PULL instead reads a line from the file STDIN. This means that PARSE PULL can be used to accept interactive input from a program user, as shown in the following example:

```
/* Keyboard input may be read two different ways.  The first,        */
/* PARSE PULL will read a line from the REXX queue if anything is    */
/* there, and from the keyboard if the queue is empty.               */
SAY 'Enter a line of input from your keyboard.'
PARSE PULL line
SAY 'You entered' line
```

The instruction PARSE UPPER PULL, which translates the input to uppercase characters, can also be given as simply, PULL, as shown in the following example:

```
/*   The PULL instruction is a short version of PARSE UPPER PULL.   */

test = 'This is a test'
DO 3
   PUSH test        /* Puts three lines on the external data queue. */
END

PARSE UPPER PULL input1
PULL input2

SAY 'Parse Upper Pull got' input1  /* This displays 'THIS IS A TEST'.*/
SAY 'Pull got' input2              /* This displays 'THIS IS A TEST'.*/

IF input1 == input2 THEN
```

```
    SAY 'The two different instructions work the same'
ELSE
SAY 'You will never see this message.'

PARSE PULL input3
SAY 'Parse Pull got' input3          /* This displays 'This is a test'.*/
```

Using PARSE LINEIN: The PARSE LINEIN instruction takes as input the next line from STDIN, regardless of whether there is anything in the REXX external data queue. Otherwise, it performs the same operations as PARSE PULL.

The following is an example:

```
PUSH 'Parse linein will not touch this line.'
SAY 'Enter another line of input from your keyboard.'
PARSE LINEIN line
SAY 'You entered' line
PARSE PULL firstline
        /* This will contain 'Parse linein will not touch this line.'. */
```

Using PARSE VALUE: The PARSE VALUE instruction takes as input the evaluated result of an expression.

The following is an example:

```
PARSE VALUE 1 + 2 with result
SAY 'The value' result 'was calculated by the Parse value instruction'
```

The WITH subkeyword separates the expression from the template. Templates are discussed further in "Using Templates" on page 64.

Using PARSE VAR: The PARSE VAR instruction takes as input the contents of a named variable.

The following is an example:

```
var1 = 1 + 2
PARSE VAR var1 newvariable
SAY 'The value' newvariable 'was assigned to Var1'
                    /* This displays 'The value 3 was assigned to Var1'. */
```

The contents can be converted to uppercase by using the UPPER keyword:

```
/* PARSE VAR EXAMPLE  */
mixed='This Data Started Out In Mixed Case'
PARSE UPPER VAR mixed uppervar
SAY 'The uppercase version of' mixed
SAY 'is' uppervar
```

This will display:

```
The uppercase version of This Data Started Out In Mixed Case
is THIS DATA STARTED OUT IN MIXED CASE
```

Using PARSE VERSION: The PARSE VERSION instruction takes as input a string that describes the name, language level, and version date of the REXX interpreter itself.

The following is an example:

```
PARSE VERSION ver
SAY 'The version of REXX you are using is' ver
```

Using PARSE SOURCE: The PARSE SOURCE instruction takes as input a string that described the REXX program that is currently being run.

The following is an example:

```
/* Information about the program being run is available.        */
PARSE SOURCE src
SAY 'The identifying information for this program is' src
```

Using Templates

The examples shown with each of the PARSE instruction variations have used the simplest form of template, a single variable. But there is far more to parsing than simply capturing data from a given source and assigning it in its entirety to a single variable. Parsing lets you selectively assign specific components of the input string to a number of variables.

By default, REXX breaks up the input string into words and assigns them, in order, to the given list of variables. Here is an example of a PARSE instruction with a template of five variables:

```
/* Parsing a string into five variables.                       */
PARSE VALUE "I love my AS/400 system" WITH var1 var2 var3 var4 var5
/* At this point the variables are set as follows:             */
/* var1 = "I"                                                  */
/* var2 = "love"                                               */
/* var3 = "my"                                                 */
/* var4 = "AS/400"                                             */
/* var5 = "system"                                             */
```

In this case, the number of variables in the template exactly corresponds to the number of words in the input string. Notice what happens when there are fewer words in the input string than there are variables in the template:

```
/* Parsing the same string into six variables.                 */
PARSE VALUE "I love my AS/400 system" WITH var1 var2 var3 var4 var5 var6
/* At this point the variables are set as follows:             */
/* var1 = "I"                                                  */
/* var2 = "love"                                               */
/* var3 = "my"                                                 */
/* var4 = "AS/400"                                             */
/* var5 = "system"                                             */
/* var6 = ""                                                   */
```

The sixth variable is assigned an empty string. It is important to note that *all* variables listed in a parsing template are assigned new values. If no value is available for a given variable, it is assigned a null (empty) string.

The reverse occurs if there are fewer variables in the template than words in the input string. The last variable in the template is assigned *all* of the remaining words in the string:

```
/* Parsing the same string into just two variables.           */
PARSE VALUE "I love my AS/400 system" WITH var1 var2
/* At this point the variables are set as follows:            */
/* var1 = "I"                                                 */
/* var2 = "love my AS/400 system"                             */
/* Variables var3, var4, var5 are not changed.                */
```

In this example, variable VAR1 is assigned the word "I" but this time VAR2, the last template variable, is assigned the entire remainder of the string.

Using Placeholders

It is not necessary to have a variable for every word in a parsing template. To ignore a string at a given position, use a period as a dummy variable, which is called a *placeholder.*

```
/* Select just the fourth and sixth words.                    */
sentence = "I want to learn to program in REXX."
PARSE VAR sentence . . . word4 . word6 .
/* The variables now read:                                    */
/* sentence = "I want to learn to program in REXX."           */
/* word4 = "learn"                                            */
/* word6 = "program"                                          */
```

A period at the end of a template can be used to discard all unwanted words at the end of a string. If you do not have a period at the end of the template, you will get:

```
/* Discard the end of a string.                               */
PARSE VALUE "Programming is easier in REXX" WITH word1 word2
/* The result is:                                             */
/* word1 = "Programming"                                      */
/* word2 = "is easier in REXX"                                */
```

But that leaves too many words in WORD2. To get just the second word and discard the remaining words, you can use a period at the end of the template.

```
PARSE VALUE "Programming is easier in REXX" WITH word1 word2 .
/* Now the result is:                                         */
/* word1 = "Programming"                                      */
/* word2 = "is"                                               */
```

Parsing Variables and Expressions

In practice, of course, a program is rarely called on to parse a literal string. Usually the purpose is to analyze information that is unknown when the program is written. This is done by parsing variables.

The following is an example:

```
/* Parsing a variable.                                        */
PARSE VALUE "I love my AS/400 system" WITH var1 var2
/* var1 = "I"                                                 */
/* var2 = "love my AS/400 system"                             */
/* Variables var3, var4, var5 are unchanged                   */
/* Now parse the variable var2                                */
PARSE VAR var2 var3 var4 var5
```

```
/* Now the variables are as follows:                         */
/* var1 = "I"                    (no change)                  */
/* var2 = "love my AS/400 system"  (no change)               */
/* var3 = "love"                                              */
/* var4 = "my"                                                */
/* var5 = "AS/400 system"                                     */
```

Note closely the difference in syntax for the PARSE VAR instruction. The first variable named is the source of the input string, and the variables that follow make up the template.

The source variable can also be used in the template. It can be reassigned a new value in the same PARSE instruction for which it provides the input string.

```
/* Parsing a variable also named in the template.            */
var1 = "I love my AS/400 system"
PARSE VAR var1 var1 var2
/* Now:                                                       */
/* var1 = "I"                                                 */
/* var2 = "love my AS/400 system"                             */
/* And again:                                                 */
PARSE VAR var2 var2 var3

/* Now:                                                       */
/* var1 = "I"                    (no change)                  */
/* var2 = "love                                               */
/* var3 = "my AS/400 system"                                  */
```

The following is another example of PARSE VAR:

```
/* Pulling off one word at a time and leaving the rest in the  */
/* original variable.                                          */
DO i = 1 to words(var1)
   PARSE VAR var1 var2 var1
   SAY var2
END
/* This loop will display each word in var1 one at a time and  */
/* keep the rest in the original variable.                     */
```

Using Special Parsing Techniques

The parsing capabilities of REXX allow you many different ways to work with sources of input text. Some special parsing methods can expand the ways you can use parsing.

Parsing With PARSE SOURCE

The PARSE SOURCE instruction identifies the program which is currently running. The following examples give some additional uses of the PARSE SOURCE instruction within a program.

The following example program gives a brief help message, which includes the name of the program. Instead of hard-coding the name, it is obtained by the REXX PARSE SOURCE instruction. If the program is renamed, the help message will not have to be changed.

```
/* Using REXX PARSE SOURCE.              */
PARSE SOURCE . . membername filename libname

SAY 'This is a REXX program named' membername 'in' libname'/'filename'.'
SAY 'This program demonstrates how the REXX "Parse Source" instruction'
SAY 'can be used to find out the name of the program.'
```

The following example program uses the PARSE SOURCE instruction to find out whether it was called from another program or from the command line. It calculates the sum of two numbers, and if it was called from another program, it returns the sum as the function value.   If it was called as a command, it shows the sum on the display.

```
/* Using PARSE SOURCE.                           */

PARSE SOURCE . howcalled .
sum = Arg(1) + Arg(2)
IF howcalled = 'COMMAND' THEN SAY 'The answer is' sum
   ELSE RETURN sum
```

The following program issues one of four commands to display information about a file on the system. There is one command for each of the four IBM System Application Architecture systems. The PARSE SOURCE instruction determines which system this program is running on, which indicates the appropriate command to issue. An argument is passed to the program giving the names of the objects or files to display.

```
PARSE SOURCE system .

ARG fileid                       /* Get the file name.            */
SELECT
   WHEN system = 'OS/400' Then 'DSPFD FILE('fileid')'
   WHEN system = 'CMS' Then 'LISTFILE' fileid '(LABEL)'
   WHEN system = 'OS/2' Then 'DIR' fileid
   WHEN system = 'TSO' Then 'LISTDS' fileid 'STATUS'
   OTHERWISE SAY 'Sorry, I don''t know the file-listing command on this system'
END
```

Parsing With PARSE VERSION

PARSE VERSION gets information about the particular version of REXX which you are running. This is particularly useful when writing programs which will run on different levels of REXX.

For example, if you write a REXX program which uses instructions or functions which only became available with version 3.48 of REXX, you might want to include the following lines in the program to make sure it can run on another system where you are not sure of the version:

```
PARSE VERSION . ver .
IF ver < 3.48 THEN
   DO
     SAY 'This program must run on REXX version 3.48 or greater'
     EXIT 99
   END
```

Using Parsing in a Program

The following is an example of a program using parsing to extract and enumerate the words in a user-supplied sentence. Notice how the PARSE PULL and PULL instructions get user input and how PARSE VAR is called within a loop to break the input string into its component words.

```
/* Creates an array from the words in a sentence.        */
/* Ask the user to type a sentence.                      */
SAY "Type a sentence:"
PARSE PULL sentence
/* For example, if the user types "This is a test",      */
/* that string is assigned to the variable SENTENCE.     */

/* Parse the input into an array of compound variables.  */
/* With each iteration of the loop, a word is removed    */
/* from the string in the variable SENTENCE and assigned */
/* to the next variable in the array.                    */
DO count = 1 UNTIL sentence = ''
  PARSE VAR sentence word.count sentence
  END


/* Now the variables (as set by the PARSE VAR loop) are: */
/* count = 4                                             */
/* word.1 = "This"                                       */
/* word.2 = "is"                                         */
/* word.3 = "a"                                          */
/* word.4 = "test"                                       */
/* sentence = ""                                         */

/* Display words by number.                              */
DO FOREVER
   number = ""
   SAY "Enter a number:"
   DO WHILE DATATYPE(number,n) = 0  /* Stay in this loop */
     PULL number                    /* until a number is */
     END                            /* typed.            */
   IF number <= 0 THEN LEAVE
   IF number > count
     THEN DO
             SAY "There were only" count "words...",
                 "Try again or type 0 (zero) to quit."
             ITERATE
             END
      ELSE DO
             SAY "Word" number "is '"||word.number||"'.",
                 "Try again or type 0 (zero) to quit."
             END
   END
EXIT
```

Parsing With Patterns

Strings can be parsed using patterns instead of words. By using patterns in the parsing template, you can denote other characters or strings as delimiters or select portions of the input string by specifying character positions.

There are three types of patterns which can be used with parsing:

- Literal
- Positional
- Variable.

Using Literal Patterns

To specify delimiting characters other than spaces in a parsing template, put the character in quotation marks.

For example, to parse a string of four data items separated by a slash character (/), put the slash in quotation marks:

```
/* Parsing with literal pattern.  Assume that the          */
/* string passed to this program is RED/DARK BLUE/GREEN/YELLOW.  */

PARSE ARG data1 "/"  data2 "/" data3 "/" data4 "/" .

/* The variables are set to:                               */
/* data1 = "RED"                                           */
/* data2 = "DARK BLUE"                                     */
/* data3 = "GREEN"                                         */
/* data4 = "YELLOW"                                        */
```

The patterns, denoted by the slash characters, are removed from the parsed data. The fourth slash, having no match in the input string, is ignored (as was the placeholder period). Where REXX finds no match for a literal pattern, the end of the string is assumed. Note how this rule applies to a slightly different argument:

```
/* Same ARG instruction, same template; this time the       */
/* string passed to this program is RED/DARK BLUE/GREEN,YELLOW.  */

PARSE ARG data1 "/"  data2 "/" data3 "/" data4 "/" .

/* The variables are set to:                               */
/* data1 = "RED"                                           */
/* data2 = "DARK BLUE"                                     */
/* data3 = "GREEN,YELLOW"                                  */
/* data4 = ""                                              */
```

If no match for the third slash is found, REXX assigns the remainder of the string to the preceding variable data3 and sets data4 to null.

A literal pattern can have more than one character. As with a single character, the pattern is removed from the parsed data if it is found. If it is not found, all succeeding variables are set to null.

```
/* Using a literal parsing pattern with more than one character.  */
input = "This string has many words in it."
PARSE VAR input beginning "has many words" ending
/* Now, the variables are set as follows:                         */
/* beginning = "This string"                                      */
/* ending = "in it."                                              */
```

This brings us back to the general rule about parsing templates. Where no literal pattern is provided, one variable immediately follows another. The delimiter is assumed to be one or more blanks, which are removed.
The string is separated into words:

```
/* Using a literal parsing pattern with more than one character.  */
input = "This string has many words in it."
PARSE VAR input word1 word2 "has many words" word3 word4
/* Now, the variables are set as follows:                         */
/* word1 = "This"                                                 */
/* word2 = "string"                                               */
/* word1 = "in"                                                   */
/* word2 = "it."                                                  */
```

To parse by one and only one space, and thereby preserve leading and trailing spaces, use a quoted blank (" ") as a literal pattern.

Here is an example of how to parse the argument string when REXX is called as the CPP of a CDO. The definition for the CDO is as follows:

```
CMD          PROMPT('TESTS REXX AS A CDO')
PARM         KWD(LIB) TYPE(*CHAR) LEN(10) +
               PROMPT('LIBRARY NAME')
PARM         KWD(FILE) TYPE(*CHAR) LEN(10) +
               PROMPT('FILE NAME')
PARM         KWD(MBR) TYPE(*CHAR) LEN(10) +
               PROMPT('MEMBER NAME')
```

When REXX is called as a result of this command, the argument string will look like the following, although the values within the parentheses will match the entries you specify.

LIB(*libname*) FILE(*filename*) MBR(*mbrname*)

You can parse these parameters as shown in the following example:

```
/*  Using PARSE ARG to parse arguments passed by CDO. */
PARSE ARG  "LIB("libname")" "FILE("filename")" "MBR("mbrname")" .
SAY 'LIBRARY NAME PASSED WAS' libname
SAY 'FILE NAME PASSED WAS' filename
SAY 'MEMBER NAME PASSED WAS' mbrname
RETURN
```

Using Positional Patterns

As its name implies, a positional pattern uses character position to break up a string. Positional patterns are especially useful in applications where the format, rather than the content, is known. They offer precise selection of text, regardless of the kind of delimiters, or the absence of delimiters altogether. Any number in a template is presumed to refer to a character position.

- If the number is unsigned (without + or - signs), then it refers to an absolute position in the input string. The number specifies the position where the next segment of parsed text begins.

- If the number is preceded by a plus or minus sign, then it refers to a relative position or offset from the current character position. The current position is determined as the template runs from left to right.

Parsing by positional pattern is especially useful for selecting data from a string with fixed format, as in a data file. Here is an example:

```
/* Parsing an address record by character position.            */
/* Assume an address file with the following field lengths:    */
/* last name 15                                                 */
/* first name 10                                                */
/* street address 14                                           */
/* city 10                                                      */
/* state 2                                                      */
/* zip code 5                                                   */
/*                                                              */
/* Thus, the variable ADDRESS might contain the string:        */
/* "SMITH          JOHN      123 MELODY LN. NEW FALLS NY   11919"    */

PARSE VAR address lnname 16 fname 26 street 40 city 50 state 52 zip

/* The variables are set as:                                   */
/* lname = 'SMITH          '                                   */
/* fname = 'JOHN      '                                         */
/* street = '123 MELODY LN.'                                    */
/* city = 'NEW FALLS '                                          */
/* state = 'NY'                                                 */
/* zip = '11919'                                                */
```

A pattern that is specified using relative positions can move backward and forward through the string from any given point. In this way, a given delimiter might be located by a literal pattern and a preceding text segment picked out.

```
salesrec = "SMITHHK012345USDSAMWOBANK      "
PARSE VAR salesrec "USD" -8 loc +2  rev "USD" custtlx
          /* loc = 'HK'                */
          /* rev = '012345'            */
          /* custtlx = 'SAMWOBANK      ' */
```

Using Variables in Patterns

Variable patterns may be used as another way of parsing data or assigning variables.

Suppose you are reading a file which has variable length data fields in it, containing a name and an address. Each record starts with two numbers which give the column numbers that the name and address data start in. You could read the column numbers and break out the name and address in one instruction.

```
dataline = '10 30    Mary Ellen Friedmann123Main Street'
PARSE VAR dataline,
   namecol addrcol =(namecol) name =(addrcol) address
```

If the file instead had the column number and the width of the name, with the address immediately following the name, the data could still be broken out in one instruction.

```
dataline = '10 20    Mary Ellen Friedmann123Main Street'
PARSE VAR dataline,
   namecol namewidth =(namecol) name +(namewidth) address
```

Literal patterns also may be used as variables. Suppose a text-processing program has to read a character string that begins with an arbitrary delimiter character, and is divided into two parts by a second occurrence of that delimiter character. The following is an example:

```
string ='/First part/Second part'
PARSE VAR string delim +1 part1 (delim) part2
```

This program will set `part1` to `First part` and `part2` to `Second part`. `Delim` will be set to `/`.

Using String Functions

One group of the built-in functions provided by REXX are string functions. These functions allow you to obtain information about a string or to process text in other ways. REXX built-in functions are also discussed in "Using REXX Built-in Functions" on page 104.

For the complete syntax of each of the functions described here, see the *REXX/400 Reference*.

Managing Strings

Several of the REXX string functions manage strings by breaking up or changing the input strings. These functions manage substrings, make changes to strings, or format strings.

Using the Substring Functions

These functions return a piece of a larger string:

SUBSTR: The SUBSTRING function gets a piece of a string by numbered position.

```
SUBSTR("I Love my AS/400 system",3,4)    /* This returns "Love".  */
```

LEFT: The LEFT function gets the leftmost substring and can add trailing spaces.

```
LEFT("I Love my AS/400 system",6)         /* This returns "I Love".*/
```

RIGHT: The RIGHT function gets the rightmost substring and can add leading spaces.

```
RIGHT("I Love my AS/400 system",6)        /* This returns "system".*/
```

WORD: The WORD function gets a word from a string (by number).

```
WORD("I Love my AS/400 system",2)           /* This returns "Love".  */
```

SUBWORD: The SUBWORD function gets a substring beginning with a given word.

```
SUBWORD("I Love my AS/400 system",2,1)    /* This returns "Love".  */
```

Using the String Editing Functions
These functions change a string:

INSERT: The INSERT function inserts a substring into a string.

```
INSERT("Wonderful ","I Love my AS/400 system",10)
/* This returns "I Love my Wonderful AS/400 system". */
```

OVERLAY: The OVERLAY function overlays part of one string with another.

```
OVERLAY("LOVE","I like my AS/400 system",3)
                                /* This returns "I LOVE my AS/400 system".*/
```

REVERSE: The REVERSE function exchanges the characters in a string, end for end.

```
REVERSE("I Love my AS/400 system")   /*This returns "metsys 004/SA ym evoL I".*/
```

COPIES: The COPIES function replicates a string a given number of times.

```
COPIES("Love",3)                         /* This returns "LoveLoveLove".*/
```

DELSTR: The DELSTR function deletes a substring from the input string.

```
DELSTR("I Love my Wonderful AS/400 system",11,10)
                     /* This returns "I Love my AS/400 system".*/
```

DELWORD: The DELWORD function performs the same function as DELSTR, but begins with a given word.

```
DELWORD("I Love my Wonderful AS/400 system",4,1)
                     /* This returns "I Love my AS/400 system".*/
```

Using the String Formatting Functions
These functions change a string by adding or removing spaces or other characters. The LEFT and RIGHT functions can also add spaces.

SPACE: The SPACE function adds or deletes intervening spaces (or other delimiting characters) between words.

```
SPACE("I Love my AS/400 system",2)
                     /* This returns "I  Love  my  AS/400  system".*/
```

CENTER: The CENTER function centers the input string within a large string of a given length, and adds spaces or other characters.

```
CENTER("I Love my AS/400 system",30)
                    /* This returns "   I Love my AS/400 system   ".*/
```

The CENTRE function can also be used. It is exactly the same as the CENTER function.

STRIP: The STRIP function removes leading or trailing spaces, or both from a string.

```
STRIP("   I Love my AS/400 system   ")
                        /* This returns "I Love my AS/400 system".*/
```

Measuring Strings

Several built-in functions are available to give information about a string. They can determine the length of a string, compare strings, or locate particular positions within a string.

Using the LENGTH, WORDS, and WORDLENGTH Functions

Each of these functions returns a number which is the count of characters or words in a string.

LENGTH: The LENGTH function counts the characters in a string.

```
LENGTH("I Love my AS/400 system")        /* This returns 23.*/
```

WORDS: The WORDS function counts the words in a string.

```
WORDS("I Love my AS/400 system")         /* This returns 5. */
```

WORDLENGTH: The WORDLENGTH function returns the length of a word, specified by number.

```
WORDLENGTH("I Love my AS/400 system",2)   /* This returns 4. */
```

Using the VERIFY, ABBREV, and COMPARE Functions

Each of these functions returns a number which results from a comparison of two strings.

VERIFY: The VERIFY function determines whether one string is made up of characters in another string. It can return the position of either the first matching character or first non-matching character, which is the default.

```
VERIFY("I Love my AS/400 system"," ABCDEFGHIJKLMNOPQRSTUVWXYZ")
                                    /* This returns 4.*/
```

ABBREV: The ABBREV function returns 1 if one string matches the leading characters of another.

```
ABBREV("Love","Lo")  /* This returns 1 (true). */
```

COMPARE: The COMPARE function tells if two strings are identical.

```
COMPARE("I Love my AS/400 system","I Love my AS/400 system")
                /* This returns 0 for exact match.             */
COMPARE("I Love my AS/400 system","I Like my AS/400 system")
                /* This returns 4 for position where matching ended.*/
```

Using the POS, LASTPOS, WORDINDEX, and WORDPOS Functions
Each of these functions returns a number which indicates the position of the character for which you are looking.

POS: The POS function searches one string from the beginning looking for a given substring and returns the position of that substring.

```
POS("Love", "I Love my AS/400 system")    /* This returns 3. */

POS("like","I Love my AS/400 system")  /* This returns 0 since a    */
                                       /* given string was not found. */
```

If a given string is not found, a zero is returned

LASTPOS: The LASTPOS function searches the same as POS except from the end, backward.

```
LASTPOS(" ","I Love my AS/400 system")    /* This returns 17.*/
```

WORDINDEX: The WORDINDEX function finds a word by number and returns the initial character position.

```
WORDINDEX("I Love my AS/400 system",2)    /* This returns 3. */
```

WORDPOS: The WORDPOS function finds a word by searching for the word itself and returns its initial character position.

```
WORDPOS("Love","I Love my AS/400 system")  /* This returns 2. */
```

Using REXX Programs as String Functions
A REXX program may be called as a string function, as shown in the examples which follow.

Example 1: This example turns an input number into a string in currency format. It uses PARSE VAR with a literal pattern and PARSE VALUE with character position pattern.

```
/* This function takes a number and returns a string         */
/* in comma-delimited dollar format.                         */
/* For example, DOLLAR(1234.5555) returns '$1,234.56'.       */

ARG number                          /* Get the argument NUMBER. */

/* Round off the argument to the nearest cent. Then           */
/* parse the result into the integer (DOLLARS), the          */
/* decimal point, and the decimal fraction (CENTS).          */

PARSE VALUE format(number,,2,0) WITH dollars "." cents

dollars = ABS(dollars)              /* Make DOLLARS positive.    */

backin = REVERSE(dollars)           /* Reverse the digits in     */
                                    /* DOLLARS so you can parse  */
                                    /* them into groups of 3     */
                                    /* (see REVERSE(), above).   */

backout = ""                        /* Initialize a variable     */
                                    /* for re-concatenation.     */

DO WHILE LENGTH(backin) > 3         /* While three digits or     */
                                    /* more remain in BACKIN,    */
   PARSE VAR backin group 4 backin  /* take each group of three  */
                                    /* remaining digits, and     */
   backout = backout||group||","    /* then join it to the end   */
                                    /* of the BACKOUT variable   */
                                    /* and add a comma.          */

   END

backout = backout||backin||"$"      /* Concatenate the digits    */
                                    /* that remain; add '$'.     */

IF number < 0 THEN                  /* If the argument was       */
   backout = backout||"-"           /* negative, restore the     */
                                    /* minus sign.               */

bucks = REVERSE(backout)||"."||cents  /* Restore the proper order */
                                    /* of the digits; add the    */
                                    /* decimal point and cents.  */

RETURN dollars                      /* Return the string.        */
```

Example 2: This example shows how the DOLLAR function can be used in a program.

```
/* Using the DOLLAR function                                   */
total = 0
DO FOREVER
   SAY "Enter amount:"
   PULL entry
   IF ¬DATATYPE(entry,n)            /* If entry is not a valid number,*/
   THEN LEAVE                       /* leave the loop.               */
   total = total + entry
   SAY "Total = " DOLLAR(total)     /* Display total in dollar format.*/
END
SAY entry "is not a number.  Returning to SYSTEM."
```

Example 3: Here is a useful search-and-replace string function. Notice how the second PARSE instruction uses a variable as a pattern.

```
/* Function:  CHANGE(string,old,new)                 */
/*                                                   */
/*                                                   */
/* Changes all occurrences of "old" in "string"      */
/* to "new".  If "old" == "", then "new" is prefixed  */
/* to "string".                                      */

PARSE ARG string, old, new
IF old=="" THEN RETURN new||string

out=""
DO WHILE POS(old,string) ¬= 0
   PARSE VAR string prepart (old) string
   out=out||prepart||new
END
RETURN out||string
```

Example 4: This example shows how to call the CHANGE function in a program.

```
/* This is an example using the CHANGE function. */
direction = "north by northwest"
wrong = "north"
right = "south"
SAY direction
SAY change(direction,wrong,right)        /* This says "south by southwest".*/
```

Chapter 7. Understanding Commands and Command Environments

Commands are a special type of clause, which are passed to another program to run. In order to work with commands, this chapter covers the following topics:

- Understanding Commands
- Understanding Command Environments
- Understanding the Error and Failure Conditions.

Understanding Commands

A command is an expression which is passed to a command environment to run. The default command environment is CL. You can specify an environment which you define, by using the command interface discussed in the *REXX/400 Reference*.

Understanding Clause Interpretation

Single clauses consisting of just an expression are instructions known as commands. The expression is evaluated and the result is passed as a command string to an environment external to REXX. If the external environment is the CL command environment, which is the default, then information about pseudo-CL variables occurring in this result is also passed. For information on pseudo-CL variables, see "Understanding Pseudo-CL Variables" on page 88.

REXX interprets a clause as a command if it is not a comment, keyword instruction, variable assignment, or label. Any line that starts with a literal string, a string within quotation marks, will be treated as a command. A line that just contains a variable, a function call, or a more complicated expression will also be treated as a command. For example, the following line in a REXX program will issue a command because it is just a line containing a REXX variable:

```
STRSEU
```

This would presumably run the Start Source Entry Utility (STRSEU) command, although the actual result could be different because:

- STRSEU is a variable and could be set to some other value.

- REXX may be sending commands to an environment other than CL.

It is a good practice to put commands within quotation marks, except for those parts where you are using variable values. This will give you a slight performance improvement, because the number of variables REXX must find is reduced. Putting your command names in quotation marks will also prevent confusion between REXX instructions and commands with the same names as REXX instructions. In particular, REXX has a CALL instruction and CL has a CALL command. When you want to use the REXX CALL instruction, the format is:

►►—CALL————————►◄
          └─*parameters*─┘

and when you want to specify the CALL command, the format is:

►►—"CALL"————————►◄
          └─*paramters*─┘

Understanding Command Environments

Associated with running every REXX program is a command environment. When the REXX interpreter finds a command within the REXX program, control will be given to the command environment which in turn runs the command and then returns control to REXX to continue running the program.

There are two types of command environments available to REXX:

1. System-defined command environments. These include:

   * Control language (CL) command environment, called COMMAND. The command environment lets you issue CL commands, and will be the only called environment used by most REXX programs.

   * Common Programming Interface (CPI) Communications environment, called CPICOMM. The CPICOMM environment, which is the communications element of the SAA Common Programming Interface (CPI), lets you issue CPI-Communications commands. For more information on CPI Communications, see the *SAA Common Programming Interface Communications Reference*.

   * Structured Query Language (SQL) environment, called EXECSQL. The EXECSQL environment lets you use SQL, which is the standard database interface language used by DB2/400. For more information on SQL statements, see the *DB2 for AS/400 SQL Reference*. For more information on the EXECQSL environment, see the *DB2 for AS/400 SQL Programming*.

     Note: To use the EXECSQL environment on a system which does not have the DB2/400 Query Management and SQL Development Kit Version 3 LPP, 5716-ST1, installed, see the *DB2 for AS/400 SQL Programming* for special instructions for handling the REXX program.

2. User-defined command environments. These are application programs which are written to handle commands issued by REXX programs. You identify these in library/object format. For example, 'MYLIB/FRED' would refer to program FRED in library MYLIB. The default for library is *LIBL.

   Programmers can create special names for their command environments by using exit programs and the REXX system exit interface to handle commands issued in REXX programs. In this case, the name of the environment does not have to be a program name. It will be passed to the exit programs as two ten-character strings. A slash character in the name divides it into the two strings. The maximum length for the name of the environment is 21 characters. For more information about system exits, see the *REXX/400 Reference*.

REXX programs start out with an initial command environment, specified when the REXX interpreter is started. If the REXX program was run by using the Start REXX Procedure (STRREXPRC) command, the initial command environment can be explicitly set with the CMDENV parameter or be allowed to default to COMMAND. The command environment may be changed within the REXX program by using the ADDRESS instruction. The command environment can be checked with the ADDRESS built-in function.

Using the ADDRESS Instruction

The ADDRESS instruction changes the command environment from within a REXX program or send a single command to a specified command environment.

The ADDRESS instruction takes the form:

```
►►──ADDRESS──environment─────────────►◄
                        └─expression─┘
```

Where

environment   is the destination of the string

expression   is evaluated and the string passed to the environment.

For example, you could run the Display Library List (DSPLIBL) command as follows:

```
ADDRESS COMMAND "DSPLIBL"
```

Using the ADDRESS instruction with only the name of an environment makes a lasting change in the destination of commands.

```
ADDRESS 'MYLIB/FRED'
SAY 'Now commands issued in this program go to the program MYLIB/FRED'

'DO WHAT I MEAN'          /* DO WHAT I MEAN is passed to MYLIB/FRED    */
                          /* as a command.                            */

ADDRESS COMMAND           /* This sets the command destination back to */
                          /* the default command environment (CL).    */

/* When the ADDRESS instruction is used with the name of a user      */
/* command program, and a command given, one command is sent        */
/* to the command environment but it does not make a lasting change  */
/* to the destination of commands.                                   */

ADDRESS 'MYLIB/FRED' 'DO WHAT I MEAN'
                          /* DO WHAT I MEAN is passed to MYLIB/FRED    */
                          /* as a command.                            */

'ANOTHER COMMAND FOR FRED'
/* This will get a return code of CPF0001, because "ANOTHER" is not  */
/* a CL command, and CL is the destination of commands because of    */
/* an ADDRESS COMMAND.                                               */
```

Understanding Messages

Communication between programs occurs through messages. CL commands and commands to user-defined command environments may send messages to report their status or indicate errors.

For more information about the types of messages and when they are used, see the *CL Programming*.

Messages issued by commands may set return codes and raise conditions in your REXX programs. The following section discusses how your REXX programs can find out when messages are issued by commands, and how the REXX programs can handle the messages.

Understanding Return Codes

Any command environment must send a return code back to REXX following the completion of a command. The return code provides an indication of whether the command was run successfully or not. REXX makes the return code available to the REXX program in the special REXX variable RC. By convention, a zero return code means that the command was run successfully while a nonzero return code means an error or some abnormal condition occurred. REXX places no restriction on what a nonzero return code can be other than limiting the value to 500 bytes in length. Any nonzero value, whether numeric or character, can be used. REXX also does not interpret the meaning of a nonzero return code. The meaning of such a return code is determined strictly by the command environment that returned the value.

Understanding Return Codes From the CL Command Environment

If an error is found in a CL command while it is being run, an escape message will be sent back to the user of the command. REXX automatically monitors for all escape messages. Should an escape message be sent from the CL command environment, the return code will be set to the message ID that is associated with that message. Thus, the special REXX variable RC will be set to the message ID. Moreover, the escape message will be on the message queue that is associated with REXX. The REXX program may obtain this message plus any accompanying diagnostic messages by using the Receive Message (RCVMSG) command. To receive messages that were sent by the CL command environment, the message queue should be specified by using the *SAME * value for the PGMQ parameter of the RCVMSG command. If no escape message is sent from the CL command environment, the special variable RC will be set to a zero return code.

The following is a simple example of checking for a zero return code from a CL command:

```
"SNDMSG MSG('HELLO') TOUSR(AMIR)"
IF RC = 0 THEN SAY 'The SNDMSG command ran correctly.'
   ELSE SAY 'The SNDMSG command did not run correctly.'
```

A CL command can also cause the CL command environment to send a notify or status message. Such messages are not automatically treated as error conditions. The default action taken on messages of these two types is to ignore them. Unless an escape message is later sent, the special variable RC will be set to a zero return code, and the message will not be left on the message queue.

However, the REXX program can control how messages of these types are to be treated. The SETMSGRC built-in function can cause some or all notify and status messages to be treated the same as escape messages. Any notify or status message that is identified through the use of the built-in function will cause the special variable RC to be set to the associated message ID. Moreover, the notify or status message will be on the message queue associated with REXX. From there it can be received in the same way an escape message can be.

If the SETMSGRC function is used, and a status or notify message is received, the program which sent the status or notify message ends and REXX raises the ERROR condition.

It is important to note that if the special variable RC is zero after a CL command is run, there is no message on the message queue that can be received. If RC is nonzero, there is at least one message that can be received. Thus, before an attempt is made to receive a message, RC should be tested to see if it contains a nonzero value.

For some messages, you can decide what to do by just looking at the message number in RC. The following program tests the return code and decides what error has occurred:

```
libname = 'MYLIB'
DO UNTIL RC = 0
  'CHGCURLIB CURLIB('libname')'

   SELECT
     WHEN RC = 0 THEN NOP

     WHEN RC = 'CPF2110' THEN DO
        SAY 'Sorry, Library' LIBNAME 'not found.'
        SAY 'Enter a new library name.'
        PULL libname
        END

     WHEN RC = 'CPF2176' THEN DO
        SAY 'Library' LIBNAME 'is damaged.'
        SAY 'This program is stopping now.'
        EXIT
        END

     OTHERWISE
        SAY 'The CHGCURLIB command set Return Code' RC
        SAY 'This program is stopping now.'
        EXIT

   END
END
```

Understanding Return Codes from the CPICOMM Command Environment

The REXX variable RC indicates either a successful call to the CPI-Communications interface, or failure to call to the CPI-Communications interface. If the call to the CPI-Communication interface completed successfully, then the REXX RC variable will be set to zero. If the call to the CPI-Communications interface was not successful, then the REXX RC variable will be set to a negative number and the failure condition raised.

The following table lists the possible negative values for the REXX RC variable.

Negative Value	Brief Description
(-3)	CPICOMM command does not exist or was spelled incorrectly
(-9)	Out of Memory Failure
(-10)	Too many parameters specified on the CPICOMM command
(-11)	Too few parameters specified on the CPICOMM command
(-14)	Internal system error occurred in the CPICOMM environment
(-24)	Unable to fetch value for REXX variable
(-25)	Unable to set value to REXX variable
(-28)	Variable name contains restricted character(s)

Only when the REXX RC variable is set to zero have the parameters to the CPICOMM command environments been updated with valid values.

Understanding Return Codes from the EXECSQL Command Environment

The REXX variable RC indicates either a successful call to the SQL interface or a failure to call the SQL interface. If the call completes successfully, the REXX RC variable is set to zero. Any warnings will be shown with a positive RC variable. If the call cannot be completed, the REXX RC variable is set to a negative number and the failure condition is raised. For more information on the EXECSQL environment, see the *DB2 for AS/400 SQL Programming*.

Note: To use the EXECSQL environment on a system which does not have the DB2/400 Query Management and SQL Development Kit Version 3 LPP, 5716-ST1, installed, see the *DB2 for AS/400 SQL Programming* for special instructions for handling the REXX program.

The following table lists the possible negative values for the REXX RC variable.

Negative Value	Brief Description
(+10)	An SQL warning resulted, signaled by a + SQLCODE or a non-blank entry in SQLWARN.
(0)	Successful execution
(-10)	An SQL error resulted from execution of an SQL statement
(-100)	A REXX interface error resulted. The SQLCA information is invalid.

For a REXX RC value of +10 or -10, you may analyze the problem further by looking at the generated SQLCA variables. For more information on values returned in the REXX RC variable for the EXECSQL environment, the SQL communications area (SQLCA), and the SQL description area (SQLDA), refer to the *DB2 for AS/400 SQL Reference*.

Understanding Return Codes from User-Defined Command Environments

A user-defined command environment must return a return code, just like the CL command environment does. However, the user-defined command environment has two methods by which it can accomplish this task:

1. The defined interface between REXX and the command environment lets a program directly return a value. The value that is returned is placed in the special variable RC. In this case, because the value was returned directly, there is no message on the message queue that the REXX program can receive.

2. The program for the user-defined command environment can send an escape message, or a status or notify message identified by SETMSGRC, and not return a value directly. In this case, the message ID for the message is placed into the variable RC. The message is left on the message queue and can be received.

When a user-defined command environment is being used, the REXX program should not attempt to receive a message unless the command environment sends messages. A nonzero return code may result, without a message on the queue to receive. Messages should be received only when the return code is nonzero, and the current command environment is one that sends messages.

Understanding the Error and Failure Conditions

A REXX program cannot monitor for exceptions that are sent from a command environment. Rather, REXX automatically monitors for such exceptions and then informs the REXX program that an exception has occurred. The special variable RC is one way that the REXX program is informed. A second way is through the use of conditions.

All exceptions that can be received from a command environment are divided into two REXX conditions: ERROR and FAILURE. Usually, the ERROR or FAILURE condition will occur in response to action taken by the command environment. However, the following are two cases where REXX itself will raise a FAILURE condition:

- The program that is to be called for a command environment cannot be found by REXX.

- The program for a command environment cannot be used because the user of REXX is not authorized to that program.

Conditions and condition traps are discussed further in Chapter 11, "Understanding Condition Trapping" on page 131.

Understanding CL Command Environment Conditions

An exception from the CL command environment can cause either the ERROR or FAILURE condition to occur. If the exception was CPF0001 or CPF9999, the FAILURE condition will occur. Any other exception will cause the ERROR condition to occur.

If the SETMSGRC built-in function was used, any notify or status messages that were identified when the function was used will also cause the ERROR condition to occur.

Understanding CPICOMM and EXECSQL Command Environment Conditions

An exception from the CPICOMM command environment will cause a failure condition to occur. A negative value will be placed in the REXX RC variable indicating the type of exception. For additional details, see "Understanding Return Codes from the CPICOMM Command Environment" on page 83.

An exception from the EXECSQL command environment will cause a failure condition to occur. A negative value will be placed in the REXX RC variable and details will be available in the SQLCA variables. For additional details, see "Understanding Return Codes from the EXECSQL Command Environment" on page 84.

Understanding User-Defined Command Environment Conditions

A user-defined command environment has two methods that can be used to cause the ERROR or FAILURE condition to occur in the REXX program:

- The command environment can indicate directly through the defined interface with REXX that either the ERROR or FAILURE condition is to occur.

- The command environment can send an escape message. This will cause the FAILURE condition to occur. If the SETMSGRC built-in function was used, any notify or status messages that were identified when the function was used will also cause the FAILURE condition to occur.

Understanding the Control Language (CL) Command Environment

The control language (CL) command environment must be used whenever a REXX program contains CL commands. This command environment is the default environment. It can also be specified directly as a command environment on the Start REXX Procedure (STRREXPRC) command, the QREXX interface, and on the ADDRESS instruction.

Before using a CL command in your REXX program, you should make sure that it is allowed. You can confirm this in one of two ways:

1. You can check the syntax diagram for a CL command in the *CL Reference.* This will show the *IREXX value for the ALLOW parameter if the command can be used in an interactive REXX program or a *BREXX value for the ALLOW parameter if the command can be used in a batch REXX program. Either or both values can be specified, along with other values for the ALLOW parameter.

2. For a CL command that you create using the Create Command (CRTCMD) command, the ALLOW parameter must specify the *IREXX or *BREXX values in order for the command to be used within the REXX program.

When a CL command is used in a REXX program, it will have the same effect as when the command is used in a CL program. However, because REXX programs operate differently than CL programs, two areas require special attention:

- CL commands that return values

- CL commands that are sensitive to program boundaries.

Understanding CL Command Parameters

When a CL command is run from within a REXX program, values for the various parameters on the command must be specified or default values, as determined by the command, will be used. Within the REXX program, there are four general ways that parameter values can be specified:

1. As constant values

2. As quoted constant values

3. By assigning a value to a REXX variable and then using that variable within an expression thereby allowing REXX to build the final command using the content of the variable

4. By assigning the value to a pseudo-CL variable and using that variable within the command thereby effectively passing the variable rather than just the value of the variable.

A constant value is a value that is given directly within the command as a constant. Moreover, the value is not enclosed within single quotation marks. A numeric value can be specified this way — either as a standard REXX number or as a number in REXX exponential notation. A character value can also be specified this way. However, with a character value, two transformations are performed before the value is used by the command:

1. Any leading and trailing blank are removed

2. The value is then folded to uppercase

A quoted constant is also a value that is given directly within the command. In this case, however, the value is enclosed between single quotation marks. The value is always taken to represent a character value. The value will be taken as everything that is enclosed between the quotation marks, including any leading and trailing blanks. The quotation marks are not considered part of the value and are removed before the command receives the value. In addition, no folding will be done on the value.

A REXX variable can be used in place of either a constant or a quoted constant. The value to be used is first assigned to the variable. If the value is to be a quoted constant then the enclosing single quotation marks must be part of the value that is assigned. The REXX variable is then used within an expression. When REXX evaluates the expression, the value of the REXX variable is placed into the resulting command at the specified point. A REXX variable used in this way can thus be used to pass a value to the command.

A pseudo-CL variable is a REXX variable whose name is specially identified within an expression that is to be evaluated to a command. In this case, the REXX variable is effectively passed to the command and not just the value it may contain. Pseudo-CL variables are used for those command parameters that can return a value. That is, the pseudo-CL variable can provide an input value to the command parameter and can receive a value from the command for the same parameter. This is different than a usual REXX variable which can be used only to provide input to a parameter.

To use a pseudo-CL variable, the input value for the parameter is assigned to the variable. The input value can be a numeric value or a character value. However, if it is a character value, it will be always processed as a quoted constant. There is no need to include enclosing quotation marks for a value in this case. This is in contrast to a usual REXX variable. If enclosing single quotations marks are included, they will be considered as part of the value and will be received by the command. More information on pseudo-CL variables is provided in the following section.

The input value for any command parameter is checked to ensure that it is of the correct data type and length for that parameter. For example, a parameter that is of type decimal can only receive a value that is a REXX number in standard notation. A parameter that is of type character can only receive a character value. This check is done regardless of whether the command actually uses that value input or not. This fact is important in the use of pseudo-CL variables which would be used for parameters that only return a value. In these cases, the pseudo-CL variable must nonetheless be initialized with an input value. The input value must be compatible with the definition of the parameter.

Any value that is returned from a command will be in the correct form when it is placed into the pseudo-CL variable. If, for example, the command returns a numeric value, the value will be converted to a REXX number and that number will be stored in the variable that is identified. Normally, the complete value that is returned will be placed into the pseudo-CL variable. However, there is one special case where this may not be true. If the CL command parameter returns a varying length value, that is, the parameter was created with type *CHAR, and the *RTNVAL and *VARY attributes, the length of the pseudo-CL variable becomes significant. The value returned from the command will be no longer than the value in the pseudo-CL variable at the time the command is run. This case can only occur with character data. To protect against having the returned value truncated, the pseudo-CL variable should be initialized with a value of adequate length. CL commands which retrieve or receive information, such as the Receive Message (RCVMSG) command, are examples of the types of commands where data might be truncated. For more information, see the *CL Reference*.

Understanding Pseudo-CL Variables: A pseudo-CL variable is a REXX variable that is specified with a special syntax within a CL command. When a REXX variable is specified with this syntax, the variable will provide the same function as a CL variable does in a CL program. The variable will be able to provide an input value to a command parameter and will be able to receive a return value from the command for that parameter.

To use a pseudo-CL variable within a command, in place of the value for the corresponding parameter, the name of the variable is used. The variable name must be specified using the following rules:

- The variable name must conform to the rules of both REXX and CL. The characters #, @, !, $, and REXX extension characters cannot appear in the name.

- The variable name must be 10 characters or less in length.

- The variable name must be preceded by an ampersand (&).

The use of the ampersand is critical because it causes the variable to be used rather than only the name of the variable. For example, the following two

commands, both specified as literals and both having a reference to the same REXX variable name, are not the same and will produce different results.

```
"RTVDTAARA DTAARA(MYDATA) RTNVAR(RXVAR1)"
"RTVDTAARA DTAARA(MYDATA) RTNVAR(&RXVAR1)"
```

In the first literal, RXVAR1 will be taken as a character value. In this case, the command will not run correctly. In the second string, the ampersand identifies RXVAR1 as being the name of a REXX variable. In this case, the variable RXVAR1 will be available to receive the returned value, and the command will run properly.

Similar consideration must be given when the command is specified as a REXX expression. The following two expressions are not the same:

```
'RTVDTAARA DTAARA('RXVAR2') RTNVAR('RXVAR1')'
'RTVDTAARA DTAARA('RXVAR2') RTNVAR(&RXVAR1)'
```

Assume that the variables RXVAR1 and RXVAR2 have been assigned values as follows:

```
RXVAR1='Data that should not be used'
RXVAR2='MYDATA'
```

Because the command is specified as an expression, REXX will evaluate the expression, and the result will be the command that is passed to the command environment. For the first expression, the result of the evaluation will be as follows:

```
"RTVDTAARA DTAARA(MYDATA) RTNVAR(Data that should not be used)"
```

REXX substituted the values for RXVAR1 and RXVAR2 wherever the variable names appeared in the expression. Again, this command will fail because a REXX variable is not in the evaluated command for the RTNVAR parameter.

For the second expression, the evaluated result will be:

```
"RTVDTAARA DTAARA(MYDATA) RTNVAR(&RXVAR1)"
```

Notice that for RXVAR2, REXX has substituted the value for the variable name. For RXVAR1, the substitution is not made. Instead, the variable name remains in the command. The contents of the data area MYDATA will be returned into RXVAR1. Note also in this example, the RTVDTAARA command is one that returns a varying length character string. The RTVDTAARA command is an example of a command that returns a varying length value without truncation, that is, the length of the pseudo-CL variable is not significant.

Example 1: This example uses pseudo-CL variables to work with CL commands. It also has a condition trap, identified as error_handler routine.

```
/* This program shows different ways to achieve the identical result. */
/* It adds libraries to the library list using the ADDLIBLE command.  */
/* In some cases, pseudo-CL variables are used, in others REXX        */
/* variables are concatenated with literal tokens.                    */

PARSE ARG lib1 lib2 lib3 lib4 rest
CALL ON ERROR NAME error_handler        /* Set up ERROR condition trap.*/
```

```
            'ADDLIBLE' lib1          /* Add a library to *LIBL,             */
                                     /* concatenate a literal, and a variable. */
            'ADDLIBLE  &lib2'        /* Add a library to *LIBL,             */
                                     /* using a pseudo-CL variable.         */
            'ADDLIBLE  LIB(&lib3)'   /* Add a library to *LIBL, in keyword  */
                                     /* format using a pseudo-CL variable.  */
            'ADDLIBLE  LIB('lib4')'  /* Add a library to *LIBL, in keyword  */
                                     /* format, concatenate a literal, and  */
                                     /* a variable.                         */
            EXIT

            /* error_handler routine */
            error_handler:


             IF RC ¬= CPF2103        /* Library is not already in the library, */
                                     /* then continue.                      */
                THEN DO
                  SAY 'ERROR:'rc 'detected on line number:'sigl'. The command is:'
                  SAY SOURCELINE(sigl)
             END
             RETURN
```

The following example shows a small program which issues CL commands, and
checks the return code on some and uses pseudo-CL variables on others.

```
/****************************************************************/
/* REXX procedure to determine if a given library is in the *LIBL. */
/*                                                              */
/*    Parameters :  Library name                                */
/*                                                              */
/*    Returns:      '1' if the library is found in the system portion*/
/*                      of the *LIBL.                           */
/*                  '2' if the library is found in the user portion  */
/*                      of the *LIBL.                           */
/*                  '3' if the library is not in the *LIBL.     */
/*                  '4' if the library does not exist.          */
/*                                                              */
/****************************************************************/


PARSE UPPER ARG lib

 /*  Check to see if the library exists.                        */
 /*  Pseudo-CL variable is not required for OBJ parameter since  */
 /*  it is not a RTNVAL(*YES) parameter.                        */
 'CHKOBJ OBJ(QSYS/'lib') OBJTYPE(*LIB)'

                            /* Check for object not found exception. */
 IF POS('CPF98',rc) ¬= 0 THEN DO
   "SNDPGMMSG MSG('Library: "lib "not found')"
   EXIT(4)
 END

      /* Retrieve the user and system library lists into REXX.  */
      /* Pseudo CL variables &urslibl and &syslibl used since both*/
      /* USRLIBL and SYSLIBL are RTNVAL(*YES) parameters.        */
'RTVJOBA    USRLIBL(&usrlibl) SYSLIBL(&syslibl)'

                                /* Is lib in the syslibl?        */
IF POS(lib,syslibl) ¬= 0 THEN
 EXIT('1')                              /* Exit: found in *SYSLIBL. */

ELSE IF POS(lib,usrlibl) ¬= 0 THEN
     EXIT('2')                          /* Exit: found in *USRLIBL. */
     ELSE
      EXIT('3')                      /* Exit: library not in *LIBL. */
```

Appendix F, "Sample REXX Programs for the AS/400 System" on page 157
contains two REXX programs which contain more examples of using CL commands
and pseudo-CL variables. These programs also use return codes and condition
traps to check on the success or failure or their commands. The first program
moves an object from a test library to a production library. The second one
displays the contents of a library.

Understanding CL Commands that are Sensitive to Program Boundaries

A number of CL commands are sensitive in some manner to program and
activation group boundaries. Examples of such commands are commands related
to messages and the various override file commands. The Send Program Message
command (SNDPGMMSG), for example, includes a parameter that specifies which

program is to receive the message, the program that is sending the message, or the program that called that program. The override file commands can scope to the program call level, activation group, or job. For a complete list of CL commands and APIs that are sensitive to boundaries, refer to the *ILE/C Concepts* book, SC41-4606.

Commands that are sensitive to program boundaries need special consideration when used in a REXX program. These considerations are required because of the way in which REXX programs are run. A REXX program is not a program object. Running a REXX program requires that the REXX interpreter be called, which then runs the REXX program interpretively. Therefore, the REXX interpreter is the program which determines how these CL commands will run.

The REXX interpreter does not call itself. To start another instance of the REXX interpreter you must use a CL command. Calling a REXX program by using the REXX CALL instruction will not cause a new instance of the REXX interpreter. Instead, the same instance of the interpreter will stop running the current REXX program and will then start running the new program. This is in contrast to the case where the STRREXPRC command is used within the current REXX program to start a new REXX program. In this case, a new instance of the interpreter will be started, and the new program will be run by that new instance.

All REXX programs that are called using the REXX CALL instruction or are used as a function within a REXX expression run under the same instance of the REXX interpreter. The REXX programs share the same message queue. Moreover, any override file command that is run from within any of these REXX programs will remain in effect until the REXX interpreter itself completes.

The importance of how the program is run is highlighted in the following example:

Assume that REXX program A calls REXX program B by using the REXX CALL instruction. Program B now wants to send its caller a program message by using the Send Program Message (SNDPGMMSG) command. In a CL program, the TOPGM parameter would be specified as *PRV *. This ensures that the message was sent to the caller of the CL program. In REXX, the message will be sent to the caller of the REXX interpreter and not REXX program A. To ensure that message is available to A, the TOPGM parameter must be specified as *SAME *.

Understanding the Call Program (CALL) Command

Within a REXX program, the CALL command can be used to call programs that are written in programming languages other than REXX. In addition to calling the identified program, this command permits parameters to be passed to the called program and lets the called program return values to the caller. The following rules apply to the parameter values that are passed:

- If the parameter value is a number in REXX notation and the program being called is not written in C/400, the REXX number is converted to a packed decimal (15,5) number. The called program will receive the packed number and not the original REXX number. Note that if the fractional part of the number contains more than five places after the decimal point, the fractional part will be truncated. If the integer part of the number contains more than 10 significant digits, an error will occur and the CALL will not run.

  Any numeric value for a parameter can be specified in one of three ways:

  1. By placing the numeric value directly in the command as a constant

2. By assigning the numeric value to a REXX variable and then using the variable within an expression which causes REXX to build the final command string using the REXX variable

3. By placing the numeric value within a pseudo-CL variable and then effectively passing that variable to the called program

- If the parameter value is a number in REXX notation and the program being called is written in ILE/C, the complete numeric value is passed directly, in REXX notation. A null character will be placed immediately after the last digit of the numeric value.

- If the parameter being passed is a number in REXX exponential notation and the program being called is not an ILE/C program, the value will be converted to a double precision floating point value. The program being called will receive this floating point value.

- If the parameter being passed is a number in REXX exponential notation and the program being called is an ILE/C program, the complete numeric value will be passed directly in REXX exponential notation. A null character will be placed immediately after the last digit of the value.

- If the parameter value being passed is a character value and that value is specified as a constant, the following will occur:

  – Any leading and trailing blanks in the value will be removed.

  – The value will then be folded to uppercase using the CL rules for folding. If the value cannot be folded, an error will occur and the CALL will not run.

  – If the program being called is not an ILE/C program and the length of the value is less than 32 characters, the value will be padded on the right with blanks. The 32-character value will then be passed to the program.

  – If the program being called is not an ILE/C program and the length of the value is equal to or greater than 32 characters, the value will be passed as is. No padding will be done. The length of the value that the called program will receive will be equal to the length that the calling program passed.

  – If the program was written in ILE/C the length of the value is not significant in determining how a character value is passed. In this case, a null character will be placed immediately after the last character of the value. This is then passed.

Note that a character constant value is a value that is not a REXX number and is not enclosed in single quotation marks. A character constant value can be specified by putting the value within the command string or by first assigning the value to a REXX variable and then using the variable to cause REXX to build the final command. A constant can never be specified by using a pseudo-CL variable.

- If the parameter value is a character value and that value is specified either as a quoted constant or through the use of a pseudo-CL variable, the following will occur:

  – The value of the parameter will be all characters between the enclosing single quotation marks or the complete value of the pseudo-CL variable. No folding will be done in this case. Enclosing quotation marks are not considered part of the value and are removed before the value is passed. However, as discussed in the following section, there are considerations for

using single quotation marks within a value that is assigned to a pseudo-CL variable.

– The quoted constant is processed in the same manner as described for a constant value.

Any constant value that is between single quotation marks is always considered a character value and never a numeric value. A quoted constant can be specified in one of two ways:

1. The value, including the enclosing single quotation marks, can be entered directly within the command.

2. The value, including the enclosing single quotation marks, can be assigned first to a REXX variable. The REXX variable can then be used within an expression which will cause REXX to build the final command using the REXX variable.

In addition, a pseudo-CL variable can be used. If the value within a pseudo-CL variable is not a REXX number, then it is assumed to be a character value. A character value within a pseudo-CL variable is treated as though it is enclosed in single quotation marks. There is no need to actually have the quotation marks as part of the value. If quotation marks are present, however, they will be considered as part of the value rather than as special delimiters. It is important to note the difference between usual REXX variables and pseudo-CL variables in this area.

- There is no enforcement to ensure that the program being called has declared the parameters properly. Only program convention between the calling and the called programs ensures the parameter values are handled properly. Note that the program being called need not receive the entire value being passed. For example, the program that is called may only use the first 10 characters of a 20 character value. This does not cause a problem as long as the first 10 characters are sufficient for the program to do its work.

- When a pseudo-CL variable passes a parameter value, the program that is called can effectively change the value of that variable. The change is reflected back into the calling program. This is the method that can be used to return a value to the calling program. However, the following precautions should be observed:

  – The return value should be of the same data type as the received value. For example, if a packed decimal (15,5) value is received, a packed decimal (15,5) value should be returned. Similarly, a return value for a parameter that was received as a double precision floating point number should also be in double precision floating point format. Where necessary, the return value will be automatically converted to a proper format for the REXX program to use.

  – For character parameters, the length of the returned value should not exceed the length of the received value. There is no enforcement to ensure that this does not happen. Should the return length be greater than the received length, the additional characters could cause unexpected results to occur when the calling program is returned to. To ensure that this does not happen, both the calling program and the program being called must observe the same conventions for passing parameter values.

    For ILE/C programs, when determining the maximum length that can be returned, only the length of the actual value received should be considered.

The null character at the end of a received value does not allow one additional character to be returned.

– If the length of the returned value is less than the length of the received value, residual data may be left over from the received value. Depending on how the returned value is assigned, the returned value may simply overlay the first part of the original received value, leaving the remainder of the received value intact. The calling program could thus receive data that is incorrect. To ensure that this does not happen, the parameter should be cleared, if necessary, such as setting it to blanks, before the return value is assigned.

An ILE/C program should not use a null character to mark the end of a return value. REXX does not recognize the null character as a special delimiter.

Pseudo-CL variables can be used to pass a value to the called program and also lets the called program return a value. In contrast, a usual REXX variable only lets a value be passed to the called program.

Understanding the Delete File and Remove Member Commands

Both the Delete File (DLTF) and Remove Member (RMVM) commands can be run from within a REXX program. Caution must be used when the source member which is to be deleted or removed is the one that contains the REXX program which caused the command to run in the first place. Doing this is equivalent to a program deleting itself.

If the source type of the member is not REXX, the member can be deleted or removed without affecting the running of the REXX program. The delete or remove will be done, and the REXX interpreter will continue to run the program. In this case, the internal form of the program is not directly associated with the member. Deleting or removing the member does not affect the internal form. If the source type of the member is REXX, deleting or removing the source member will cause the internal form of the REXX program to be deleted at the same time. As a result, after the member has been deleted or removed, the REXX interpreter will be unable to continue running the REXX program and will stop with an error condition.
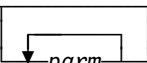
When the source type is REXX, the source member will be suitably locked for the duration of the run. Thus it will not be possible for one job to delete the source file or remove the source member while the REXX program is being run in a second job. The precaution mentioned above applies when the DLTF or RMVM command is run from within the same job that the REXX program is running in. When the source type is not REXX, the member will be locked only during the time that REXX is reading the source and building the internal form. After the internal form is built, the source will be unlocked so any other job can use the member.

Understanding the CPICOMM Command Environment

You can use the ADDRESS CPICOMM statement in your REXX program to call program-to-program communications routines. These communications routines must be part of Common Programming Interface (CPI) Communications, which is defined in IBM's Systems Application Architecture.

CPI Communications routines are described in the *SAA Common Programming Interface Communications Reference*.

Here is the format to use when calling a CPI Communications routine from a REXX program:

```
►►──ADDRESS CPICOMM 'rtnname─────────────────return_code'──►◄
                              └─◄─┬─parm─┬─┘
```

rtnname
   is the name of the CPI Communications routine to be called.

parm
   is the name of one or more parameters to be passed to the CPI
   Communications routine. The number and type of these parameters are
   routine-dependent. A parameter being passed must be the name of a variable.

return_code
   is the name of a parameter to receive the *return_code* from the CPI
   Communications call. Do not confuse this *return_code* from CPI
   Communications with the reserved REXX variable RC. The reserved REXX
   variable RC is the return code from the CPICOMM command environment. CPI
   Communications values for *return_code* are described in the *SAA Common
   Programming Interface Communications Reference*.

The following is a typical example using the ADDRESS CPICOMM statement:

```
ADDRESS CPICOMM 'CMINIT CONV_ID SYM_DEST_NAME RETURN_CODE'
```

Note: If REXX successfully calls the CPI Communications routine, then the REXX
variable *RC* contains a zero return code. Any value in *return_code* is the return
code from the called CPI Communications routine; its value is greater than or equal
to zero.

If REXX detects an error, (for example, is **not** able to successfully call the CPI
Communications routine), then the REXX variable *RC* contains a negative return
code. A message is issued to the job log describing the nature of the error. Any
value in *return_code* is meaningless because the CPI Communications routine was
not successfully called.

Understanding the EXECSQL Environment
You can use the ADDRESS EXECSQL statement in your REXX program to issue
Structured Query Language (SQL) statements to access your DB2/400 databases.
Here is the format to use when issuing an SQL statement:

```
►►──ADDRESS──EXECSQL──SQL_statement──►◄
```

SQL statements are described in the *DB2 for AS/400 SQL Reference*. Following is
a typical example using the ADDRESS EXECSQL statement:

```
ADDRESS EXECSQL 'INSERT INTO DB/TABLE VALUES(789)'
```

The EXECSQL environment, including instructions for using REXX variables to set
and receive data, is described in the *DB2 for AS/400 SQL Programming*. This
book also contains instructions for creating a REXX program that can use the
EXECSQL environment when the DB/400 Query Management and SQL
Development Kit Version 3 LPP, 5763-ST1, is not installed.

Return Codes from EXECSQL: The REXX variable RC indicates either a successful call to the SQL interface or a failure to call the SQL interface. If the call completes successfully, the REXX RC variable is set to zero. If the call cannot be completed, the REXX RC variable is set to a negative number and the failure condition is raised. For more information on values returned in the REXX RC variable for the EXECSQL environment, the SQL communications area (SQLCA), and the SQL description area (SQLDA), refer to the *DB2 for AS/400 SQL Reference*.

Chapter 8. Using REXX Functions and Subroutines

REXX programs often use functions and subroutines. REXX provides many functions which may be called in any REXX program. These are known as built-in functions. Subroutines and functions have a lot in common, and when discussed at the same time, they are called routines.

The following topics are covered in this chapter:

- Understanding Functions and Subroutines
- Using Internal Routines
- Using External Routines
- Accessing Parameters
- Returning Results
- Understanding the Function Search Order
- Using REXX Built-in Functions
- Understanding Conversion Functions.

Understanding Functions and Subroutines

REXX lets you create and call your own internal and external functions and subroutines. Internal routines are identified by labels within a REXX program, and can only be called from inside that REXX program. External routines are in separate source members and may be called by different REXX programs. External routines may be written in REXX or in other languages which support the REXX external function interface.

Both internal and external routines may call themselves. This is called *recursive invocation*.

You can include function calls to internal, built-in, and external routines using the following syntax:

```
►►──function_name(─────────────────────────────)──►◄
                    └─expression─┬──────────────┘
                                 └─,expression─┘
```

Up to 20 parameters are allowed on a function call. The parameters may be of any length, and some parameters may be omitted when the function being called allows that. A distinction is made by REXX between a null value and an omitted parameter. The called program can distinguish between the two. You can distinguish between the two by using the EXISTS and OMITTED parameters on the ARG function. For more information on the ARG function, see the *REXX/400 Reference*.

When a function call is run, the expressions are evaluated and passed to the function, and the routine calculates a return value. That return value is used in the calling REXX program where the function call appears.

Understanding the Differences Between Functions and Subroutines

In REXX, functions and subroutines are quite similar. There is one difference between them in the way they are called and in the way the routines end.

- Functions are run when they are written as part of an expression, such as function-name (expression) and subroutines are run when they are named on the REXX CALL instruction. The CALL instruction syntax is as follows:

```
►►──CALL──routine_name──┬───────────────────────┬──►◄
                        └─expression─┬──────────┘
                                     └─,expression─┘
```
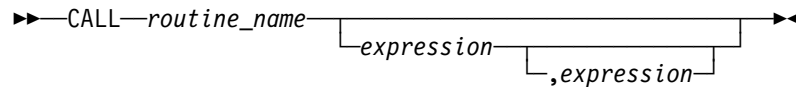
  Note that the parameters on CALL are not enclosed in parentheses. As with function calls, up to twenty parameters are allowed on a CALL instruction, and parameters may be omitted.

- Functions must return a value when they end, but subroutines may end without returning a value. When a routine is run as a subroutine, and it returns a value, that value is assigned to the REXX special variable RESULT. If it does not return a value, RESULT is dropped, and is returned to its default value.

All REXX built-in routines return a value, so they are true functions.

Any routine that returns a value may be called either as a function or as a subroutine. All routines may be called as subroutines. This is true for built-in functions, as well as internal and external routines. Here is an example of calling some REXX built-in functions both ways:

```
/* Calling built-in functions as functions and subroutines.      */
today = DATE()                   /* Call date as a function.     */
SAY "Today's date is" today
CALL Date                        /* Call date as a subroutine.   */
SAY "Today's date is" RESULT     /* Now the data is in 'result'. */
SAY "The current time is" TIME() /* Call time as a function.     */
CALL Time                        /* Call time as a subroutine.   */
SAY "The current time is" result /* Now the time is in 'result'. */
```

Using Internal Routines

Internal routines are identified by labels within the same source member as the main program which calls them. They end when they run a RETURN instruction. If a value is specified on the RETURN instruction, that value is returned by the routine. If no value is given on the RETURN instruction, then no value is returned. If an EXIT instruction is run within an internal routine, the program will stop immediately, exactly as it would if the EXIT was run in the main program.

The variables in the main program are available in internal routines. If you want to create a new generation of variables in an internal routine, you can use the PROCEDURE instruction, which is described in "Using the PROCEDURE Instruction" on page 27.

Here is an example of an internal subroutine which uses the main program's variables as its input, and sets a variable in the main program as its output.

```
/* A simple example of using the CALL instruction.                */
SAY "This is the main program"
DO num = 1 to 5
   CALL square                      /* This calls the subroutine.     */
   SAY "Back in the main program."
   SAY num "squared is" num2        /* This displays the result.      */
END
EXIT                                /* This ends the program.         */

square:                             /* The subroutine begins here.    */
SAY "This is the subroutine."
num2 = num * num                    /* This calculates the square.    */
RETURN                              /* This resumes the main program. */
```

Using External Routines

External routines are functions or subroutines which are not part of the source member of the calling program. External routines can be written in REXX or other languages.

Understanding External Routines Written in REXX

To create an external function written in REXX, you create a source member with the name you want to give the routine. You must put this member in the first QREXSRC file in your library list or put it in the same source file as the REXX programs that will be calling it.

External routines do not start with labels and cannot start with the PROCEDURE instruction. They cannot access the variables of the REXX program which called them. They can end with either the RETURN or EXIT instructions.

External routines may have internal routines within them. In that case, a RETURN instruction in the internal routine returns to the point in the external routine that did the calling, not to the program which called the external routine. These internal routines may use the PROCEDURE instruction.

Understanding External Routines Written in Other Languages

To create an external function written in another language, you must create a program which is written to accept parameters according to the way the REXX external function interface passes them. See the *REXX/400 Reference* for that definition. Give the program the name that you want REXX to know it by, and put it in a library which is in your library list. If the external routine is written in an ILE language, C/400, or Pascal, then it must be a main program.

External routines in other languages can access the variables of the REXX program which called them by using the variable pool interface service QREXVAR. External functions use QREXVAR to set their return value. For more information on QREXVAR, see Appendix G, "Communication Between REXX/400 and ILE/C" on page 175 and the *REXX/400 Reference*.

Accessing Parameters

The rules for accessing the parameters passed to routines are the same for functions and subroutines, and are the same for internal routines and external routines written in REXX. Formal parameters are accessed with the ARG, PARSE ARG, and PARSE UPPER ARG instructions and the ARG built-in function.

In REXX, parameters may contain many different words and character strings. The following examples pass only a single parameter to ROUTINE1 even though many pieces of information are used.

```
CALL routine1 "first second third"    /* This parameter contains    */
                                       /* three words.               */
CALL routine1 "first, second, third"  /* This parameter contains    */
                                       /* three words, with commas.  */
CALL routine1 first second third       /* This an expression made    */
                                       /* from values of three       */
                                       /* variables.                 */
CALL routine1 "Hello" 5+1 substr("testing",1,max(4,length("testing")))
                                       /* This is an expression made */
                                       /* from a combination of      */
                                       /* operations and function    */
                                       /* calls.                     */
```

When you want to pass several pieces of information as separate parameters, separate them with commas:

```
CALL routine1 "Hello","Greetings", "Salutations"
```

When using ARG and PARSE ARG, you specify the arguments you want by the way you write the parsing template. For example, if you wanted to get the first, third, and fourth arguments passed to a routine, you could use the following instruction:

```
ARG first,,third,fourth
```

You can omit some of the arguments on a call to a routine, as in the following example:

```
CALL routine1 first,,,fourth, fifth
```

In this case, an ARG instruction in ROUTINE1 which attempted to access the second or third arguments would get null results for those arguments.

The ARG built-in function also accesses the arguments passed to a routine. If you pass it a number, as in `ARG(2)`, then it will return the argument passed in that position, or a null string if that argument was omitted. If you call ARG with no parameters, it will return the number of arguments passed to the routine.

You can call the ARG and PARSE ARG instructions, and the ARG built-in function, as many times as you need.

Returning Results

Internal routines end with the RETURN instruction. Processing continues with the instruction following the function or subroutine call. The full form of the instruction is:

```
►►──RETURN──────────────►◄
              └─expression─┘
```

where, if *expression* is specified, it will be used as the return value for the routine. The value of *expression* may be a number, or a character string of any length. If you want a routine to return several pieces of information, you can create a return string consisting of several words. The calling program may then break up the string just like any other string, using instructions like PARSE and functions like SUBSTR.

External routines end with either the RETURN instruction or the EXIT instruction. Both of these instructions may specify an expression to return as the function value. When EXIT ends an external routine, the expression value may be a string. When EXIT ends the main program, the expression value may only be a number.

Understanding the Function Search Order

When REXX encounters a function call or subroutine call, it searches for the routine in the following order:

1. An internal routine (a label within the same program)

2. A built-in function

3. External routines in the following order:

    a. A member in the same source file as the calling routine

    b. A member in the file QREXSRC in your current library

    c. A member in the first QREXSRC file in your library list.

You can alter this search order by placing the routine name in quotation marks. REXX then will not capitalize the function name and will bypass the search for internal routines. By selective use of quotation marks on routine names, you can override the name of a built-in or external function by creating an internal routine with the same name, and still be able to call the built-in or external routine when you want to.

```
/* When you call a REXX function, REXX normally searches for an    */
/* internal function before searching for a built-in or external   */
/* function.  However, if you put the name of the function in       */
/* quotation marks, REXX will bypass the search for an internal     */
/* function.  This lets you 'override' built-in REXX functions      */
/* in your program.                                                 */

/* This program overrides the DATE built-in function to provide a   */
/* Date function which has a different default and also an extra    */
/* option.                                                          */

SIGNAL main       /* Branch past the internal function.             */
```

```
                     /* This is the routine that changes the behavior of the DATE    */
                     /* function.  The default is 'E' format, and 'Y' format is defined */
                     /* to be yyyyddd.                                               */

                     Date: PROCEDURE
                     ARG format
                     SELECT
                       WHEN format = '' THEN answer = 'DATE'('E')
                       WHEN format = 'Y' THEN DO
                         PARSE VALUE 'DATE'('S') WITH answer +4 .
                         answer = answer||'DATE'('D')
                         END
                       OTHERWISE answer = 'DATE'(format)
                     END
                     RETURN answer


                     main:            /* The main program starts here.              */

                     SAY DATE()       /* Shows the date in 'E' format:  dd/mm/yy.   */
                                      /* This function call goes to the internal function.*/

                     SAY 'DATE'()     /* Shows the date in REXX default format: dd mon yy.*/
                                      /* This function call goes to the built-in function.*/

                     SAY DATE('Y')    /* Shows the date in the new 'Y' format.      */
                                      /* This function call goes to the internal function.*/

                     SAY 'DATE'('Y')  /* Causes an error because the REXX date function */
                                      /* does not have a 'Y' option.                */
                                      /* This function call goes to the built-in function.*/

                     EXIT
```

Using REXX Built-in Functions

The built-in functions allow you to perform many different types of operations. In this book, many of these functions are discussed, including:

- String functions (see "Using String Functions" on page 72)

- Conversions functions (see "Understanding Conversion Functions" on page 111)

- Numeric functions, such as MAX, MIN, ABS, and RANDOM

- Double-byte character set functions (see Appendix C, "Double-Byte Character Set Support" on page 141)

- Informational functions such as ADDRESS, DATATYPE, DATE, ERRORTEXT, SOURCELINE, TIME, TRACE, and VALUE.

For a complete description of the built-in functions, see the *REXX/400 Reference*. For a complete list of the built-in functions, see Appendix B, "REXX Built-in Functions" on page 139.

Using the ADDRESS Built-in Function

The ADDRESS function returns the name of the current command environment, as shown in the following example:

```
SAY ADDRESS()      /* Shows the default command environment "COMMAND".*/
ADDRESS 'MYLIB/APP1'
SAY ADDRESS()      /* Shows the new environment "MYLIB/APP1".        */
```

Command environments are discussed in Chapter 7, "Understanding Commands and Command Environments" on page 79. The ADDRESS instruction can be used to set the command environment within a REXX program. The string used with the ADDRESS instruction is returned by the ADDRESS function. This means that if the command environment is set using ADDRESS '*LIBL/MYPROG', the ADDRESS function will return *LIBL/MYPROG.

Using the DATE Built-in Function

The DATE function returns the current system date, which may be different from your job date. There are several options for the format of the date returned.

```
SAY DATE()       /* Shows the default:  dd mon yyy; for example,    */
                 /* 27 August 1988.  'mon' is always the first three */
                 /* letters of the English name of the month.       */
SAY DATE('E')    /* Shows the date in "European" format:  dd/mm/yy;  */
                 /* for example, 27/08/88.                          */
SAY DATE('U')    /* Shows the date in "USA" format:  mm/dd/yy;       */
                 /* for example, 08/27/88.                          */
SAY DATE('S')    /* Shows the date in "Standard" format:  yyyymmdd;  */
                 /* for example, 19880827.                          */
```

If you need to know your job date instead of the current system date, use the Retrieve Job Attributes (RTVJOBA) command.

Using the ERRORTEXT Built-in Function

The ERRORTEXT function returns the text of the message associated with a REXX error number. This function is commonly used in error-handling routines, where REXX provides the error number to the routine.

Here are a few examples of the ERRORTEXT function:

```
SAY ERRORTEXT(40)   /* Shows the text of REXX Error 40              */
                    /* "Incorrect call to routine".                */
SAY ERRORTEXT(16)   /* Shows the text of REXX Error 16              */
                    /* "Label not found".                          */
SAY ERRORTEXT(35)   /* Shows the text of REXX Error 35              */
                    /* "Invalid expression".                       */
```

Using the FORMAT Built-in Function

The FORMAT function adjusts, rounds, and formats a number to fit in a certain amount of space. FORMAT is an example of a string function. You can specify several arguments to the FORMAT function. These arguments are detailed in the *REXX/400 Reference* and described here.

```
              SAY FORMAT(3.14159)        /* This displays "3.14159", which    */
                                         /* is the number 3.14159 rounded     */
                                         /* and formatted to REXX defaults.   */
              SAY FORMAT(3.14159,,2)     /* This displays "3.14", which is     */
                                         /* the number 3.14159 rounded and     */
                                         /* formatted to show 2 places to the*/
                                         /* right of the decimal point.        */
              SAY FORMAT(3.14159,2,2)    /* This displays " 3.14", which is    */
                                         /* the number 3.14159 rounded and     */
                                         /* formatted to show 2 places to the*/
                                         /* right and 2 places to the left     */
                                         /* of the decimal point.              */
              SAY FORMAT(12345.73,,3,,0) /* This displays "1.235E+4", which    */
                                         /* is the number 12345.73 in          */
                                         /* exponential notation with 3        */
                                         /* places to the right of the         */
                                         /* decimal point.                     */
```

Using the MAX and MIN Built-in Functions

The MAX function returns the largest number from a specified list of numbers, up to 20. The MIN function returns the smallest number from a specified list of numbers, up to 20.

The following are some examples:

```
big = MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
                                /* MAX returns '20', big is assigned 20. */
small = MIN(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,20)
                                /* MIN returns '1', small is assigned 1. */
```

If your list is longer than 20 expressions, you can nest calls to MAX or MIN as shown in the following examples:

```
bigger = MAX(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,,
        MAX(18,19,20,21,22,23,24,25,26,27,28,29,30))
                          /*  MAX returns '30', bigger is assigned '30'.   */

smaller = MIN(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,16,17,18,19,,
        MIN(20,21,22,23,24,25,26,27,28,29,30))
                          /*  MIN returns '1', smaller is assigned '1'.    */
```

Using the SETMSGRC Built-in Function

The SETMSGRC function lets you control what REXX does with status and notify messages sent to the program message queue. The SETMSGRC function can specify certain messages or a range of messages that, if received, will be returned to the program in the special variable RC.

Example 1: This example is a program that calls ABCPROG while monitoring for one particular message. Program ABCPROG issues a status message when it finds that a data file it uses must be expanded. The program also issues other messages for serious errors.

```
SIGNAL ON ERROR
SAY 'Do you want the data file to be automatically expanded? (Y or N)'
PULL answer .
IF left(answer,1) ¬= 'Y' THEN
   CALL SETMSGRC 'SET','CPZ6604'

'CALL ABCPROG'
SAY 'ABCPROG ran successfully.'
RETURN

Error:
IF RC = 'CPZ6604' THEN DO
   SAY 'Data file needs to be expanded.  Please expand the file'
   SAY '  and then try this program again.'
END
ELSE
   SAY 'ABCPROG had a serious error.  Message number is' RC
RETURN
```

Example 2: This program issues a command called ABCPROG while monitoring for a range of messages.

```
SIGNAL ON ERROR
SAY 'Do you want minor errors to stop ABCPROG? (Y or N)'
PULL answer .
IF left(answer,1) = 'Y' THEN
   CALL SETMSGRC 'SET','CPZ6601:CPZ6609'

'CALL ABCPROG'
SAY 'ABCPROG ran successfully.'
RETURN

Error:
IF RC >= 'CPZ6601' & RC <= 'CPZ6609' THEN DO
   SAY 'A minor error occurred while ABCPROG was running.  The'
   SAY '  program was stopped.'
END
ELSE
   SAY 'ABCPROG had a serious error.  Message number is' RC
RETURN
```

Example 3: This example uses the PUSH option to preserve any previous settings.

```
SIGNAL ON ERROR
SAY 'Do you want minor errors to stop ABCPROG? (Y or N)'
PULL answer .
IF left(answer,1) = 'Y' THEN
   CALL SETMSGRC 'PUSH','CPZ6601:CPZ6609'
ELSE
   CALL SETMSGRC 'PUSH'  /* Don't trap ANY status messages */

'CALL ABCPROG'
SAY 'ABCPROG ran successfully.'
CALL SETMSGRC 'RESTORE'  /* Restore old message ID list    */
RETURN

Error:
IF RC >= 'CPZ6601' & RC <= 'CPZ6609' THEN DO
   SAY 'A minor error occurred while ABCPROG was running.  The'
   SAY '  program was stopped.'
END
ELSE
   SAY 'ABCPROG had a serious error.  Message number is' RC
CALL SETMSGRC 'RESTORE'  /* Restore old message ID list    */
RETURN
```

Example 4: In this example the QUERY function returns the current messages handled. The message list is searched for a specific message being monitored.

```
SIGNAL ON ERROR
cur_msgids = SETMSGRC('QUERY')

/* To add CPZ6604 to the list of status messages monitored,    */
/* concatenate cur_msgids and 'CPZ6604' and set the new list   */
/* by PUSHing it, saving the old list in the process.          */

IF cur_msgids ¬= '' THEN         /* If the current list is not */
   cur_msgids = cur_msgids ','   /* empty, add a comma to it   */
CALL SETMSGRC 'PUSH', cur_msgids 'CPZ6604'

'CALL ABCPROG'
SAY 'ABCPROG ran successfully.'
CALL SETMSGRC 'RESTORE'
RETURN

Error:
IF RC = 'CPZ6604' THEN DO
   SAY 'Data file needs to be expanded.  Please expand the file'
   SAY '  and then try this program again.'
END
ELSE
   SAY 'ABCPROG had a serious error.  Message number is' RC
RETURN
```

SETMSGRC settings are not saved and restored over function calls. Internal and external REXX functions and subroutines inherit the SETMSGRC settings of their caller, and any changes they make remain in effect when they return to their caller.

For more information on the SETMSGRC function, see the *REXX/400 Reference*.

Using the SOURCELINE Built-in Function

The SOURCELINE function returns the *n*th line in the currently running REXX program, if you specify *n*. If you omit *n*, then the number of lines in the REXX program is returned, as shown in the following example:

```
SOURCELINE(1) might return '/*  This is a 10-line program  */'
SOURCELINE()  returns 10   /* when the program has 10 lines */
```

The following example uses SOURCELINE function in a program.

```
/* This program asks you to enter an expression.  It then evaluates  */
/* the expression and tells you the result.  If the expression       */
/* has an error in it, the program will tell you what that error is. */


/* This program does one other thing.  If you call it with           */
/* a parameter of 'help', it will write out the first three lines of */
/* the program, to explain what the program does.                    */
/* It uses the REXX built-in function SOURCELINE to do this.          */


ARG parm

IF parm = 'HELP' THEN DO i = 1
   helpline = SOURCELINE(i)
   SAY helpline
   IF helpline = '' THEN LEAVE    /* Quit on the first blank line.  */
   END

SAY 'Enter an expression'
PULL expr
Signal on Syntax
INTERPRET 'answer =' expr
SAY 'That expression evaluates to' answer
EXIT

Syntax:
SAY 'Your expression had an error in it.'
SAY 'The error was' ERRORTEXT(RC)
EXIT
```

Using the TIME Built-in Function

The TIME function returns the time in the 24-hour clock format. However, there are several options that allow you to obtain alternative formats. The following are examples:

```
SAY TIME()    /* Shows time in the default format:         */
              /* hh:mm:ss                                  */
SAY TIME(H)   /* Shows time in "Hours" format:  hh         */
SAY TIME(M)   /* Shows time in "Minutes" format: mm        */
SAY TIME(S)   /* Shows time in "Seconds" format: ss        */
```

For more information on other options, refer to the *REXX/400 Reference*.

The TIME function can also perform elapsed-time calculations. The following
example measures the amount of time it took to run the Send Network File
(SNDNETF) command:

```
Call TIME 'R'
'SNDNETF FILE(MYLIB/BIGFILE) TOUSRID(THEBOSS)'
elapsed = TIME('E')
SAY 'The SNDNETF command took' elapsed 'seconds.'
```

Using the TRANSLATE Built-in Function

The TRANSLATE function translates characters within a string or to reorder the
characters in a string. In the following example, the TRANSLATE function changes
punctuation.

```
/* Using the TRANSLATE function to change                       */
/* unwanted characters to blanks.                               */

TEXT= "Listen, my children, and you shall hear",
      "Of the midnight ride of Paul      Revere"

SAY WORDPOS("my children", TEXT) /* Displays '0', because the    */
                                 /* word in TEXT is             */
                                 /* 'children'.                 */

/*                                                              */
/* Say whether 'my children,' can be found in' TEXT.            */
/*                                                              */

nopunct= TRANSLATE(TEXT,"       ",".;:!,?")
                                      /* Remove punctuation.    */
SAY SIGN(WORDPOS('my children', nopunct)) /* Displays '1'.      */

SAY SIGN(WORDPOS('kids', nopunct))        /* Displays '0'.      */
```

To help make up strings to put in a translation table, you can use the XRANGE
built-in function. For more about this function, see the *REXX/400 Reference*.

Using the VERIFY Built-in Function

You can use the verify function to find out whether a string contains only characters
of a particular set of characters.

```
VERIFY(string,reference)
```

returns the position of the first character in *string* that is not also in *reference*. If all
the characters in *string* are also in *reference*, zero is returned.

The following is an example:

```
/* Test that all input characters are valid.                  */

SAY "Please enter the serial number"
SAY "(eight digits, no embedded blanks or periods)"

PULL serial rest
IF VERIFY(serial, "0123456789")=0,
   & LENGTH(serial)= 8,
   & rest = ""
THEN SAY "Accepted"
ELSE
```

```
SAY "Serial number not valid.  Please enter another serial number."
```

Understanding Conversion Functions

At times you may need to write a program that works with the hexadecimal or binary representations of some of its data. REXX provides several built-in functions to help you do this. These are called *conversion functions*.

Understanding Data Formats

Data in REXX programs is normally entered, stored, and used in character form. REXX also lets you enter literal strings in your program in hexadecimal or binary form, and REXX will automatically convert them to character form when the program runs.

To enter data in hexadecimal form, enclose the hexadecimal data within quotation marks, and follow it with the letter X. This is a *hexadecimal string.* Here is an example that shows how the letter A can be entered in hexadecimal instead of as a character. The letter A is stored as the hexadecimal number C1.

```
lettera = 'C1'X
IF lettera = 'A' THEN SAY 'Yes, they are equal'
   ELSE SAY 'You will never see this message.'
```

To enter data in binary form, enclose the data within quotation marks, and follow it with the letter B. This is a *binary string.*

```
lettera = '1100 0001'B
IF lettera = 'A' THEN SAY 'Yes, they are equal'
   ELSE SAY 'You will never see this message.'
```

Using Conversion Functions

The conversion functions give you more ways to work with data in special formats in addition to the literals above. The literals only allow you to enter data in your program in those special forms. The conversion functions let you change the format of data between any of the formats that REXX supports.

Here is an example. The C2X function takes one or more characters and returns a character string that shows the hexadecimal representation of the input string.

```
SAY "The output of C2X('A') is" C2X('A')  /* Displays C1. */
```

This function always produces an output string with twice as many characters as the input string, because every character is represented by two hexadecimal digits.

The following table lists the data formats supported by REXX. The character indicators listed are used in the names of the conversion functions to indicate the type of data used by the function.

Data Format	Character Indicator	Data Description
Binary	B	Data consisting of the characters 0 and 1, not including binary strings
Character	C	Data consisting of any characters
Decimal	D	Data consisting of the characters 0-9
Hexadecimal	X	Data consisting of the characters 0-9, a-f, and A-F, not including hexadecimal strings

Names of the Conversion Functions

The conversion functions have three-character names. The first indicates the input data type of the function, the second is always the digit 2, and the last indicates the format of the output of the function.

These functions are fully defined in the *REXX/400 Reference*. Here are some examples of using the functions. These examples all show use of the functions on data which represents just one byte, but longer strings of characters may be used.

When any of these functions produce output showing a hexadecimal number, the alphabetic characters are returned in uppercase. When these functions are given input in hexadecimal form, it may be in uppercase or lowercase.

Function Call	Output	Description
B2X('1100 0001')	C1	The input to this function is a character string consisting of ones and zeros. The data cannot be entered as a binary string. You can place blanks between groups of four characters to make it easier to read.
B2X('1111 0000'B)	0	This binary string represents the character '0'. This binary zero is padded with zeros to a length of four, and this converts to a hexadecimal zero.
B2X('1100 0001'B)	Error	This binary string represents the character 'A'. This is not a binary character, so it causes an error.
C2D('A')	193	The decimal value of the character encoding for A is 193. Note that the value 193 produced by this function is in character form. You can perform REXX arithmetic in the usual way on the output of this function.
C2X('A')	C1	The hexadecimal value of the character encoding for A is C1.
D2C(193)	A	This is the reverse of C2D.
D2X(193)	C1	The decimal number 193 is the same value as the hexadecimal number C1.
X2B('C1')	1100 0001	This converts character strings representing hexadecimal values to character strings representing binary values. You can place blanks between pairs of characters to make it easier to read.
X2C('C1')	A	This is the reverse of C2X.
X2D('C1')	193	This is the reverse of D2X.

You may have noticed that not all possible conversion functions are provided. If you want to perform a conversion not provided by one of these functions, you can do that by combining the functions as shown in the following table.

To change *n*	to Binary	to Character	to Decimal
From Binary		X2C(B2X(n))	X2D(B2X(n))
From Character	X2B(C2x(n))		
From Decimal	X2B(D2X(n))		

Chapter 9. Using the REXX External Data Queue

The REXX external data queue provides a way to temporarily hold data which REXX, and any suitably tailored application programs, can use. The data on the queue is accessible by and visible to users as lines or as buffers. A buffer is a sub-grouping of lines within a queue, and lines are character strings of arbitrary lengths. Each line can contain up to 32,767 characters. The individual characters have no special meaning or effect to REXX. The external data queue can be used to replace user input.

The data on the queue can be used by REXX programs and user-written programs in an arbitrary manner. Thus, the REXX queue services can be used as a way of exchanging data between programs, providing a device for inter-program communication.

Learning About the REXX External Data Queue

A REXX external data queue comes into existence when a job is started and persists until the job is ended. All programs that run under the same job have access to that external data queue.

The following operations can be performed on the REXX external data queue:

- A line can be placed at the end of the current queue buffer.
- A line can be placed at the front of the current queue buffer.
- A line can be retrieved from the front of the queue.
- The number of lines on the queue can be queried.
- A new queue buffer can be created.
- A queue buffer can be removed.
- The entire queue can be cleared.

These operations are made available directly to a REXX program through REXX instructions and CL commands. The same operations can be performed within other programming languages through the queue services application program interface (QREXQ). For more information on this interface, see the *REXX/400 Reference*. Some examples are provided in the Appendix F, "Sample REXX Programs for the AS/400 System" on page 157.

Using the REXX Queue Services on the AS/400 System

REXX provides the following instructions and functions for working with the queue:

QUEUE	This instruction places a given line at the end of the current queue buffer.
PUSH	This instruction places a given line at the front of the queue.
PULL	This instruction retrieves a line from the front of the queue or reads from STDIN.
PARSE UPPER PULL	This instruction retrieves a line from the front of the queue or reads from STDIN.
QUEUED	This built-in function determines the number of lines in the queue.

These provide all the necessary tools for working with the queue. In addition, the Add REXX Buffer (ADDREXBUF) and Remove REXX Buffer (RMVREXBUF) commands are available to REXX programs for working with queue buffers. These commands extend the flexibility of the REXX queue instructions.

Starting Queuing Services

When a job is started, the queuing services are immediately made available to the job. These services remain available until the job ends. Similarly, entries placed on the queue will remain on the queue and be available until explicitly removed or until the job is ended. After starting a job, any program that is part of the job can continue to use any of the queuing services until the job ends. Thus, one program can place an entry on the queue and end. Another following program can then receive what was placed in the queue at a later time. This can serve as a device for inter-program communication.

Understanding Queue Management Instructions

This section discusses the instructions used to manage the REXX external data queue.

Using the PUSH Instruction

The PUSH instruction expects a REXX string which is then placed at the front of the queue. The PUSH instruction imitates pushing data onto a stack. That is, the last data placed by PUSH is the data at the front of the queue. The following example illustrates the use of the PUSH instruction.

```
/* Using PUSH and PULL.                                          */
SAY 'To start with, the queue has' queued() 'elements.'
                                    /* QUEUED() returns 0 elements  */
PUSH TIME()          /* Add the current time to the queue.       */
PUSH DATE()          /* Add the current date to the queue.       */
                     /* This goes before what is already in the queue.*/
SAY 'Now the queue has' queued() 'elements.'
                                    /* QUEUED() returns 2 elements  */


PULL data1           /* PULL gets the date off the queue.        */
PULL data2           /* PULL gets the time off the queue.        */

SAY 'The first element on the queue was' data1
SAY 'The second element on the queue was' data2
SAY 'Now the queue has' queued() 'elements again.'
                                    /* QUEUED() returns 0 elements. */

EXIT
```

Using the QUEUE Instruction

The QUEUE instruction places a string at the end of the current queue buffer. The QUEUE instruction imitates the classical manner of placing information on a queue. That is, the last data placed by QUEUE is the data at the end of the queue and would be the last data available for a PULL. The following example illustrates the use of the QUEUE instruction.

Example 1: Using the PULL and QUEUE instructions

```
SAY 'To start with, the queue has' queued() 'elements.'
                            /* QUEUED() returns 0 elements. */
QUEUE TIME()        /* Add the current time to the queue.       */
QUEUE DATE()        /* Add the current date to the queue.       */
                    /* This goes after what is already in the queue.*/
SAY 'Now the queue has' queued() 'elements.'
                            /* QUEUED() returns 2 elements. */

PULL data1          /* PULL gets the time off the queue.        */
PULL data2          /* PULL gets the date off the queue.        */

SAY 'The first element on the queue was' data1
SAY 'The second element on the queue was' data2
SAY 'Now the queue has' queued() 'elements again.'
                            /* QUEUED() returns 0 elements. */

EXIT
```

Example 2: Using the PARSE LINEIN instruction.

```
/* This program illustrates how PARSE LINEIN reads a response from  */
/* the user, even when there are lines in the queue.                */

/* Suppose you want to send a message to selected users of your     */
/* system.  And suppose you already have an external function which */
/* puts a list of all users into the REXX queue.  This program shows */
/* how you can select some of the users listed by that function.    */

message = 'Please come to a meeting in my office at 10:00.'

CALL queryusers                     /* Function queues a list of users.*/

DO QUEUED()
   PULL userid
   SAY 'Enter YES if you want to send the message to' userid
   PARSE UPPER LINEIN answer
   IF answer = 'YES' THEN "SNDMSG MSG('"message"') TOUSR("userid")"
END
```

Example 3: This example shows a CL command that uses a REXX program as its command processing program. This example copies the contents of a file into the REXX external data queue.

```
          CMD          PROMPT('Copy file to REXX data que')

          PARM         KWD(FROMFILE) TYPE(QUAL1) MIN(1) PROMPT('FROM +
                         file')

          PARM         KWD(MBR) TYPE(*NAME) LEN(10) DFT(*FIRST) +
                         SPCVAL((*FIRST)) MIN(0) PROMPT('Member')

          PARM         KWD(NMBRCDS) TYPE(*DEC) LEN(6) DFT(*ALL) +
                         SPCVAL((*ALL 999999)) PROMPT('Number of +
                         records to copy')

 QUAL1:   QUAL         TYPE(*NAME) LEN(10) MIN(1)

          QUAL         TYPE(*NAME) LEN(10) DFT(*LIBL) +
                         SPCVAL((*CURLIB) (*LIBL)) PROMPT('Library')


/*  Command Processing REXX program for CPYFTOREXQ command.      */

                            /* Parse out the library and file.  */
PARSE UPPER ARG 'FROMFILE(' lib '/' file ')'

                            /* Parse out the member.            */
PARSE UPPER ARG 'MBR(' mbr ')'

                  /* Parse out the number of records to copy. */
PARSE UPPER ARG 'NMBRCDS(' count ')'

                            /* Check if object exists.          */
'CHKOBJ OBJ('lib'/'file') OBJTYPE(*FILE) MBR('mbr')'
IF rc ¬= '0' THEN
  IF POS(rc,'CPF9801 CPF9810') ¬= 0 THEN DO
    msg = 'File member specified:' lib'/'file mbr 'was not found'
    'SNDPGMMSG MSG(&msg)'
    EXIT
END

IF count = '*ALL' THEN count = '999999999'
                   /*  Override STDIN to the LIB/FILE parms.   */
   'OVRDBF FILE(STDIN) TOFILE('lib'/'file') MBR('mbr')'

DO count
                   /* Read data from STDIN.                    */
   PARSE LINEIN  data

   IF data == '' THEN
     LEAVE
           /* QUEUE data into REXX queue, FIFO order from PULL. */
   QUEUE data
END
EXIT
```

Using the PULL Instruction

The PULL instruction retrieves a line from the front of the queue. This instruction is equal to PARSE UPPER PULL, which uppercases the line pulled from the queue. To pull from the queue without uppercasing, use PARSE PULL. The PULL instruction imitates pulling data from a stack. That is, the last data placed by the PUSH instruction would be the first retrievable data by the PULL instruction. The following example illustrates the use of the PULL instruction.

```
/* The Pull instruction is a shorthand version of Parse Upper Pull.  */

test = 'This is a test'
PUSH 'test'
PUSH 'test'
PUSH 'test'
PARSE UPPER PULL input1
PULL input2

SAY 'Parse Upper Pull got' input1
SAY 'Pull got' input2

IF input1 == input2 THEN SAY 'The two different instructions work the same.'
   ELSE SAY 'You will never see this message.'
PARSE PULL input3
```

Whenever the queue is empty, the PULL instruction operates in a manner identical with the PARSE UPPER LINEIN instruction which reads lines from the STDIN. The PARSE UPPER LINEIN instruction can, however, be used directly even if the queue is not empty, when input from STDIN is also required or when there is data on the queue which must not be disturbed.

Using the Add REXX Buffer (ADDREXBUF) Command

The ADDREXBUF command expects a REXX variable into which the identification number of the newly created queue buffer would be returned. The command allows a REXX program to establish new buffers in the REXX external data queue. The ADDREXBUF command can only be used in REXX programs and CL programs. If the queue is not buffered with this command, it is treated as one large buffer.

```
PUSH  'LINE ONE'
PUSH  'LINE TWO'
PUSH  'LINE THREE'
QUEUE 'LAST LINE'
```

Assuming a previously empty REXX external data queue, these instructions would result in the following on the queue:
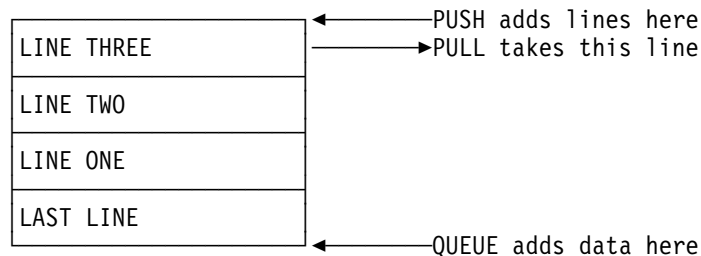
Figure 1. External data queue before ADDREXBUF

Running this set of instructions affects the queue as shown:

```
BUFFERNUM = 0
'ADDREXBUF &BUFFERNUM'
QUEUE 'NEWBUF FIRSTQ'
PUSH  'NEWBUF PUSH'
QUEUE 'NEWBUF SECONDQ'
```

```
                                         ┌──────── PUSH adds lines here
                                    ◄─────┤
┌─────────────────────┐            ──────┼──────► PULL takes this line
│ NEWBUF PUSH         │
├─────────────────────┤
│ NEWBUF FIRSTQ       │
├─────────────────────┤
│ NEWBUF SECONDQ      │
├─────────────────────┤    ◄──────────────── QUEUE adds data here
  │ LINE THREE          │
  ├─────────────────────┤
  │ LINE TWO            │
  ├─────────────────────┤
  │ LINE ONE            │
  ├─────────────────────┤
  │ LAST LINE           │
  └─────────────────────┘
```
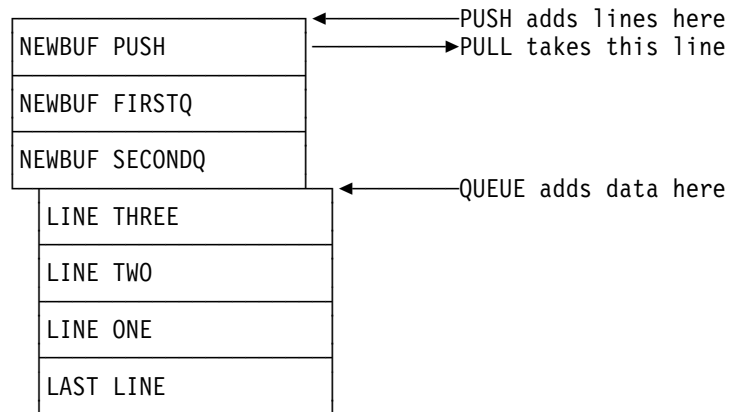
Figure 2. External data queue after ADDREXBUF

Using the Remove REXX Buffer (RMVREXBUF) Command

The Remove REXX Buffer (RMVREXBUF) command expects an identification number of the queue buffer which is to be removed. This command deletes buffers and any lines they may contain from the REXX external data queue.

Assuming the last example above, using ADDREXBUF, if the following REXX instructions were run:

```
 'RMVREXBUF' buffernum
 PULL aline
```

The REXX external data queue would then look like this:

```
                                         ┌──────── PUSH adds lines here
                                    ◄─────┤
┌─────────────────────┐            ──────┼──────► PULL takes this line
│ LINE TWO            │
├─────────────────────┤
│ LINE ONE            │
├─────────────────────┤
│ LAST LINE           │
└─────────────────────┤    ◄──────────────── QUEUE adds data here
```
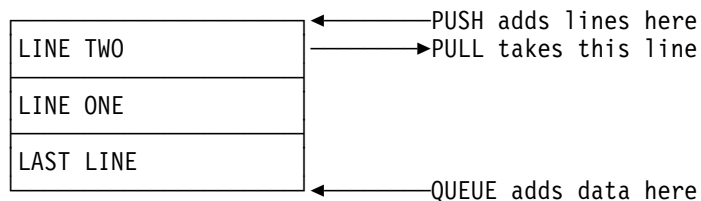
Figure 3. External data queue after RMVREXBUF

Now, if instead of issuing a RMVREXBUF command after getting the queue into the state shown in Figure 2, you were to run the following set of REXX instructions:

```
 PULL line1    /* line1 = 'NEWBUF PUSH'   */
 PULL line2    /* line2 = 'NEWBUF FIRSTQ' */
 PULL line3    /* line3 = 'NEWBUF SECONDQ' */
 PULL line4    /* line4 = 'LINE THREE'     */
```

The queue would look the same as in the Figure 3. Buffer boundaries created with the ADDREXBUF command are ignored by the PULL instruction.

Specifying a buffer number on the RMVREXBUF command causes existing buffers created after the one whose number is specified to be destroyed. Thus, this command can be used to "cleanup" after a particularly unruly user of the REXX external data queue. Also, if you want to totally erase the queue, specify the *ALL value on the buffer parameter of the RMVREXBUF command. This will cause the queue to be cleared. In addition, if the queue is damaged, this command can be used to delete and re-create the queue.

The following example shows how to use ADDREXBUF and RMVREXBUF:

```
/* REXX program which demonstrate how to use the ADDREXBUF and    */
/* RMVREXBUF commands with other REXX queuing services.           */
bufn = 0
'ADDREXBUF &BUFN'                  /* Add buffer '1' to the queue. */
PUSH 'Buffer' bufn ': Line 1 '     /* Add a line to the queue.     */
PUSH 'Buffer' bufn ': Line 2 '     /* Add a line to the queue.     */
PUSH 'Buffer' bufn ': Line 3 '     /* Add a line to the queue.     */

'ADDREXBUF &BUFN'                  /* Add buffer '2' to the queue. */

PUSH 'Buffer' bufn ': Line 1 '     /* Add a line to the queue.     */
PUSH 'Buffer' bufn ': Line 2 '     /* Add a line to the queue.     */

PULL entry                         /* Pull a line from the queue.  */
SAY entry                          /* Displays 'Buffer 2: line 2'. */

'RMVREXBUF &BUFN'                  /* Remove buffer '2'            */
                                   /* from the queue.              */

PULL entry                         /* Pull a line from the queue.  */
SAY entry                          /* Displays 'Buffer 1: line 3'. */

'ADDREXBUF &BUFN'                  /* Add buffer '2' to the queue. */
PUSH 'Buffer' bufn ': Line 1 '     /* Add a line to the queue.     */

'ADDREXBUF '                       /* Add buffer '3',              */
                                   /* do not save bufn.            */
PUSH 'Buffer' bufn + 1 ': Line 1 ' /* Add a line to the queue.     */

'RMVREXBUF &BUFN'                  /* Remove buffer '2' from queue.*/
                                   /* Will remove '2' and '3'.     */

PULL entry                         /* Pull a line from the queue.  */
SAY entry                          /* Displays 'Buffer 1: line 2'. */

'ADDREXBUF &BUFN'                  /* Add buffer '2' to the queue. */
PUSH 'Buffer' bufn ': Line 1 '     /* Add a line to the queue.     */

'RMVREXBUF *CURRENT'               /* Remove buffer '2'            */
                                   /* from the queue.              */

 /* At this point, buffer 1 has 1 entry.                           */

'ADDREXBUF &BUFN'                  /* Add buffer '2' to the queue. */
PUSH 'Buffer' bufn ': Line 1 '     /* Add a line to the queue.     */
```

```
        'ADDREXBUF &BUFN'                    /* Add buffer '3' to the queue. */
        PUSH 'Buffer' bufn ': Line 1 '       /* Add a line to the queue.     */
        PUSH 'Buffer' bufn ': Line 2 '       /* Add a line to the queue.     */

        SAY QUEUED()                         /* Displays '4', total of all   */
                                             /* entries for all buffers in   */
                                             /* the queue.                   */

        PULL entry                           /* Pull a line from the queue.  */
        SAY entry                            /* Displays 'Buffer 3: line 2'. */

        PULL entry                           /* Pull a line from the queue.  */
        SAY entry                            /* Displays 'Buffer 3: line 1'. */

        PULL entry                           /* Pull a line from the queue.  */
        SAY entry                            /* Displays 'Buffer 2: line 1'. */

        PULL entry                           /* Pull a line from the queue.  */
        SAY entry                            /* Displays 'Buffer 1: line 1'. */

        /* Because the queue is empty,                                       */
        /* any further PULLs will pull from STDIN, not the queue.            */

        PULL entry                    /* Pull a line from STDIN.      */
        SAY entry                     /* Displays your entry.         */

        'RMVREXBUF *ALL'        /* Clears all entries from the queue. */

        RETURN
```

Chapter 10. Determining Problems with REXX Programs

REXX provides several tools for determining where problems are within the REXX program. In this chapter the following topics are covered:

- Using the TRACE Instruction and the TRACE Function
- Using the Trace REXX (TRCREX) Command.

For more information on other REXX tools for identifying problems within a REXX program, see the *REXX/400 Reference*.

Using the TRACE Instruction and the TRACE Function

The TRACE instruction provides you with a powerful tool for determining errors. TRACE lets you see how expressions are being evaluated while the program is actually running.

The TRACE instruction can be placed anywhere within a REXX program where you want more specific information about how the program is being interpreted.

When a TRACE instruction is being interpreted, the first letter of the second word determines the type of tracing, and the remainder of the second word is ignored.

The following are the most commonly used trace settings. For more information on tracing and the other trace settings, see the *REXX/400 Reference*.

Trace Intermediates As each expression is evaluated, the result of each operation, the Intermediate results, is shown on the display.

Trace Results When each expression has been evaluated, the final result is shown on the display.

Trace Normal Only commands that would raise the failure condition are displayed, along with their return code. This is the default setting.

Trace Errors Commands that would raise an error or failure are displayed, along with their return code.

The TRACE function returns the current trace settings, when used without arguments. When an argument is specified, the trace setting is changed to the specified argument, and the previous setting is returned.

In order for TRACE results to be shown on the display, the character identifier (CHRID) must match the character set and code page (CCSID) of the REXX source file member being traced.

Using Interactive Tracing

Interactive tracing is turned on and off by placing a question mark immediately in front of the TRACE setting. For example, TRACE ?R will provide interactive tracing of results. Interactive tracing lets you examine each clause, one at a time, and continue interpretation by pressing the Enter key. Interactive tracing causes the program to pause after it interprets most instructions that result in trace output. The exceptions include SIGNAL, CALL, and reiterations of DO loops.

If interactive tracing is turned on and another TRACE instruction is encountered, the new trace setting is ignored. Unlike the TRACE instruction, the TRACE function will change the current trace setting when it is encountered during interactive tracing.

When interactive tracing is turned on, you are prevented from overriding STDIN.

Note: Do not turn interactive tracing on for REXX programs which override STDIN.

Using Trace Settings

The following example lets you experiment with different trace settings. By entering the particular trace setting you are interested in, you can see how tracing works differently with each option.

```
/* This program performs a variety of operations.  The different   */
/* trace options are used in this program to show how they differ. */

SAY 'Enter trace option to use'
PULL traceopt
TRACE VALUE traceopt
one = 1
SAY 'The maximum of 1 and -2 is' max(one,-2)
"SNDMSG('Hello') TOUSR(*WRONG)"
```

Trace Output for the TRACE Intermediates Setting: If you choose the intermediates option when you run this program, the trace output looks like the following:

```
    i
       5 *-*     one = 1;
         >L>       "1"
       6 *-*     Say 'The maximum of 1 and -2 is' MAX(one, - 2);
         >L>       "The maximum of 1 and -2 is"
         >V>       "1"
         >L>       "2"
         >P>       "-2"
         >F>       "1"
         >O>       "The maximum of 1 and -2 is 1"
The maximum of 1 and -2 is 1
       7 *-*     'SNDMSG('Hello') TOUSR(*WRONG)';
         >L>       "SNDMSG('Hello') TOUSR(*WRONG)"
         +++       RC(CPF0001)
Press ENTER to end terminal session.




    ==>  _____
         _____
    3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
    18=Bottom  F19=Left   F20=Right F21=User Window
```

Trace Output for the TRACE Results Setting: If you choose the results option,
the trace output looks like the following:

```
Enter trace option to use
r
    5 *-*     one = 1;
      >>>        "1"
    6 *-*     Say 'The maximum of 1 and -2 is' MAX(one, - 2);
      >>>        "The maximum of 1 and -2 is 1"
The maximum of 1 and -2 is 1
    7 *-*     'SNDMSG('Hello') TOUSR(*WRONG)';
      >>>        "SNDMSG('Hello') TOUSR(*WRONG)"
      +++        RC(CPF0001)
Press ENTER to end terminal session.
End of terminal session.







==>  _____
    _____
3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
18=Bottom  F19=Left    F20=Right F21=User Window
```

Trace Output for the TRACE Normal Setting: If you choose the normal option,
the trace output looks like the following:

```
Enter trace option to use
n
The maximum of 1 and -2 is 1
    7 *-*     'SNDMSG('Hello') TOUSR(*WRONG)';
      +++        RC(CPF0001)
Press ENTER to end terminal session.











==>  _____
    _____
3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
18=Bottom  F19=Left    F20=Right F21=User Window
```

Trace Output for the TRACE Errors Setting: If you choose the errors option, the
trace output looks like the following:

```
   Enter trace option to use
   e
   The maximum of 1 and -2 is 1
        7 *-*     'SNDMSG('Hello') TOUSR(*WRONG)';
          +++        RC(CPF0001)
   Press ENTER to end terminal session.
```

```
   ==>  _____
        _____
   3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
   18=Bottom  F19=Left   F20=Right F21=User Window
```

Using Trace Settings in Subroutines

When tracing is changed in an internal routine, the previous trace setting is
restored when the internal routine returns to the main program. This lets you easily
turn off tracing at the top of your working subroutines while you are still debugging
other parts of your program, as shown in the following example.

```
TRACE I
SAY 'This program calculates 10 factorial divided by 9 factorial'
SAY '(And it does it the hard way.)'
answer = fact(10)/ fact(9)
SAY 'The answer is' answer
EXIT

fact: PROCEDURE
CALL TRACE O
ARG number
answer = 1
DO i = 1 TO number
   answer = answer * i
END
RETURN answer
```

When you run this example, the following will be displayed:

Trace Output

```
    2 *-*     SAY 'This program calculates 10 factorial divided by 9 factorial';
     >L>         "This program calculates 10 factorial divided by 9 factorial"
This program calculates 10 factorial divided by 9 factorial
    3 *-*     SAY '(And it does it the hard way.)';
     >L>         "(And it does it the hard way.)"
(And it does it the hard way.)
    4 *-*       answer = fact(10) / fact(9);
     >L>         "10"
    8 *-*       fact:
    8 *-*       Procedure;
    9 *-*       CALL TRACE  O;
     >L>          "0"
     >F>        "3628800"
     >L>          "9"
    8 *-*       fact:
    8 *-*       Procedure;
    9 *-*       CALL TRACE  O;
     >L>          "0"
     >F>        "362880"
     >O>         "10"
    5 *-*     SAY 'The answer is' answer;
     >L>         "The answer is"
     >V>         "10"
     >O>         "The answer is 10"
The answer is 10
    6 *-*     EXIT;
```

Interactive Tracing Example: The following example uses interactive tracing to recover from a command error.

```
/* This program has an error in it.  Look below to see how          */
/* interactive tracing can be used to recover from the error.       */


TRACE ?E            /* Set tracing to pause after error commands.    */
SAY 'Tracing will turn on when a command has an error'
user = '*SYSORP'    /* This is misspelled                            */
"SNDMSG MSG('Hello') TOUSR("user")"
SAY 'Now we are past the error'
"SNDMSG MSG('Hello again') TOUSR("user")"
```

Trace Output: When this program is run the error occurs. It is interactively corrected, `user = '*SYSOPR'`. The = sign causes the last instruction, which is the command, to be run again. With the assignment changed, the program runs.

```
Tracing will turn on when a command has an error
    5 *-*     'SNDMSG MSG('Hello') TOUSR('user')';
      +++       RC(CPF0001)
      +++    Interactive trace. "Trace Off" to end debug ENTER to Continue.
user = '*SYSOPR'
=
Now we are past the error
Press ENTER to end terminal session.




 

 

 



==>  _____
   _____
 3=Exit F4=End of File F6=Print F9=Retrieve F17=Top
 18=Bottom  F19=Left    F20=Right F21=User Window
```

Interpreting Trace Results

Output from the TRACE instruction is always written to STDERR. In addition, it is placed in the job log as command (CMD) messages. When you are running a job interactively, all input is also placed in the job log. If you are running in batch mode and the REXX program finds an interactive trace setting, the interactive setting is ignored. For example, if a REXX program issues the instruction TRACE ?I while in batch mode, it is treated as if the instruction was TRACE I.

TRACE Symbols

Output from the TRACE instruction shows a listing with lines prefixed by various symbols. Each symbol identifies actions taken within the program. These symbols can be used to differentiate REXX command (CMD) messages from other command messages in the job log. User input during interactive tracing is placed in the job log with no prefixes.

-   This identifies the source of a single clause, that is, the data actually in the program.

+++   This identifies a trace message. This may be the nonzero return code from a command, the prompt message when interactive debug is entered, an indication of a syntax error when in interactive debug, or the traceback clauses after a syntax error in the program.

>>>   This identifies the result of an expression (for TRACE R) or the value assigned to a variable during parsing, or the value returned from a subroutine call.

>.>   This identifies the value assigned to a placeholder during parsing

If you are using TRACE Intermediates (TRACE I), the following symbols are also used.

>C> The data traced is the name of a compound variable, traced after substitution and before use, provided that the name had the value of a variable substituted into it.

>F> The data traced is the result of a function call.

>L> The data traced is a literal (string, uninitialized variable, or constant symbol).

>O> The data traced is the result of an operation on two terms.

>P> The data traced is the result of a prefix operation.

>V> The data traced is the contents of a variable.

Using the Trace REXX (TRCREX) Command

The TRACE instruction is used within a REXX program. The Trace REXX (TRCREX) command lets you set interactive tracing of a REXX program from outside that program. See the *CL Reference* for the complete syntax of this command.

TRCREX sets the trace option which is in effect when interpretation of a REXX program begins. The TRCREX setting remains in effect for all REXX programs until it is changed by issuing another TRCREX command or until the job ends.

Since TRCREX turns on interactive tracing of a program, and TRACE instructions in REXX programs are not run during interactive tracing, TRACE instructions may be ignored while a TRCREX setting is in effect. The TRACE function in a program can override any trace setting.

Changing the trace setting in a program will not affect the TRCREX setting. If you run a program with a TRCREX setting in effect, change the TRACE setting in it, and then run another REXX program, the TRCREX setting will be in effect at the start of the second program.

Chapter 11. Understanding Condition Trapping

Conditions are a set of problems which might occur while running your REXX program. In REXX/400, condition traps can be set within a REXX program to handle ERROR, FAILURE, NOVALUE, HALT, or SYNTAX conditions. This chapter summarizes the key information you need to know to understand conditions and use condition traps. Further detail is provided in the *REXX/400 Reference*. To help you understand how to use condition traps within a REXX program, this chapter covers:

- Defining Conditions
- Defining Condition Traps
- Using Condition Trapping.

Defining Conditions

Conditions are problems or other occurrences which may arise while your REXX program is running. The following conditions are recognized by REXX and can be controlled by setting condition traps.

ERROR
The ERROR condition arises whenever a REXX program issues a command that results in an error being indicated by the command environment. For the CL command environment (COMMAND), this means any command that causes an escape message (or status or notify messages identified by the SETMSGRC built-in function) to be sent to the interpreter, except for those which cause the FAILURE condition, as noted below. Other command environments can have their own rules when they want to raise the error condition. They indicate an error description as specified for the command interface (see the user-defined command environment discussion in the *REXX/400 Reference*). In all cases, command failures will raise the ERROR condition when a FAILURE condition trap is not set.

FAILURE
The FAILURE arises whenever a REXX program issues a command that results in a failure being indicated by the command environment. For the CL command environment (COMMAND), this means any command that causes escape message CPF0001 or CPF9999 to be sent to the interpreter. For all other command environments, a failure condition results any time: (1) an escape message is received by the interpreter from the command environment; (2) when the command environment could not be found or you were not authorized to the command environment; or (3) when the command environment indicates an error condition as specified for the command interface (see the user-defined command environment discussion in the *REXX/400 Reference*). In all cases, command failures will raise the ERROR condition when a FAILURE condition trap is not set.

NOVALUE
The NOVALUE condition arises whenever a REXX program tries to use the value of a variable that has not been given a value. If no trap for this condition is set, then the value used

for the variable is the same as the name of the variable with all the lowercase characters translated to uppercase.

HALT   The HALT condition arises when a REXX exit program indicates that the HALT condition is to be raised. Unless an exit program is specified on the call of the interpreter for the RXHLT function code, there is no way to raise this condition. For more information on the RXHLT exit, see the *REXX/400 Reference*. If this condition is not trapped, the program ends with an error.

SYNTAX   The SYNTAX condition arises whenever the REXX interpreter finds a serious error while running a REXX program. This may be a syntax error in the program, an error in a system service called by the interpreter, or an error found within the interpreter itself. If this condition is not trapped, the program ends immediately.

Defining Condition Traps

A REXX program will be made aware that a condition has occurred if the program is using a condition trap for that condition. A condition trap is a routine within a REXX program to which control is given when the related condition occurs. A condition trap must be enabled before REXX will call it. Routines are discussed in more detail in Chapter 8, "Using REXX Functions and Subroutines" on page 99. A condition trap is enabled by using the SIGNAL ON or CALL ON instructions. The instruction that enables a condition trap must identify the condition the trap is to be used for and a REXX label. The label identifies the point in the program that control is to be given to when the specified condition occurs.

If the condition trap is enabled by using the SIGNAL ON instruction, a branch will be made to the label that was identified. In this case, the condition trap is not considered to be an internal routine and, therefore, is not expected to make a return. If the condition trap is enabled by using a CALL ON, the condition trap is considered an internal routine. The condition trap is expected to return, at which time control returns to the point following the command that had the error.

When the condition trap gains control, there will be several pieces of information available that can be used to analyze the error:

- The variable RC will contain the return code that the command environment returned.

- A message will be on the message queue, from which it can be received, if the command environment sent a message.

- The special variable SIGL will contain the line number of the clause that was being run when the condition was raised.

- The CONDITION function can return the text of the failing command, the current trapped condition, and whether the condition was raised by a CALL or a SIGNAL instruction.

Using a condition trap is optional. The actions taken if you do not use a trap is described in "Defining Conditions" on page 131. Moreover, after a condition trap is enabled, it can be later disabled. When a condition trap does not exist or is

disabled, the REXX program will continue to run, starting with the next clause that follows the command in error.

Using the CONDITION Built-in Function

The condition built-in function returns information associated with a condition when that condition is trapped. You can specify several options to retrieve different information about the condition:

Option	Information
C	Name of the trapped condition (SYNTAX, NOVALUE, ERROR, FAILURE, or HALT)
D	Descriptive string. This value depends on which condition is trapped.
	SYNTAX Returns the error number
	NOVALUE Returns the name of the variable
	ERROR Returns the command string
	FAILURE Returns the command string
	HALT Returns an optional informational string provided by the exit program which raised the condition
I	Returns the instruction which set the condition (CALL or SIGNAL). This is the default.
S	Returns the status of the condition (ON, OFF, or DELAYED).

For more information on ON, OFF, and DELAYED states, see the *REXX/400 Reference*.

Using Condition Trapping

You can trap three types of errors: errors in syntax, errors in using variables, and errors in commands.

Trapping Syntax Errors

Errors in REXX syntax can be trapped using the SYNTAX condition. The following program provides a syntax checker. You can enter any REXX instruction or a command, and the condition trap will run if you have an error in the syntax.

```
/* This program uses a syntax handler to recover from syntax errors.  */
/* The REXX error number is passed to the variable RC.                */
restart:
Signal on Syntax

DO UNTIL input = ''
  SAY 'Enter a REXX statement or command to run, or null line to exit'
  PULL input
  INTERPRET input
  END

EXIT

Syntax:
SAY 'You had an error in that instruction.'
SAY 'The error was number' RC 'which means' ERRORTEXT(RC)
```

```
SAY 'The instruction was' SOURCELINE(SIGL)
Signal restart
```

The SYNTAX condition is also raised for other serious errors that may have nothing to do with program syntax. The following are examples of situations where the SYNTAX condition would be raised:

- A REXX exit program indicates that it has encountered a serious error, which results in REXX error 48, "Failure in system service."

- A non-REXX external function ends with an escape message, which results in REXX error 40, "Incorrect call to routine."

- The interpreter finds an internal error, which results in REXX error 49, "Interpretation error."

- An attempt was made to divide by zero, producing REXX error 42, "Arithmetic overflow/underflow."

In general, when any REXX error occurs (an error identified by number), the SYNTAX condition is raised. The only exception to this is REXX error 4, "Program interrupted", which raises the HALT condition.

Trapping Errors in Using Variables

In REXX, you do not declare variables. When you use a variable without assigning a value to it, its name, in uppercase, is used as its value. For example, `SAY hello` produces `HELLO` as its output. This means you can create new variables without noticing. To determine when variables are used before they are assigned a value, use a NOVALUE trap. The REXX special variable SIGL stores the line number in the REXX program where the error occurred, as shown in the following example:

```
Signal on Novalue
message = The current time is TIME()
"SNDMSG MSG('"message"') TOUSR(*SYSOPR)"
EXIT

novalue:
SAY 'A variable was referenced before being used in line number' SIGL
SAY 'The variable name was' CONDITION('D')
SAY 'The instruction was' SOURCELINE(SIGL)
EXIT
```

The VALUE function returns or sets the value of a variable whose name may be a string expression. VALUE does not trigger a NOVALUE condition, so it can be used to check the contents of variables without triggering a NOVALUE trap.

Trapping Errors in Commands

Errors resulting from commands are trapped by using the ERROR condition. This condition is also discussed in "Understanding the Error and Failure Conditions" on page 85.

Example 1: This example shows a REXX program which tries to send a
command to the CL command environment. The error is trapped and processing
continues at the ERROR: routine. The line number where the error occurred is
placed in SIGL, and a message is displayed.

```
/* Signal on Error traps system commands that cause errors.         */

Signal on Error

"SNDMSG MSG('This message is missing a quote) TOUSR(*SYSOPR)"
EXIT

Error: SAY 'The command on line' SIGL 'had an error'
SAY 'The error return code on that command is' RC
EXIT
```

Example 2: This example of trapping command errors adds the SOURCELINE
built-in function. Now, not only can you discover the line in the REXX program
where the error occurred, you can use SIGL to determine the actual command
string, and possibly recover from the error. This program uses the CALL
instruction, rather than the SIGNAL instruction so a new command can be entered,
and the REXX program will continue to run.

```
/* Signal on Error traps system commands that cause errors.         */
/* This program shows one option for recovering from errors.        */

Call on Error       /* Using CALL means we can return after the trap. */

"SNDMSG MSG('This message is missing a quote) TOUSR(*SYSOPR)"
SAY 'This is the instruction after the command'
EXIT

Error: SAY 'The command on line' SIGL 'had an error'
SAY 'The error return code on that command is' RC
SAY 'The line from the program was:' SOURCELINE(SIGL)
SAY 'Enter the correct command, or null line to exit'
PARSE PULL newcommand
IF newcommand = '' THEN EXIT
   newcommand
RETURN
```

Example 3: Different traps may be used for the same condition at different points
in the program. The following example shows an outline of how errors might be
handled differently, when some commands are necessary to the program, and
others are less important.

```
Signal on Error Name setuperr

/* This is the setup part of the program where the program should    */
/* immediately stop without doing anything else if command errors    */
/* occurs.                                                            */
.
.
.

Signal on Error Name runerr
/* This is the main part of the program where the program should also*/
/* immediately stop without doing anything else if command errors    */
```

```
/* occurs.  Any cleanup from setup is done.                          */
.
.
.

Call on Error Name cleanuperr
/* At the end of the program, some commands are issued to "clean up" */
/* after the program.  We assume that these commands do not depend   */
/* on each other, and that even if one has an error, the other should*/
/* still be run.                                                     */
.
.
.

Exit

/**********************************************************************/
Setuperr:
Say 'An error occurred during program setup.  The program is stopping.'
Exit

Runerr:
Say 'An error occurred in the main part of the program.'
Say 'The program is stopping.'
Exit

Cleanuperr:
Say 'An error occurred during program clean-up.  The program is continuing.'
Return              /* Goes back to next instruction of program.      */
```

Trapping Multiple Conditions

You can trap several conditions within a REXX program, and change the traps as the program progresses. The following example shows a REXX program which monitors for both ERROR and NOVALUE conditions:

```
Call on Error
Signal on Novalue
'BADCMD'
SAY hello
EXIT                /* Normal exit */

Error:
SAY 'An error occurred on a command in line number' SIGL
SAY 'The command that caused the error was' CONDITION('D')
SAY 'The command set a return code of' RC
SAY 'Enter 1 if you want the program to continue.'
SAY 'Enter anything else to quit now'
PULL answer
IF answer = '1' THEN RETURN
   ELSE EXIT

Novalue:
SAY 'The variable' CONDITION('D') 'was used in line number' SIGL
SAY 'before any value was assigned to it.  Program stopping.'
EXIT
```

Appendix A. REXX Keywords

This list shows the words which REXX interprets as REXX keyword instructions.

ADDRESS

ARG

CALL

DO

DROP

EXIT

IF

INTERPRET

ITERATE

LEAVE

NOP

NUMERIC

OPTIONS

PARSE

PROCEDURE

PULL

PUSH

QUEUE

RETURN

SAY

SELECT

SIGNAL

TRACE

Appendix B. REXX Built-in Functions

The following is a list of the REXX built-in functions:

ABBREV (Abbreviation)
ABS (Absolute Value)
ADDRESS
ARG (Argument)
BITAND (Bit by Bit AND)
BITOR (Bit by Bit OR)
BITXOR (Bit by Bit Exclusive OR)
B2X (Binary to Hexadecimal)
CENTER/CENTRE
COMPARE
CONDITION
COPIES
C2D (Character to Decimal)
C2X (Character to Hexadecimal)
DATATYPE
DATE
DBCS (Double-Byte Character Set) functions. See Appendix C, "Double-Byte
Character Set Support" on page 141, for more information.
DELSTR (Delete String)
DELWORD (Delete Word)
DIGITS
D2C (Decimal to Character)
D2X (Decimal to Hexadecimal)
ERRORTEXT
FORM
FORMAT
FUZZ
INSERT
LASTPOS (Last Position)
LEFT
LENGTH
MAX (Maximum)
MIN (Minimum)
OVERLAY
POS (Position)
QUEUED
RANDOM
REVERSE
RIGHT
SETMSGRC (Set Message Return Code)
SIGN
SOURCELINE
SPACE
STRIP
SUBSTR (Substring)
SUBWORD
SYMBOL
TIME
TRACE

TRANSLATE
TRUNC (Truncate)
VALUE
VERIFY
WORD
WORDINDEX
WORDLENGTH
WORDPOS (Word Position)
WORDS
XRANGE (Hexadecimal Range)
X2B (Hexadecimal to Binary)
X2C (Hexadecimal to Character)
X2D (Hexadecimal to Decimal)

Appendix C. Double-Byte Character Set Support

Double-byte character sets (DBCS) support languages that have more characters than can be represented by eight bits, such as Japanese Kanji and Korean Hangeul. REXX supports DBCS by using:

1. DBCS functions that specifically support the processing of DBCS character strings:

   DBADJUST
   DBBRACKET
   DBCENTER
   DBLEFT
   DBRIGHT
   DBRLEFT
   DBRRIGHT
   DBTODBCS
   DBTOSBCS
   DBUNBRACKET
   DBVALIDATE
   DBWIDTH

2. The OPTIONS instruction, which controls how REXX evaluates DBCS data. This instruction uses two options:

   - ETMODE which supports DBCS literal strings

   - EXMODE which provides full logical character support to enable data operations

The effect of OPTIONS 'ETMODE' depends on the CCSID of the REXX source file. For SBCS CCSIDs (those without a DBCS component), OPTIONS 'ETMODE' is coded when literal strings or comments containing DBCS characters are to be checked for being valid DBCS strings. This is useful to prevent, for example, premature termination of a comment by the occurrence of the '*/' code points within a double-byte character string.

For DBCS CCSIDs (those with a DBCS component), OPTIONS 'ETMODE' is automatically in effect for the REXX source file, and should therefore not be coded in the REXX procedure.

For more information on understanding the difference between SBCS CCSIDs and DBCS CCSIDs, see the *National Language Support*.

Since each DBCS character consists of two bytes, REXX distinguishes DBCS data from single-byte data by the presence of shift-out (SO (X'0E')) and shift-in (SI (X'0F')) bracket characters. The SO and SI characters are the delimiters for DBCS text.

When a DBCS string is written to STDOUT, the string will be segmented and each segment will have its own SO and SI pair, at the beginning of each line. The length of each segment will be shortened to take in account the SO and SI characters.

Notational conventions

This book uses the following notational conventions:

```
DBCS character          ->    .A .B .C .D
SBCS character          ->    a b c d e
DBCS blank              ->    '. '
EBCDIC shift-out (X'0E') ->    <
EBCDIC shift-in  (X'0F') ->    >
```

Note: In EBCDIC, the shift-out (SO) and shift-in (SI) characters distinguish DBCS characters from SBCS characters.

For more details about the instructions and functions that support the processing of DBCS character strings (used as data) and symbols, see the *REXX/400 Reference*.

Appendix D. Operators and Order of Operations

Operators

Table 1. String Concatenation

Operator	Operation
(blank)	Concatenate terms with one blank in between
||	Concatenate terms without a blank (force abuttal)
(abuttal)	Concatenate without an intervening blank

Table 2. Arithmetic Operators

Operator	Operation
+	Add
-	Subtract
*	Multiply
/	Divide
%	Divide and return the integer part of the result
//	Divide and return the remainder (not modulo, because the result may be negative)
**	Power (raise a number to a whole-number power)
Prefix -	Negate the following term. Same as '0-term'
Prefix +	Take following term as if it was '0+term'

Table 3. Logical Operators

Operator	Operation	Result
&	AND	Returns 1 if both are true.
|	Inclusive OR	Returns 1 if either is true.
&&	Exclusive OR	Returns 1 if either (but not both) is true.
Prefix \	Logical NOT	Negates. 1 becomes 0 and vice-versa
Prefix ¬	Logical NOT	Negates. 1 becomes 0 and vice-versa

Note: The symbols \ and ¬ are synonymous. Either may be used as a <u>not</u> symbol. Usage is a matter of availability or personal preference.

Table 4. Comparison Operators

Operator	Operation
==	True if terms are strictly equal (identical)
=	True if the terms are equal (numerically or when padded)
\ = =	True if the terms are NOT strictly equal
¬ = =	True if the terms are NOT strictly equal
\ =	Not equal (inverse of =)
¬ =	Not equal (inverse of =)
>	Greater than
<	Less than
> >	Strictly greater than
< <	Strictly less than
> <	Greater than or less than (same as not equal)
< >	Less than or greater than (same as not equal)
> =	Greater than or equal to
\ <	Not less than
¬ <	Not less than
> > =	Strictly greater than or equal to
\ < <	Strictly NOT less than
¬ < <	Strictly NOT less than
< =	Less than or equal to
\ >	Not greater than
¬ >	Not greater than
< < =	Strictly less than or equal to
\ > >	Strictly NOT greater than
¬ > >	Strictly NOT greater than

Note: The symbols \ and ¬ are synonymous. Either may be used as a <u>not</u> symbol. Usage is a matter of availability or personal preference.

Order of Operations

Table 5. Precedence

Operation	Operators	Precedence
Prefix Operations	Prefix -	1
	Prefix +	
	Prefix \	
	Prefix ¬	
Exponentiation	* *	2
Multiplication and Division	*, /, %, //	3
Addition and Subtraction	+, -	4
Concatenation (with or without blanks)	" ", ||	5
Comparison Operations	= > <	6
	== >> <<	
	\= ¬=	
	> < < >	
	\> ¬>	
	\< ¬<	
	\== ¬==	
	\>> ¬>>	
	\<< ¬<<	
	>= >>=	
	<= <<=	
Logical And	&	7
Logical Or	| &&	8

Note: Expression evaluation is controlled by using parentheses, because expressions within parentheses are evaluated first, and by operator precedence. See the *REXX/400 Reference* for more information.

Appendix E. Sample REXX Programs

This appendix provides four longer sample REXX programs. These programs have been used in other SAA environments and may provide you with some additional insights about how REXX can be used.

REXXTRY: The REXXTRY program lets you sample what happens when you enter various REXX instructions. This program can be used as a learning tool.

```
/* REXXTRY: This program lets you interactively enter REXX        */
/* instructions.  If you run it with no parameter, or with a      */
/* question mark for a parameter, it will briefly describe itself. */

/* This program uses meaningless variable names because the user   */
/* may create or use variables with any name.  The names used in   */
/* this program are "unusual", so that there is not much chance that*/
/* the user will change one.                                       */

  Parse Source . . fnyx .        /* Get the name of the program.  */
  Parse Arg fynz                 /* Get the argument string.      */

  If fynz='' | fynz='?' Then Do
    Say fnyx 'allows you to interactively execute REXX instructions -'
    Say 'each instruction string is executed when you press Enter.'
    End

  If fynz='?' Then Do
    Say 'You may also specify a REXX statement directly on the call for'
    Say 'immediate execution.'
    Exit
    End

  /* A border of periods is written out after each of the user's   */
  /* instructions.  The string of periods is created here.         */
  fzzs=copies('.',60)

  If fynz¬=''
   /* If an instruction was given as a parameter, it will be pushed */
   /* into the REXX queue.                                          */
   Then Push fynz
   Else Do
   /* No parameter was given, so the program prompts the user to    */
   /* enter instructions.                                           */
    Say 'Go on - try a few...'
    Say
    Say 'To end enter "EXIT".'
    End
  /* The user can set the variable 'trace' to a trace value.        */
  /* That value will be used to trace each of the user's            */
  /* instructions but not for the rest of the program.              */
  /* The initial trace value is 'Off'.                              */
  trace='Off'

/* The main action of the program starts here.                     */
top:
    Signal On Syntax
```

```
      Do iynz=1                      /* This is an infinite loop.     */
        last=lynz                    /* Available for user to retrieve.*/
        Parse Pull lynz              /* Get a line of input.          */
       Select
        When lynz=''
          Then Say fnyx': Enter EXIT to End.'

        Otherwise
         /* This section of the program handles one of the user's     */
         /* instructions.  The variable rc is set so that a change    */
         /* can be noticed, and the user's trace setting is set.      */
         /* Then the user's instruction is Interpreted.               */
         rc='X'
         Call setline; Trace (trace)
         Interpret lynz
         Trace 'Off'

         /* Now write out the border.  If rc has changed, the new     */
         /* value is shown.                                           */
         If rc='X'
           Then Say fzzs
             Else Say overlay('Rc ='rc' ',fzzs)
       End /* select */

       If fynz ¬= '' & queued()=0 Then Exit   /* Single-line request.*/
      End /* iynz */

/* That ends the main program.                                        */

/* This subroutine sets the variable traceline to the line number     */
/* of the line the subroutine was called from.                        */
Setline: traceline=sigl
  return result

/* The syntax routine gets called if a syntax error occurs.           */
/* It tells what the error was, and then uses SIGNAL to go back to    */
/* the main program.  Or it will EXIT if the program was being        */
/* used to run just a single instruction as a parameter.              */
Syntax: Trace Off;
  Say
  If sigl<=traceline Then /* The syntax error was caused by a bad     */
                       /* value in the trace variable.                */
    Do
    Say 'Invalid trace argument, set to "All".'
    trace='All'
    Signal Top
    End

  Say 'You had a Syntax error there (RC =' rc':' errortext(rc)')'
  Say 'Try it again.'
  If fynz¬='' Then Exit rc
    Else Signal Top

/* This subroutine is available for the user to test the CALL         */
/* instruction with.                                                  */
Testsub: Say 'In test subroutine'
  Say 'Returning...'
  return 1234
```

QT (Query Time): The QT (Query Time) program is a REXX classic.

```
/* Displays time in real English, also chimes.      */
/* Mike Cowlishaw,  December 1979 - December 1982   */

   Do; Parse source . . . . . EN .;arg  arg deb; End
   if arg=?
   then call tell
   if arg='PUSH' then do
     stack=1
     parse var deb arg deb
     end
    else stack=0
   if arg='TEST'
    then c8=deb
    else do
     if deb\='' then trace ?r
     c8=time()
     end
   ot='It''s'

   hr=substr(c8,1,2)+0
   mn=substr(c8,4,2)
   sc=substr(c8,7,2)

   h.1 = 'one' ; h.2 = 'two';   h.3 = 'three';   h.4 = 'four'
   h.5 = 'five'; h.6 = 'six';   h.7 = 'seven';   h.8 = 'eight'
   h.9 = 'nine'; h.10= 'ten';   h.11= 'eleven';  h.12= 'twelve'

   if sc>29 then mn=mn+1      /* Round up mins */
   if mn>32 then hr=hr+1      /* something to.. */

   mod=mn//5
   select
     when mod=0 then nop           /* Exact */
     when mod=1 then ot=ot 'just gone'
     when mod=2 then ot=ot 'just after'
     when mod=3 then ot=ot 'nearly'
     when mod=4 then ot=ot 'almost'
     end                     /* Select */

   mn=mn+2                   /* Round up */
   if hr//12=0 & mn//60<=4
    then signal midnoon      /* Noon and midnight */
   mn=mn-(mn//5)             /* to nearest 5 mins */
   if hr>12
    then hr=hr-12            /* Get rid of 24-hour clock */
    else
     if hr=0 then hr=12      /* Cater for midnight */

   select
     when mn=0  then nop         /* Add O'clock later */
     when mn=60 then mn=0
     when mn= 5 then ot=ot 'five past'
     when mn=10 then ot=ot 'ten past'
     when mn=15 then ot=ot 'a quarter past'
```

```
        when mn=20 then ot=ot 'twenty past'
        when mn=25 then ot=ot 'twenty-five past'
        when mn=30 then ot=ot 'half past'
        when mn=35 then ot=ot 'twenty-five to'
        when mn=40 then ot=ot 'twenty to'
        when mn=45 then ot=ot 'a quarter to'
        when mn=50 then ot=ot 'ten to'
        when mn=55 then ot=ot 'five to'
        end
    ot=ot h.hr
    if mn=0
    then ot=ot "o'clock"
    ot=ot'.'
    if \stack then do
      if mod=0 & mn//15=0
      then call chime
      say; say ot; say
      end
     else push ot
    exit

MIDNOON:
    if hr=12
     then ot=ot 'Noon.'
     else ot=ot 'Midnight.'
    if \stack
     then do
      hr=12
      if mn//60=2
       then do
        mn=0
        call chime
        end
      say; say ot; say
      end
     else push ot
    exit

CHIME:
    /* Give chimes */
    if mn//60=0
     then do
      chime='Bong'
      num=hr
      end
     else do          /* Quarter */
      chime='Ding-Dong'
      num=mn%15
      end
    say; say ot
    ot='('chime
    do num-1
      ot=ot||',' chime
      end
    ot=ot||'.)'
    return          /* Chime */
```

```
TELL:
   say
   say en 'will query the time and display or return it in English.'
   say 'Call without any parameter to display the time, or with "PUSH"'
   say '  to push the time-string onto the Stack.'
   say en 'will "chime" at the quarter-hours and on the hours, but the'
   say '  chimes are not placed on the stack.'
   say 'English (British) idioms are used in this program.'
   return
```

SAY: This program is a simple REXX example that evaluates the argument passed to it and displays the result. This program displays this description if called with a parameter of '?' or 'help'. If the expression is not a valid REXX expression, this program will issue a message saying so. You can add REXX instructions to the expression, by using a semicolon as a delimiter.

```
Parse Arg rest    /* Capture the argument (in mixed case).*/
/* If the argument is ? or help (in upper or lower case),*/
/* then display the prolog comment.                      */
If rest=? | translate(rest)='HELP' Then
   Do i = 1 Until Sourceline(i) = '*/'
   Say Sourceline(i)
   End
Else
   Do
   Signal on Syntax          /* Catch not valid expressions.*/
   Interpret 'I =' rest;     /* Evaluate the expression.  */
   Say i                     /* Display the result.       */
   End
Exit


Syntax:
 Say 'Sorry, that argument was not a valid REXX expression.'
 Say 'The error was:' Errortext(rc)    /* rc gets set to the error   */
 Exit                                  /* number.  Errortext returns */
                                       /* the error message for a    */
                                       /* given error number.        */
```

METRIC: This program converts metric measurements to imperial/US.

```
arg val unit
if val='' | val='?' then signal help

drop outmsg.

select

/* Temperature */

when unit='C' then do          /* Celsius to Fahrenheit */
   output = format(32 + (9 * val / 5),,1)
   outmsg.1 = val 'degrees Celsius equals' output 'Fahrenheit'
   end
when unit='F' then do          /* Fahrenheit to Celsius */
   output = format((5 * (val - 32) / 9),,1)
   outmsg.1 = val 'degrees Fahrenheit equals' output 'Celsius'
   end
```

```
                    /* Weight */

                    when unit='TON' then do      /* Tons to Metric tons and vice versa */
                       x = 453.59237
                       /* Imperial to metric */
                       tonne = val * 2240 * x / 1000000
                       outmsg.1 = val 'imperial tons equals' format(tonne,,3) 'metric tonnes'
                       tonne = val * 2000 * x / 1000000
                       outmsg.2 = val 'short tons equals' format(tonne,,3) 'metric tonnes'
                       /* Metric to imperial */
                       tons = val / (x * 2240 / 1000000)
                       outmsg.3 = val 'metric tonnes equals' format(tons,,3) 'imperial tons'
                       tons = val / (x * 2000 / 1000000)
                       outmsg.4 = val 'metric tonnes equals' format(tons,,3) 'short tons'
                       end
                    when unit='CWT' then do      /* Hundredweights to Kg */
                       x = 453.59237
                       kg = val * 112 * x / 1000
                       outmsg.1 = val 'hundredweight equals' format(kg,,3) 'Kg'
                       end
                    when unit='LB' then do       /* Pounds to Kg */
                       output = format((val * 0.45359237),,2)
                       outmsg.1 = val 'pounds equals' output 'Kg'
                       end
                    when unit='OZ' then do       /* Ounces to gm */
                       output = format((val /16 * 453.59237),,2)
                       outmsg.1 = val 'ounces equals' output 'gm'
                       end
                    when unit='KG' then do       /* Kilograms to pounds and ounces */
                       x = 0.45359237
                       oz =  16 * val / x
                       lbs = oz % 16
                       oz  = format((oz // 16),,1)
                       outmsg.1 = val 'Kg equals' lbs 'lb' oz 'oz'
                       end
                    when unit='GM' then do       /* Grams to (pounds and) ounces */
                       x = 453.59237
                       oz =  16 * val / x
                       if oz<16 then
                          outmsg.1 = val 'grams equals' format(oz,,2) 'ounces'
                       else do
                          lbs = oz % 16
                          oz  = format((oz // 16),,1)
                          outmsg.1 = val 'grams equals' lbs 'lb' oz 'oz'
                          end
                       end

                    /* Length */

                    when unit='M' then do        /* Metres to Yards and Miles to Km */
                       /* Metres to Yards, Feet, and Inches */
                       in = val / .0254
                       ft = in % 12
                       in = format((in // 12),,1)
                       yd = ft % 3
                       ft = ft // 3
                       outmsg.1 = val 'metres equals' yd 'yards,' ft 'feet,' in 'inches'
                       /* Miles to Km */
```

```
      outmsg.2 = val 'miles equals' format((val * 63360 * 0.0000254),,3) 'Km'
      end
   when unit='KM' then do        /* Km to miles */
      outmsg.1 = val 'Km equals' format((val / (63360 * 0.0000254)),,2) 'miles'
      end
   when unit='YD' then do        /* Yards to metres */
      outmsg.1 = val 'yards equals' format((val*36*0.0254),,3) 'metres'
      end
   when unit='FT' then do        /* Feet to metres */
      outmsg.1 = val 'feet equals' format((val*12*0.0254),,3) 'metres'
      end
   when unit='IN' then do        /* Inches to centimetres */
      outmsg.1 = val 'inches equals' format((val*2.54),,3) 'centimetres'
      end
   when unit='CM' then do        /* Centimetres to (feet and) inches */
      in = val/2.54
      if in<12 then
         outmsg.1 = val 'centimetres equals' format(in,,2) 'inches'
      else do
         ft = in % 12
         in = in // 12
         outmsg.1 = val 'centimetres equals' ft 'feet,' format(in,,2) 'inches'
         end
      end
   when unit='MM' then do        /* Millimetres to (feet and) inches */
      in = val/25.4
      if in<12 then
         outmsg.1 = val 'millimetres equals' format(in,,2) 'inches'
      else do
         ft = in % 12
         in = in // 12
         outmsg.1 = val 'millimetres equals' ft 'feet,' format(in,,2) 'inches'
         end
      end

/* Petrol consumption */
   when unit="MPG" then do              /* MPG to km/l and l/100km          */
      x = 63360 * 0.0000254
      y = val * x / 4.5461             /* Imperial gallon                  */
      outmsg.1 = val 'mpg (Imperial) equals' format(y,,1) 'km/litre,',
         format(100/y,,1) 'litres per 100 km'
      outmsg.2 = ' '
      y = val * x / 3.7853             /* US gallon                        */
      outmsg.3 = val 'mpg (USA) equals' format(y,,1) 'km/litre,',
         format(100/y,,1) 'litres per 100 km'
      end
   when unit="KM/L" then do             /* Km/litre to mpg           */
      x = 63360 * 0.0000254
      y = val / x * 4.5461             /* Imperial gallon               */
      z = val / x * 3.7853             /* US gallon                     */
      outmsg.1 = val 'km/litre equals' format(y,,1) 'mpg (Imperial),',
         format(z,,1) 'mpg (USA).'
      end
   when unit="L/100KM" then do          /* Litres/100 km to mpg          */
      x = 63360 * 0.0000254
      y = (100 / x) / (val / 4.5461)
      z = (100 / x) / (val / 3.7853)
      outmsg.1 = val 'litres/100 km equals' format(y,,1) 'mpg (Imperial),',
```

```
               format(z,,1) 'mpg (USA)'
            end

      /* Area */

      when unit='SQ.YD' then do    /* Square yards to square metres */
         x = (.0254 * 36) ** 2
         sqm = val * x
         outmsg.1 = val 'square yards equals' format(sqm,,3) 'square metres'
         end
      when unit='SQ.FT' then do    /* Square feet to square metres */
         x = (.0254 * 12) ** 2
         sqm = val * x
         outmsg.1 = val 'square feet equals' format(sqm,,3) 'square metres'
         end
      when unit='SQ.M' then do      /* Square metres AND square miles */
         /* From square metres */
         x = (.0254 * 36) ** 2
         sqyd = format((val / x),,2)
         x = (.0254 * 12) ** 2
         sqft = format((val / x),,2)
         outmsg.1 = val 'square metres equals' sqyd 'square yards,' sqft 'square feet'
         /* From square miles */
         x = (63360 * 0.0000254) ** 2
         sqkm = val * x
         outmsg.2 = val 'square miles equals' format(sqkm,,3) 'square kilometres'
         end
      when unit='SQ.KM' then do    /* Square kilometres to square miles */
         x = (63360 * 0.0000254) ** 2
         sqm = val / x
         outmsg.1 = val 'square kilometres equals' format(sqm,,3) 'square miles'
         end
      when unit='H' then do         /* Hectares to acres */
         x = (.0254 * 36) ** 2
         acre = val * 10000 / (4840 * x)
         outmsg.1 = val 'hectares equals' format(acre,,2) 'acres'
         end
      when unit='ACRE' then do     /* Acres to hectares */
         x = (.0254 * 36) ** 2
         sqm = val * x * 4840
         hec = sqm / 10000
         sqm = format(sqm,,0)
         hec = format(hec,,3)
         outmsg.1 = val 'acres equals' sqm 'square meters, or' hec 'hectares'
         end

      /* Volume */

      when unit='L' then do        /* Litres to appropriate measure */
         impg = val / 4.5461          /* Imperial gallons                */
         usg  = val / 3.7853          /* US gallons                      */
         if impg >= 1 then            /* Not less than 1 Imperial gallon */
            outmsg.1 = val 'litres equals' format(impg,,1) 'Imperial gallons.'
         else outmsg.1 = val 'litres equals' format(impg*8,,1) 'Imperial pints.'
         outmsg.2 = ' '
         if usg >= 1 then             /* Not less than 1 US gallon       */
            outmsg.3 = val 'litres equals' format(usg,,1) 'US gallons.'
         else outmsg.3 = val 'litres equals' format(usg*8,,1) 'US pints.'
```

```
         end
   when unit='G' then do          /* Gallons to litres */
      impg = val * 4.5461              /* Imperial gallons                   */
      usg  = val * 3.7853              /* US gallons                         */
      outmsg.1 = val 'Imperial gallons equals' format(impg,,3) 'litres,',
         val 'US gallons equals' format(usg,,3) 'liters.'
      end
   when unit='P' then do          /* Pints to litres */
      impg = val / 8 * 4.5461          /* Imperial gallons                   */
      usg  = val / 8 * 3.7853          /* US gallons                         */
      outmsg.1 = val 'Imperial pints equals' format(impg,,3) 'litres,',
         val 'US pints equals' format(usg,,3) 'liters.'
      end
   when unit="CU.IN" then do      /* Cubic inches to cc/litres */
      output = val * (2.54**3)   /* cc */
      if output < 1000 then
         outmsg.1 = val 'cu.in equals' format(output,,0) 'cc'
      else
         outmsg.1 = val 'cu.in equals' format(output/1000,,3) 'litres'
      end
   when unit="CU.YD" then do      /* Cubic yards to cubic metres */
      output = format(val*0.7646,,3)
      outmsg.1 = val 'cubic yards equals' output 'cubic metres'
      end
   when unit="M3" then do         /* Cubic metres to cubic yards */
      output = format(val*1.3080,,3)
      outmsg.1 = val 'cubic metres equals' output 'cubic yards'
      end
   /* Explain: */

   otherwise do
   say 'Unit of measure' unit 'not supported.'
   say
   help:
   say 'Format: METRIC value unit'
   say "  E.g. STRREXPRC METRIC PARM('10 G') to convert 10 gallons to litres."
   say
   say "Calculations are made using precise conversion factors,"
   say "but are rounded to a sensible number of decimal places."
   say
   say "Units of measure that are supported are:"
   say
   say "   C ......... Celsius to Fahrenheit"
   say "   F ......... Fahrenheit to Celsius"
   say "   TON ....... Tons to Metric tons and vice versa"
   say "   CWT ....... Hundredweights to Kg"
   say "   LB ........ Pounds to Kg"
   say "   OZ ........ Ounces to gm"
   say "   KG ........ Kilograms to pounds and ounces"
   say "   GM ........ Grams to (pounds and) ounces"
   say "   M ......... Metres to Yards and Miles to Km"
   say "   KM ........ Km to miles"
   say "   YD ........ Yards to metres"
   say "   FT ........ Feet to metres"
   say "   IN ........ Inches to centimetres"
   say "   CM ........ Centimetres to (feet and) inches"
   say "   MM ........ Millimetres to (feet and) inches"
   say "   MPG ....... MPG to km/l and l/100km"
```

```
            say "  KM/L ...... Km/litre to mpg"
            say "  L/100KM ... Litres/100 km to mpg"
            say "  SQ.YD ..... Square yards to square metres"
            say "  SQ.FT ..... Square feet to square metres"
            say "  SQ.M ...... Square metres AND square miles"
            say "  SQ.KM ..... Square kilometres to square miles"
            say "  H ......... Hectares to acres"
            say "  ACRE ...... Acres to hectares"
            say "  L ......... Litres to appropriate measure"
            say "  G ......... Gallons to litres"
            say "  P ......... Pints to litres"
            say "  CU.IN ..... Cubic inches to cc or litres"
            say "  CU.YD ..... Cubic yards to cubic metres"
            say "  M3 ........ Cubic metres to cubic yards"
         exit
         end
    end
    do i=1 to 3
       if outmsg.i='OUTMSG.'i then leave i
       say outmsg.i
       end
```

Appendix F. Sample REXX Programs for the AS/400 System

This appendix contains some examples of REXX programs which can be used on the AS/400 system.

Example 1: The P2D function converts a string containing packed numeric data into an unpacked string.

```
/*********************************************************************/
/*                                                                   */
/*  P2D - Converts a string containing packed numeric data           */
/*        into a string of the same value in an UNPACKED format.     */
/*                                                                   */
/*  Input - string containing valid PACKED numeric data              */
/*          decimal positions to adjust decimal point in UNPACKED    */
/*          string.                                                  */
/*                                                                   */
/*  Returns - string containing UNPACKED value of input string.      */
/*                                                                   */
/*********************************************************************/
parse arg pack_str , dec_place
unpack_str=C2X(pack_str)
dec_str=left(unpack_str,length(unpack_str)-1)
if datatype(dec_place)=='NUM' then
  do
    if dec_place > length(dec_str) then
      do
        say 'Invalid decimal Place'
        exit    /* Exit with no data gets error 44 in caller */
      end
    dec_str=insert('.',dec_str,length(dec_str)-dec_place)
  end
if pos(right(unpack_str,1),'BDE') > 0 then
  dec_str=-dec_str
exit dec_str
```

Example 2: The D2P function converts a numeric string to packed format.

```
/*********************************************************************/
/*                                                                   */
/*  D2P - Converts a string containing numeric data into a           */
/*        string containing the same value in PACKED format.         */
/*                                                                   */
/*  Input - string containing valid numeric data                     */
/*          decimal positions (for validation purposes)              */
/*                                                                   */
/*  Returns - string containing packed value of input string         */
/*                                                                   */
/*********************************************************************/
parse arg unpack_str , dec_place .
if datatype(unpack_str)¬='NUM' then
  do
    say 'Invalid input string, it must be a decimal string'
    exit    /* Exit with no data to force syntax error 44 in caller */
  end
```

```
                   /*  Delete decimal point.*/
       if pos('.',unpack_str) ¬= 0 then
         do
           if dec_place ¬= pos('.',unpack_str) then
             do
               say 'Decimal position passed to this routine does not match the',
                   'decimal number passed'
               say 'Conversion will continue'
             end
           unpack_str=delstr(unpack_str,pos('.',unpack_str),1)
         end

           /*  Check for negative number, if so remove it.*/
       if substr(unpack_str,1,1) = '-' then
         do
           minusflag=yes
           unpack_str=right(unpack_str,length(unpack_str)-1)
         end
       else  /* Check for PLUS sign, if so remove it.*/
         if substr(unpack_str,1,1) = '+' then
           unpack_str=right(unpack_str,length(unpack_str)-1)

           /* If the length of the number is even, insert a 0 in front.*/
       if length(unpack_str)//2=0 then
         unpack_str=insert('0',unpack_str,0)

           /* Also need to add a 0 to the end to convert to the sign.*/
       unpack_str=insert('0',unpack_str,length(unpack_str))

           /* Convert to character.*/
       pack_str=X2C(unpack_str)

       last_byte=substr(pack_str,length(pack_str))
       if minusflag=yes then
         or_str=substr(pack_str,1,length(pack_str)-1)||bitxor(last_byte,'0D'x)
       else
         or_str=substr(pack_str,1,length(pack_str)-1)||bitxor(last_byte,'0F'x)
       exit bitor(pack_str,or_str)
```

Example 3: The REXX program, which is the CPP for the following command,
copies a file member to the REXX external data queue.

```
             CMD        PROMPT('Copy file to REXX data queue')

             PARM       KWD(FROMFILE) TYPE(QUAL1) MIN(1) PROMPT('From +
                          file')

             PARM       KWD(MBR) TYPE(*NAME) LEN(10) MIN(1) +
                          PROMPT('Member')

             PARM       KWD(NMBRCDS) TYPE(*DEC) LEN(6) DFT(*ALL) +
                          SPCVAL((*ALL 999999)) PROMPT('Number of +
                          records to copy')

  QUAL1:     QUAL       TYPE(*NAME) LEN(10) MIN(1)

             QUAL       TYPE(*NAME) LEN(10) DFT(*LIBL) +
                          SPCVAL((*CURLIB) (*LIBL)) PROMPT('Library')
```

```
/********************************************************************/
/*  Command Processing REXX procedure for CPYFTOREXQ command.      */
/*   This program reads the contents of a file member and          */
/*   places it, line by line, on the REXX External Data Queue.     */
/*                                                                  */
/*   This example assumes the existence of a command called        */
/*   CPYFTOREXQ.  This is not an IBM-supplied command, but there   */
/*   are several approaches you could use to write a program which */
/*   would read a file and push the lines read into the REXX queue.*/
/*   You could then create your own CPYFTOREXQ command based on your*/
/*   program.  For examples of this type of program, see Appendix H.*/
/*                                                                  */
/*     Parameters passed:                                          */
/*                                                                  */
/*       From file name - lib     - Library that contains file     */
/*                       - file    - File that contains member     */
/*                                                                  */
/*       Member name    - mbr     - Member to be copied into queue */
/*                                                                  */
/*       Number of records - count - Number of records to be read  */
/*                                  from file                      */
/*                         *ALL  - Read all records                */
/*                                                                  */
/*    Returns:   '0' SUCCESS                                        */
/*               '1' FAIL                                           */
/*                                                                  */
/********************************************************************/

/* Parse out the library and file */
PARSE UPPER ARG 'FROMFILE(' lib '/' file ')'

/* Parse out the member */
PARSE UPPER ARG 'MBR(' mbr ')'

/* Parse out the number of records to copy */
PARSE UPPER ARG 'NMBRCDS(' count ')'

/* Check if file and member exist */
'CHKOBJ OBJ('lib'/'file') OBJTYPE(*FILE) MBR('mbr')'
IF rc ¬= '0' THEN DO
  /* Save the return code */
  save_rc = rc

  /* If there was a problem, let the user know and exit. */
  IF POS(rc,'CPF9801 CPF9810 CPF9815') ¬= 0 THEN DO
    msg = 'File' lib'/'file', member' mbr 'was not found.'
    'SNDPGMMSG MSG(&msg)'
  END
  ELSE DO
    msg = 'Unknown error' save_rc 'occurred during CHKOBJ processing.'
    'SNDPGMMSG MSG(&msg)'
  END
  EXIT 1
END

IF count = '*ALL' THEN count = '999999'
```

```
                    /* Override STDIN to the LIB/FILE parms. */
                    'OVRDBF FILE(STDIN) TOFILE('lib'/'file') MBR('mbr')'

                    DO count
                      /* read data from STDIN stream */
                      PARSE LINEIN data

                      /* If the data read is strictly equal to a null string, */
                      /* then we have reached the end of the file, so get out */
                      /* of this loop.                                        */
                      IF data == '' THEN LEAVE

                      /* Put data into REXX queue (FIFO order for PULL) */
                      QUEUE data
                    END

                    EXIT
                      PGM     PARM(&FILE_LIB &MBR &RECS)
                      DCL     VAR(&FILE_LIB) TYPE(*CHAR) LEN(20)
                      DCL     VAR(&FILE) TYPE(*CHAR) LEN(10)
                      DCL     VAR(&LIB) TYPE(*CHAR) LEN(10)
                      DCL     VAR(&MBR) TYPE(*CHAR) LEN(10)
                      DCL     VAR(&RECS) TYPE(*DEC) LEN(6)
                      DCL     VAR(&RECSA) TYPE(*CHAR) LEN(6)
                      DCL     VAR(&REXXPARM) TYPE(*CHAR) LEN(256)

                      CHGVAR &FILE %SST(&FILE_LIB 1 10)
                      CHGVAR &LIB  %SST(&FILE_LIB 11 10)
                      CHGVAR &RECSA &RECS

                      OVRDBF      FILE(STDIN) TOFILE(&LIB/&FILE) MBR(&MBR)

                      CHGVAR &REXXPARM VALUE('CPYFTOREXQ FROMFILE(' *TCAT +
                                           &LIB *TCAT '/' *TCAT &FILE +
                                           *TCAT ') NMBRCDS(' *TCAT    +
                                           &RECSA *TCAT ') MBR(' *TCAT +
                                           &MBR *TCAT ')')

                      /* Substitute your own library name for EXAMPLES in the */
                      /* next statement.  EXAMPLES is the name of the library */
                      /* that contains the previous REXX program.            */

                      STRREXPRC  SRCMBR(CPYFTOREXQ) SRCFILE(EXAMPLES/QREXSRC) +
                                 PARM(&REXXPARM)

                    ENDPGM
```

Example 4: The REXX program, which is the CPP for the following command, displays a profile of the contents of a library.

```
          CMD          PROMPT('Display Library Profile')
          PARM         KWD(LIBRARY) TYPE(*NAME) LEN(10) MIN(1) +
                          PROMPT('Library')
          PARM         KWD(FORMAT) TYPE(*CHAR) LEN(8) RSTD(*YES) +
                          DFT(*SUMMARY) VALUES(*DETAIL *SUMMARY) +
                          MIN(0) PROMPT('Output format')
          PARM         KWD(OUTPUT) TYPE(*CHAR) LEN(6) RSTD(*YES) +
                          DFT(*PRINT) VALUES(* *PRINT) +
                          PROMPT('Output Type')

/**********************************************************************/
/*  REXX program to DISPLAY a profile of the contents of a Library.  */
/*                                                                    */
/*    Parameters passed:                                              */
/*                                                                    */
/*       Library        - Libname  - Library name to search.         */
/*                                                                    */
/*       Output format  - *SUMMARY - Summarize output by object type.*/
/*                        *DETAIL  - Produce detail output.          */
/*                                                                    */
/*       Output type    - *PRINT   - Output to a listing.            */
/*                        - *      - Output to Display               */
/*                                                                    */
/*    Returns:   '0' SUCCESS                                          */
/*               '1' FAIL                                             */
/*                                                                    */
/**********************************************************************/

 SIGNAL on NOVALUE

 /* Setup ERROR,FAILURE & SYNTAX condition traps.*/
 SIGNAL ON SYNTAX

 /* Parse out the library value using the CDO "LIBRARY" keyword.*/
 PARSE ARG  'LIBRARY(' library ')'

 /* Parse out the output_format using the CDO "FORMAT" keyword.*/
 PARSE ARG  'FORMAT(' output_format ')'

 /* Parse out the output_type using the CDO "OUTPUT" keyword.*/
 PARSE ARG  'OUTPUT(' output_type ')'

 /* Does the library specified exist ? */
 'CHKOBJ OBJ(QSYS/'library') OBJTYPE(*LIB)'

 IF POS('CPF98',rc) ¬= 0 THEN DO
   'SNDPGMMSG  MSG( Library:'library 'not found)'
   EXIT(rc)
 END

 /* Set ERROR & FAILURE condition traps.*/
 CALL   ON ERROR name COMMAND_ERROR_TRAP
 CALL   ON FAILURE name command_error_trap

 /* Create temporary work file containing objects in library.*/
 'DSPOBJD OBJ('library'/*ALL) OBJTYPE(*ALL) ',
```

```
                  'DETAIL(*FULL) OUTPUT(*OUTFILE) ',
                  'OUTFILE(QTEMP/OBJD)'

       /* Setup SHARE(*YES) for OPNQRYF.*/
       'OVRDBF FILE(OBJD) TOFILE(QTEMP/OBJD) SHARE(*YES) SEQONLY(*NO)'

       /* Create ODP for work file containing records to be read.*/
       IF output_format = '*SUMMARY' THEN
          /* Sort by object type & object attribute.*/
          'OPNQRYF FILE((QTEMP/OBJD)) KEYFLD((ODOBTP) (ODOBAT))'
       ELSE
          /* Sort by object name.*/
          'OPNQRYF FILE((QTEMP/OBJD)) KEYFLD((ODOBNM))'

       X = QUEUED()     /* Save in X the number of entries in REXX QUEUE*/
                        /* prior to CPYFTOREXQ command.                 */


       'ADDREXBUF'

       /* Copy all records from the DSPOBJD outfile to the REXX QUEUE. */

       /* Use CPYFTOREXQ (Appendix example).                           */
       'CPYFTOREXQ FROMFILE(QTEMP/OBJD) NMBRCDS(*ALL) MBR(OBJD)'

       /* Close the ODP after records are copied to REXX QUEUE.        */
       'CLOF OBJD'
       /* Select output device - display (default) printer.           */
       /* Override output to spooled file & specify other attributes.  */
       IF output_type = '*PRINT' THEN DO
        'OVRPRTF    FILE(STDOUT) TOFILE(QSYSPRT) '
       END

       /* Initialize work values.*/
       y=''
       count = 0
       detail_line = '' COPIES(' ',80)
       prev_odobtp = ''; prev_odobat = ' ';
       total_objects = '0'
       total_size = '0'
       total_count = '0'
       total_odobsz ='0'
       page_rec_count = '0'
       pagenum = '0'
       total_count = '0'

       count = QUEUED() - X
       DO i=1 TO count
                   /* Do not PULL entries that were already on REXX QUEUE.*/
       /* Parse out the record from the QUEUE into the following  template. */
                /* From   Name    Length       Description           */

         PARSE PULL      .                    , /* Place holder            */
                     24    odobnm   +10  , /* Object name             */
                     34    odobtp   +8   , /* Object type             */
                     42    odobat   +10  , /* Object attribute        */
                     53    odobsz   +6   , /* Object size (packed)    */
                     59    odobtx   +50  , /* Object text             */
                      .                    , /* Place holder            */
```

```
     /* Use P2D (Appendix example) to: unpack object size.            */
     odobsz = P2D(odobsz,0)

     IF output_format = '*SUMMARY' THEN DO
          CALL summary_output
     END
     ELSE DO
          CALL detail_output
     END

  END
  'RMVREXBUF'
  EXIT(0)


/**********************************************************************/
/* summary_output : Internal Routine to Produce summary output       */
/**********************************************************************/
summary_output:

  /* Print header for 1st page.*/
  IF i = 1 THEN DO
    pagenum = 1
    CALL write_summary_header
  END

  /* If change in control fields then display line & */
  /*      accumulate totals                          */
  IF (odobtp ¬= prev_odobtp | odobat ¬= prev_odobat) THEN DO
       IF i ¬= 1 THEN  DO
           type = prev_odobtp
           attr = prev_odobat
           CALL write_summary_line
           total_objects = total_objects + total_count
           total_size    = total_size    + total_odobsz
           total_odobsz = 0
           total_count =  0
       END
       prev_odobtp = odobtp
       prev_odobat = odobat
  END

  total_odobsz =  odobsz + total_odobsz
  total_count = total_count + 1

  /* If last record being processed, then display total.*/
  IF  i = count THEN DO
   IF total_count > 0  THEN DO
           type = odobtp
           attr = odobat
           CALL write_summary_line
           total_objects = total_objects + total_count
           total_size    = total_size    + total_odobsz
   END
   CALL write_summary_total
  END

RETURN
```

```
/********************************************************************/
/* write_summary_header: Write a summary header output lines        */
/********************************************************************/
write_summary_header:
SAY,
'                     Display Library Profile         Page 'pagenum
SAY,
'    Object Type   attr    # of objects  Total Size               '
SAY,
'    _____   _____   _____    _____               '
RETURN


/********************************************************************/
/* write_line: Write a summary output line                          */
/********************************************************************/
write_summary_line:
line = ''
line=INSERT(type,line,7,10)
line=INSERT(attr,line,21,5)
line=INSERT(total_count,line,29,10)
line=INSERT(total_odobsz,line,43,10)
line=strip(line,'t')
call check_overflow
SAY line
RETURN


/********************************************************************/
/* write_summary_total:  Write a summary total line                 */
/********************************************************************/
write_summary_total:
line = ''
line=INSERT(total_objects,line,29,10)
line=INSERT(total_size,line,43,10)
line=strip(line,'t')
call check_overflow
SAY,
'                Totals --> ===========   ===========            '
call check_overflow
SAY line
RETURN


/********************************************************************/
/* check_overflow:  Check for page overflow, if so output header    */
/********************************************************************/
check_overflow:
/* If page overflow, then perform header output.   */
IF  page_rec_count > 24 THEN DO
   page_rec_count = 0
   pagenum = pagenum + 1
   IF output_format = '*SUMMARY' THEN
      CALL write_summary_header
   ELSE CALL write_detail_header
END
ELSE page_rec_count = page_rec_count + 1
RETURN
```

```
/********************************************************************/
/* detail_output : Internal Routine to produce detail output      */
/********************************************************************/
detail_output:

  /* Print header for 1st page.*/
  IF i = 1 THEN DO
    pagenum = 1
    CALL write_detail_header
  END

  /* If change in control fields, then display line      */
  /*    and accumulate totals.                           */

  total_odobsz =  odobsz + total_odobsz
  total_count = total_count + 1
  CALL write_detail_line

  /* If last record being processed, then display total.*/
  IF  i = count THEN DO
   CALL write_detail_total
  END


RETURN
/********************************************************************/
/* write_detail_header: Write a detail header output line          */
/********************************************************************/
write_detail_header:
SAY,
'                     Display Library Profile          Page 'pagenum
SAY,
'  Object name Object Type  attr.   size      Short description      '
SAY,
'  _____ _____  ____ _____  _____'
RETURN


/********************************************************************/
/* write_detail_line:   Write a detail output line                 */
/********************************************************************/
write_detail_line:
line = ''
line=INSERT(odobnm,detail_line,3,10)
line=INSERT(odobtp,line,15,10)
line=INSERT(odobat,line,28,5)
line=INSERT(format(odobsz,10),line,32,10)
line=INSERT(odobtx,line,45,25)
line=strip(line,'t')
call check_overflow
SAY line
RETURN

/********************************************************************/
/* write_detail_total : Write a detail total line                  */
/********************************************************************/
write_detail_total:
line = ''
```

```
line=INSERT(format(total_odobsz,10),line,32,10)
call check_overflow
SAY,
'          Total size(bytes) -->   =========                    '
line=strip(line,'t')
SAY line
RETURN


/**********************************************************************/
/* command_error_trap: ERROR & FAILURE condition trap (routine)     */
/**********************************************************************/
COMMAND_ERROR_TRAP:
OK_exceptions = ''
IF POS(rc,OK_exceptions) = 0 THEN DO
  PARSE SOURCE . . lib srcfile srcmbr
  SAY 'REXX program:'lib'/'srcfile srcmbr 'detected exception'
  SAY 'The line number is:'sigl
  SAY 'The exception id is:'rc
  SAY 'The command that caused the error is:'
  SAY SOURCELINE(sigl)
  EXIT(1)               /* Exit the REXX program. */
END


/**********************************************************************/
/* SYNTAX:  Syntax condition trap (branch point)                    */
/**********************************************************************/

SYNTAX:
PARSE SOURCE . . lib srcfile srcmbr
SAY 'REXX program:'lib'/'srcfile srcmbr 'trapped a SYNTAX error'
SAY 'The line number is:'sigl
SAY 'The failing REXX statement is:'
SAY SOURCELINE(sigl)
SAY 'The error code is'rc '. The description of the error is:'
SAY ERRORTEXT(rc)
EXIT(1)             /* Exit the REXX program. */


/**********************************************************************/
/* NOVALUE:  Trap references to uninitialized rexx variables        */
/**********************************************************************/

NOVALUE:
PARSE SOURCE . . lib srcfile srcmbr
SAY 'REXX program:'lib'/'srcfile srcmbr 'trapped a NOVALUE condition'
SAY 'The line number is:'sigl
SAY 'The REXX clause statement is:'
SAY SOURCELINE(sigl)
EXIT(1)               /* Exit the REXX program. */
```

Example 5: This example moves objects between libraries. It shows a
comprehensive way to perform error checking within a REXX program. It checks
for nonzero return codes and responds to them with an error checking internal
subroutine.

```
/********************************************************************/
/* This REXX program will move an Object from a Test library to a   */
/* Production Library.                                              */
/*                                                                  */
/*   Arguments passed:                                              */
/*                                                                  */
/*       Object  - Object name                                      */
/*       Objtype - Object type                                      */
/*       Operation - Function to perform 'C' Copy object to Production*/
/*                                        Library.                  */
/*                                   'R' Replace object in          */
/*                                        Production Library.       */
/*                                                                  */
/********************************************************************/

PARSE UPPER ARG object objtype oper

/* Validate length and data type of file.*/
IF WORDLENGTH(file) >= 10 | obj  = '' | DATATYPE(obj,'A') = 0 THEN DO
     'SNDPGMMSG MSG('Please enter a valid file name')'
     exit
END

/* Set validate object types list.*/
valid_objtype_list = '*DTAARA *DTAQ *FILE *JOBD *JOBQ *MENU *MSGF',
                     '*MSGQ *PGMQ'


/* Validate object types.*/
IF POS(objtype,valid_objtype_list) = 0 THEN DO
  msg = 'Object Type' objtype 'is not a supported type. Please supply',
        'a valid type of:' valid_objtype_list

  'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
  RETURN
END

/* Set validate object types list.*/
valid_oper_list = 'M C MOVE COPY *MOVE *COPY'

IF POS(oper,valid_oper_list) = 0 THEN DO
  msg = 'Operation' oper 'is not a supported operation.Please supply',
        'a valid operation code of:' valid_oper

  'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
  RETURN
END


/* Check for object existence.    */
'CHKOBJ TESTLIB/&object OBJTYPE(&objtype)'
IF rc¬='' THEN
```

```
                         IF POS(rc,'CPF9801 CPF9810') THEN DO
                            msg = 'Object:' object 'of type:' objtype 'does not exist in',
                                   'TESTLIB'
                           'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
                            RETURN
                         END

                   SELECT

                      /* Copy the object into the Production library.  */
                     WHEN POS(oper,'C COPY *COPY') ¬= 0 THEN DO

                      /* Does the object already exists in the library?  */
                       'CHKOBJ PRODLIB/&object OBJTYPE(&objtype)'
                       IF rc ='' THEN  DO
                           msg = 'Object:' object 'of type:' objtype 'already exists in',
                                   'TESTLIB ... Please delete object or specify MOVE option'
                           'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'

                            CALL ON ERROR name error_handler
                            /* Move the object from TEST lib to PROD lib. */
                            'MOVOBJ OBJ(TESTLIB/&object) OBJTYPE(&objtype) TOLIB(PRODLIB)'
                            CALL OFF ERROR

                           RETURN
                       END
                     END

                     /* Move the object into the Production library.  */
                     WHEN POS(oper,'M MOVE *MOVE') ¬= 0 THEN DO
                       /* Does the object already exist in the library?   */
                       'CHKOBJ PRODLIB/&object OBJTYPE(&objtype)'
                       /* If so  */
                       IF rc ='' THEN  DO
                         /* Call routine to delete the object. */
                         CALL dltobj_routine object, objtype

                         IF result = '' THEN DO

                            CALL ON ERROR name error_handler
                            /* Move the object from TEST lib to PROD lib. */
                            'MOVOBJ OBJ(TESTLIB/&object) OBJTYPE(&objtype) TOLIB(PRODLIB)'
                            CALL OFF ERROR

                         END
                         ELSE DO
                            EXIT(result)                    /*Exit, returning exception id.*/
                         END
                       END
                     END

                   /************************************/
                   /* dltobj_routine -                 */
                   /*    Internal REXX routine called  */
                   /*  to delete a specific object type */
                   /*  from the PRODLIB.               */
                   /************************************/
```

```
dltobj_routine:

PARSE ARG object, objtype

SELECT  /* Select DLT command based on object being deleted.*/

   WHEN objtype = '*FILE' THEN  dlt_noun = 'FILE'
   WHEN objtype = '*DTAARA' THEN dlt_noun = 'DTAARA'
   WHEN objtype = '*DTAQ' THEN dlt_noun = 'DTAQ'
   WHEN objtype = '*JOBD' THEN dlt_noun = 'JOBD'
   WHEN objtype = '*JOBQ' THEN dlt_noun = 'JOBQ'
   WHEN objtype = '*MENU' THEN dlt_noun = 'MENU'
   WHEN objtype = '*MSGF' THEN dlt_noun = 'MSGF'
   WHEN objtype = '*MSGQ' THEN dlt_noun = 'MSGQ'
   WHEN objtype = '*PGM' THEN dlt_noun = 'PGM'
   OTHERWISE

 DLTCMD = 'DLT'dlt_noun       /* Concat DLT and dlt_noun into 1 word.*/

 DLTCMD 'PRODLIB/&object'    /* Issue COMMAND with var DLTCMD.      */

 IF rc ¬= '' THEN DO
    msg = 'Unexpected error:'rc 'encountered while issuing:' DLTCMD,
          'Please see Joblog for details'
    'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'

    RETURN(rc)              /* Return to caller with exception id.*/
 END
 RETURN

/************************************/
/* error_handler -              */
/*    REXX ERROR condition handler   */
/*  routine 'SET' to handler ERRORS  */
/*  occurring from a MOVOBJ command. */
/************************************/

error_handler:

msg = 'Error encountered during MOVOBJ command for object:'object,
      ' From TESTLIB to PRODLIB. Please see joblog for details'
      'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'

RETURN
```

Example 6: This programs determines if a job is active.

```
/**********************************************************************/
/*  REXX program to determine if a Job is active.                     */
/*                                                                    */
/*    Parameters:  Job name                                           */
/*                 Job number                                         */
/*                 User                                               */
/*                                                                    */
/*    Returns:     "1" active , "0" NOT active                        */
/*                                                                    */
/**********************************************************************/


         PARSE ARG job_name user job_number

         /* Add a Buffer to the REXX data queue.*/
          buffer = 0
         'ADDREXBUF  BUFFER(&buffer)'

         /*  Create active job listing.    */
         'WRKACTJOB OUTPUT(*PRINT)'

         /* Create temporary work file to store spooled file output (WRKACTJOB),*/
         /* if needed.*/
         'CHKOBJ QTEMP/ACTJOBLST *FILE'
         IF rc ¬= '0' THEN
           'CRTPF QTEMP/ACTJOBLST RCDLEN(132)'

         /*Copy the last spooled file from WRKACTJOB to the temporary work file.*/
         'CPYSPLF FILE(QPDSPAJB) TOFILE(QTEMP/ACTJOBLST) SPLNBR(*LAST)'

         /*Copy the WRKACTJOB output from QTEMP to the REXX queue.*/
         'CPYFTOREXQ FROMFILE(QTEMP/ACTJOBLST) MBR(ACTJOBLST)'

         DO QUEUED()

          PARSE PULL act_job_name act_user act_job_number rest

          /* If active job is same as argument, then return 1.*/
          IF job_name = act_job_name & job_number =  act_job_number & ,
            user = act_user THEN DO

             /* Remove Buffer from the REXX data queue.*/
             'RMVREXBUF BUFFER(&buffer)'

             EXIT(1)
          END
         END
         /* Remove Buffer from the REXX data queue.*/
         'RMVREXBUF BUFFER(&buffer)'
         /* No active job found on queue.*/
         EXIT(0)
```

Example 7: This program moves an object from one library to another.

```
/*******************************************************************/
/* This REXX program will move a Object from a Test library to a    */
/* Production Library.                                              */
/*                                                                  */
/*  Parameters:                                                     */
/*                                                                  */
/*      Object   - Object name                                      */
/*      Objtype  - Object type                                      */
/*      Operation - Function to perform 'C' Copy object to Production*/
/*                                        Library.                  */
/*                                   'R' Replace object in          */
/*                                        Production Library.       */
/*                                                                  */
/*  Returns:    '0' if operation successful                         */
/*              '1' if operation NOT successful                     */
/*                                                                  */
/*                                                                  */
/*                                                                  */
/*******************************************************************/


PARSE UPPER ARG object objtype oper

/* Set validate object types list.*/
valid_objtype_list = '*DTAARA *DTAQ *FILE *JOBD *JOBQ *MENU *MSGF',
                     '*MSGQ *PGMQ'

/* Validate object types.*/
IF POS(objtype,valid_objtype_list) = 0 THEN DO
  msg = 'Object Type' objtype 'is not a supported type. Please supply',
        'a valid type of:' valid_objtype_list

    'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
     EXIT(1)
END

/* Set validate object types list.*/
valid_oper_list = 'M C MOVE COPY *MOVE *COPY'

IF POS(oper,valid_oper_list) = 0 THEN DO
  msg = 'Operation' oper 'is not a supported operation. Please supply',
        'a valid operation code of:' valid_oper_list

  'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
   EXIT(1)
END

/* Check for object existence    */
'CHKOBJ TESTLIB/&object OBJTYPE(&objtype)'
IF rc¬= '0' THEN
  IF POS(rc,'CPF9801 CPF9810') ¬= 0 THEN DO
    msg = 'Object:' object 'of type:' objtype 'does not exist in',
          'TESTLIB'
   'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
    EXIT(1)
  END
```

```
            SELECT

               /* Copy the object into the Production library.  */
              WHEN POS(oper,'C COPY *COPY') ¬= 0 THEN DO

               /* Does the object already exists in the library?  */
                'CHKOBJ PRODLIB/&object OBJTYPE(&objtype)'
                IF rc ='0' THEN  DO
                   msg = 'Object:' object 'of type:' objtype 'already exists in',
                         'TESTLIB ... Please delete object or specify MOVE option'
                   'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
                   EXIT(1)
                END
                ELSE DO
                    /* Set ERROR trap ON.*/
                    SIGNAL ON ERROR name error_handler
                    /* Move the object from TEST lib to PROD lib. */
                   'MOVOBJ OBJ(TESTLIB/&object) OBJTYPE(&objtype) TOLIB(PRODLIB)'
                    /* Set ERROR trap OFF.*/
                    SIGNAL OFF ERROR
                    EXIT(0)
                END
              END

              /* Move the object into the Production library.  */
              WHEN POS(oper,'M MOVE *MOVE') ¬= 0 THEN DO
                /* Does the object already exists in the library?  */
                'CHKOBJ PRODLIB/&object OBJTYPE(&objtype)'
                /* If so, */
                IF rc ='0' THEN  DO
                  /* Call routine to delete the object. */
                  CALL  dltobj_routine  object, objtype
                END
                    /* Set ERROR trap ON.*/
                  SIGNAL ON ERROR name error_handler
                    /* Move the object from TEST lib to PROD lib. */
                 'MOVOBJ OBJ(TESTLIB/&object) OBJTYPE(&objtype) TOLIB(PRODLIB)'
                    /* Set ERROR trap OFF.*/
                  SIGNAL OFF ERROR
              END
             OTHERWISE
            END
            EXIT(0)

            /************************************/
            /* dltobj_routine -                 */
            /*    Internal REXX routine called  */
            /*  to delete a specific object type */
            /*  from the PRODLIB.               */
            /************************************/

            dltobj_routine:

            PARSE ARG object, objtype

            SELECT  /* Build DLT command based on object being deleted.*/
               WHEN objtype = '*FILE' THEN  dlt_noun = 'F'
```

```
        WHEN objtype = '*DTAARA' THEN dlt_noun = 'DTAARA'
        WHEN objtype = '*DTAQ' THEN dlt_noun = 'DTAQ'
        WHEN objtype = '*JOBD' THEN dlt_noun = 'JOBD'
        WHEN objtype = '*JOBQ' THEN dlt_noun = 'JOBQ'
        WHEN objtype = '*MENU' THEN dlt_noun = 'MNU'
        WHEN objtype = '*MSGF' THEN dlt_noun = 'MSGF'
        WHEN objtype = '*MSGQ' THEN dlt_noun = 'MSGQ'
        WHEN objtype = '*PGM' THEN dlt_noun = 'PGM'
        OTHERWISE
    END

    DLTCMD = 'DLT'dlt_noun      /* Concat DLT and dlt_noun into 1 word.*/

    DLTCMD 'PRODLIB/&object'    /* Issue COMMAND with var DLTCMD.      */
                               /* Use a PCLV for object to be deleted.*/

    IF rc ¬= '0' THEN DO
       msg = 'Unexpected error:'rc 'encountered while issuing:' DLTCMD,
             'Please see Joblog for details'
       'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'

       EXIT(1)                 /* Exit with failure indication.      */
    END
    RETURN

/************************************/
/* error_handler  -                 */
/*    REXX ERROR condition handler  */
/*  routine 'SET' to handler ERRORS */
/*  occurring from a MOVOBJ command. */
/************************************/

error_handler:

    msg = 'Unexpected error:'rc 'encountered while issuing:' MOVOBJ,
          ' From TESTLIB to PRODLIB. Please see joblog for details'
    'SNDPGMMSG  MSG(&MSG) TOPGMQ(*PRV)'
  EXIT(1)                 /* Exit with failure indication.      */

RETURN
```

Example 8: The following program can be used to try different trace options.

```
/* This program performs a variety of operations.  Different      */
/* trace options are used in this program to show how they differ.  */

Say 'Enter trace option to use'
pull traceopt
Trace Value traceopt

Say '1+1 equals' 1+1
two = 2
Say 'The maximum of 1 and 2 is' Max(1,two)
Call Time
Say 'The current time is' result

Say 'Factorial 3 is' fact(3)

"SNDMSG MSG('Hello') TOUSR(*SYSOPR)"
"SNDMSG MSG('Hello') TOUSR(*WRONG)"

Say 'This message has a missing quote

exit

Fact: Procedure
arg n
if n = 0 then return 1
   else return n * fact(n-1)
```

Appendix G. Communication Between REXX/400 and ILE/C

This appendix summarizes, with examples and descriptions, the methods by which REXX/400 and ILE/C programs can communicate. The items that are covered are:

- How a REXX program can call an ILE/C program

- How the REXX/400 program can pass parameters to an ILE/C program

- How ILE/C receives the parameters when it gets control

- How the ILE/C program can let the REXX program know whether it ran successfully

- How the ILE/C program can pass results, if any, to the REXX/400 program

- A complete example, which illustrates some of the principles mentioned above.

Note: The examples in this appendix work the same for C/400 programs and ILE/C programs.

The purpose of this appendix is to combine information relevant to inter-program communication in one place by using fully described program outlines and examples, both in the REXX/400 and ILE/C languages. For detailed descriptions of each of the interfaces described here, see the *REXX/400 Reference*.

Calling an ILE/C Program From REXX

REXX can call an ILE/C program in a number of ways, including:

- As an external subroutine

- As an external function

- As a command environment

- By using the CALL command.

Calling ILE/C as an External Subroutine

Calling an ILE/C program as an external subroutine is achieved by way of the REXX/400 CALL instruction. The following is an example of the REXX code that can be used:

```
/* REXX/400 calls ILE/C as an external subroutine.*/

parm1 = 'value1'                /* Here is where the        */
parm2 = 'This is a string '     /* parameters to be passed  */
                                /* to the C program are set */
                                /* up (see note 1).         */

call "C_PROGRAM" parm1, parm2   /* The call to the C program */
                                /* (see note 2).             */
return_value = result           /* The results from the C    */
                                /* program are received      */
                                /* (see note 3).             */
```

Notes:

1. Parameters can be set up as character strings containing any kind of data (for example, numbers or words). Each parameter must be separated from the others on the CALL instruction by a comma, in order to be received by the called ILE/C program as an individual parameter. A maximum of 20 parameters can be passed. Also, all data in REXX/400 programs is stored as character strings. The called program will have to convert the character data received from REXX to whatever format it requires.

2. Although not necessary, the name of the ILE/C program called is in capital letters and enclosed in quotation marks. This will cause the search for internal subroutines (labels in your REXX/400 program) to be bypassed and result in a faster call to your program. For the call to be successful, the called ILE/C program must be in the job's current library list.

3. The REXX program may receive a result in the special variable RESULT if it was set by the ILE/C program through the variable pool interface. Results may also be returned in the REXX external data queue. These two APIs are discussed in the following sections.

Calling ILE/C as an External Function

Special syntax is required when a REXX program calls another program or subroutine as a function. This syntax is the same as it is for calling internal or built-in functions. Otherwise, the rules are almost the same as for calling external subroutines. The following is an example of the REXX code that can be used:

```
/* REXX/400 calls ILE/C as an external function.*/

parm1 = 'value1'                      /* Here is where the      */
parm2 = 'This is a string '           /* parameters to be passed */
                                      /* to the C program are set */
                                      /* up.                     */

ret_val = "C_PROGRAM"(parm1, parm2) /* The C program is called  */
                                      /* and the results are     */
                                      /* received (see note 1).  */
```

Note: The expression calling the ILE/C program is replaced by the result obtained from it. This is why a program called as a function **must** return a result, while a result is not required when a ILE/C program is called as a subroutine. The called ILE/C program must use the variable pool interface to pass this result to the interpreter. How to do this will be discussed later.

Calling ILE/C as a Command Environment

CL is the default command environment to which the interpreter passes all clauses which are not REXX instructions. If the desired environment is an ILE/C program, it must first be identified in the REXX program. This is accomplished by using the ADDRESS instruction with the name of the ILE/C program, which may be a simple or qualified name.

In this case, the ILE/C program is serving the REXX interpreter as a command handler. The interpreter will evaluate all the parameters in the clause, concatenate them in the order written, and pass them as a command to the specified environment in a single character string.

The following is an example of the REXX code that can be used:

```
/* REXX/400 passes parameters to ILE/C as a command to a
             command environment.*/

parm1 = 'value1'                       /* Here is where the       */
parm2 = '"This is a string"'           /* parameters to be passed  */
                                       /* to the C program are set */
                                       /* up (see note 1).         */

address 'LIBRARY/C_PROGRAM'            /* Setting the command      */
                                       /* environment (see note 2). */
parm1 parm2                            /* The C program is called,  */
                                       /* passing the parameters   */
                                       /* (see note 3).            */

ret_code = RC                          /* The results from the C   */
                                       /* program are received     */
                                       /* (see note 4).            */
```

Notes:

1. All the data to be passed to a command environment program is passed in one character string parameter. If the command environment program needs to have separate parameters, then it should define a delimiter character that can be used to identify where one parameter ends and the next begins. In this example, a single parameter is delimited by quotation marks, which the C program will have to search for as it processes the command string.

2. If LIBRARY is not specified, the job's library list will be used to resolve to the command environment program. Remember that this form of the ADDRESS instruction makes a lasting change to the destination of commands. Any command further on in the REXX program will be passed to your ILE/C program unless the environment is changed by another ADDRESS instruction.

3. The expression, which may consist of a list of parameters, is evaluated. Since there is no REXX instruction in this program clause, the entire expression is submitted as a single command string to the environment declared in the last ADDRESS instruction, which is the ILE/C program. The latter is responsible to parse the single string into the individual parameters.

4. A return code will always be available in the special variable RC. If the ILE/C program does nothing to return a value, RC will have the value 0. This is because the parameter string in which the ILE/C program returns return codes is preallocated and initialized to 0, indicating that the command ran successfully. If the command environment program found a problem, such as an error in the command string, the ILE/C program must indicate this by putting a nonzero return code string in the buffer passed. Additional results obtained by the ILE/C program can only be returned by using the REXX External Data Queue or the Variable Pool Interface.

Calling ILE/C with the CL CALL Command

As was mentioned previously, CL is the default command environment for REXX/400 programs, unless it was specified otherwise on the call to the interpreter. If the current command environment is not CL, it can be reset to the CL environment by specifying an environment name of COMMAND as follows:

```
ADDRESS COMMAND
```

Any clause that the interpreter does not recognize as a REXX instruction, for example, a quoted string, will now be passed to the CL command environment where the OS/400 command processor will run it as a CL command.

When CL commands are run from REXX programs, they can include the names of REXX variables which can be used and changed by the command. These variables work in a way similar to variables in CL programs, thus they are called pseudo-CL variables. Their names in the commands are prefixed with the ampersand (&) character, and their names must conform to both the REXX and CL naming conventions. For more information on pseudo-CL variables, see the *REXX/400 Reference* and "Understanding Pseudo-CL Variables" on page 88.

With these requirements in mind, a REXX program can call an ILE/C program and obtain results from it by using the CALL command. The following REXX code illustrates how it can be done:

```
/* REXX/400 program using CL to call an ILE/C program. */

parm1 = 'This is a string'   /* Here is where the                */
parm2 = 123456               /* parameters to be passed          */
parm3 = ''                   /* to the C program are set          */
                             /* up (see note 1).                 */
address command             /* See note 2.                      */
'call C-PROGAM parm(&parm1, &parm2, &parm3)' /* See note 3.    */
say 'C program ended with return code' RC
                             /* See note 4.                      */
```

Notes:

1. The CALL command is run as if it was issued from a command line. Remember that all variable values in REXX are character strings even if they appear to be numeric as in parm2 above. So, even if a parameter looks like a number, it is only converted to packed decimal if it is not contained in a pseudo-CL variable. See the ILE/C program under "Calling ILE/C Programs with the CL CALL Command" on page 183 to see how the parameters passed in this example are received.

2. The ADDRESS instruction makes sure that the command is passed to the CL command environment. Without it, the command might be passed to another command environment.

3. Since the clause is only an expression, it is interpreted as a command and passed to the current command environment.

4. The CL command environment will always see to it that the REXX special variable RC is set to a value. RC will contain a 0 if no escape message is received by the interpreter, otherwise it will have the 7-character message ID of the escape message.

See "Returning Results and Return Codes from ILE/C Programs" on page 184 for a summary of where and how to find return codes and results received from a called ILE/C program.

Passing Parameters and Control to ILE/C

Remember that even though we may talk about a REXX program and how it is called, REXX programs are not program objects and are not called in the same way that program objects are called. A REXX program is just a source file member that contains a listing of all the actions required by the program's user. When a REXX program is run, the REXX interpreter is called with the name of the source member containing the program to be run. The interpreter then reads and runs the statements in the source file member. When the interpreter finds a statement that tells it to call a program that is not written in REXX, it calls the program passing the parameters in the special form defined by the interface.

Some of the most commonly used of these conventions are described in the following sections.

Calling External Subroutines and Functions

Whenever the interpreter comes across a call to an external subroutine or a function call, each of the parameter strings accompanying the call is changed into a RXSTRING format before it is passed to the ILE/C program.

The RXSTRING format is a structure declared in the C language as follows:

```
typedef struct
{  char          * rxstrptr;   /* Pointer to data string.*/
   unsigned long  rxstrlen;   /* Length of data string. */
} RXSTRING;
```

Note: When the interpreter stores the contents of a RXSTRING parameter before passing control to the ILE/C program, it does **not** end the string with a null character (\0). The ILE/C program must, therefore, use the length member of the structure to find the end of the string.

The interpreter then passes control to the ILE/C program together with pointers to 4 data items. These are:

1. An array of RXSTRINGs, each one pointing to a parameter string

2. The number of parameters sent from REXX

3. An indicator how the ILE/C program was called (either as a function or as a subroutine)

4. A pointer to a two-byte integer where the interpreter will expect a return code to be placed

For a complete description of these data items, see the *REXX/400 Reference*. Note that argv[1] will be pointing to an array of RXSTRING structures, each of which points to one of the parameter strings.

Calling a Command Environment

When the interpreter comes across a command, it evaluates the expression and builds a command string. It then stores the command string (which contains all of the command's parameters in a single string) in the form of a SHORT_VARSTRING structure in storage. It does **not** end the string with a null character (\0). It also allocates space for another SHORT_VARSTRING structure (in which it expects the ILE/C program to put a return value when the latter has experienced any problems) as well as a two-byte integer (in which it expects the ILE/C program to put an Error/Failure code if necessary). The addresses of these three items are now passed to the ILE/C program.

A SHORT_VARSTRING structure is declared in the C language as follows:

```
typedef struct
{  short int short_length;  /* Length of accompanying data string.*/
   char short_string[1];    /* The data string itself.           */
} SHORT_VARSTRING;
```

The SHORT_VARSTRING structure in which the interpreter expects the return value is preallocated to a length of 500 bytes and set to 0. The Error/Failure code integer is set to 0. This is done by the interpreter to save the ILE/C program the job of setting these codes if no errors were encountered.

The interpreter now calls the ILE/C program which was specified as the current command environment and passes pointers to these three data items as parameters.

Note: Although the short_string member of the SHORT_VARSTRING structure is defined as an array of length one in the example above, ILE/C does no runtime array boundary checking, so the string may be longer than one byte. Also, because the command string in storage has no ending \0, the length member must be used to obtain the correct data.

For more information, see the *REXX/400 Reference*.

Using the CL CALL Command

When REXX uses the CALL command to call a ILE/C program, the interaction between REXX and CL as well as the interaction between CL and ILE/C is transparent to the REXX programmer. The only significant items are that the ILE/C program will receive all REXX parameters as separate strings with null characters (\0) attached, and that the RC variable in REXX will be set after running the ILE/C program, if the program ended with an escape message.

Using the REXX External Data Queue

Another way of passing parameters from your REXX program to your ILE/C program is to make use of the external data queue. This method can be used in conjunction with all the interfaces mentioned above. Your REXX program puts the parameters in the REXX external data queue with the PUSH or QUEUE instructions, and then, by using any of the three methods discussed, calls your ILE/C program with or without additional parameters. Your ILE/C program then uses the QREXQ system API to pull the parameters from the external data queue.

Receiving Parameters in an ILE/C Program

With the information discussed thus far, we are now in a position to start coding our called ILE/C program and to receive the parameters sent to it by the interpreter. The main objective of this section is to show how the parameters passed from REXX are received in ILE/C variables.

Calling ILE/C Programs as External Functions or Subroutines

The example below shows some code to receive parameters from REXX when the ILE/C program was called as an external function or subroutine.

```
typedef struct                 /* The RXSTRING structure.      */
{  char          *rxstrptr ;   /* Pointer to string data.      */
   unsigned long  rxstrlen ;   /* Length of string.            */
} RXSTRING;                     /*                              */

main(int argc, char *argv[])
{

   /* Declaration of Local variables to receive the passed para- */
   /* meters follows:                                            */
   RXSTRING    *args    ;       /* Parameter array (see note 1). */
   short int    numargs ;       /* Number of elements in the     */
                                /* parameter array.              */
   short int    func_sub ;      /* Function or external          */
                                /* subroutine call.              */
   short int   *errflag ;       /* Success/Failure flag.         */

   /* Some code to get the information passed into the declared   */
   /* local variables following.                                 */

   args     = (RXSTRING *) argv[1] ;   /*    (See note 2.)       */
   numargs  = *(short int *) argv[2] ; /* Maximum of 20 allowed. */
   func_sub = *(short int *) argv[3] ; /*    (See note 3.)       */
   errflag  = (short int *) argv[4] ;  /*    (See note 4.)       */

   /* All the remaining ILE/C program statements.                */
}
```

Notes:

1. The variable *args* will contain a pointer to an array of structures of type RXSTRING. Each of these structures will contain information about one of the passed parameters.

2. Each parameter passed to the ILE/C program can now be referenced by the expression `(*args)[x]` where `x` varies from 0 to (`numargs` - 1). Remember, however, that the actual string representing a parameter is *not* ended with a null character (\0), and that the length member of the RXSTRING structures must be used to determine the length of the actual parameter. For example, if we have declared a counter `i` earlier in our program and we want to display the 'i-th' passed parameter, the following code can be used:

   ```
   printf("Parameter number %d passed is: %.*s\n", i+1,
             args[i].rxstrlen, args[i].rxstrptr);
   ```

3. The variable *func_sub* will contain a 1 if our ILE/C program was called as an external subroutine and a 2 if it was called as an external function. If our ILE/C

program was called as a function, it must return a result to REXX through the variable pool interface. If called as an external subroutine, returning a result is optional. How results are returned will be discussed under "Returning Results and Return Codes from ILE/C Programs" on page 184.

4. The variable *errflag* is used by the C program to indicate to the interpreter whether the function ran successfully or not.

Calling ILE/C Programs as Command Environments

The example below shows some code to receive the command string when your ILE/C program was called as a command environment.

```
typedef struct                 /* Declaration of SHORT_VARSTRING.*/
{   short int short_length ;   /* Length of string.             */
    char short_string[1] ;      /* String data.                 */
} SHORT_VARSTRING ;            /*                               */

main(int argc, char *argv[])
{

   /* Declaration of Local variables to receive the passed para- */
   /* meters following.                                          */
   SHORT_VARSTRING *command ;   /* Command string  - (see note 1).*/
   SHORT_VARSTRING *rc ;        /* Return code - (see note 2).   */
   short int       *err_fail ; /* Success/Failure flag.         */

   /* Some code to get the information passed into the declared   */
   /* local variables following.                                 */

   command  = (SHORT_VARSTRING *) argv[1] ; /*    (See note 1.)  */
   rc       = (SHORT_VARSTRING *) argv[2] ; /*    (See note 2.)  */
   err_fail = (short int *) argv[3] ;       /*    (See note 3.)  */

   /* All the remaining ILE/C program statements.                */
}
```

Notes:

1. In this case, `argv[1]` contains a single pointer to the SHORT_VARSTRING structure described earlier. This is because all the parameters sent by REXX have been put in a single string. Once again, the command string does not have an ending null character (\0), so your code must use the length field of the SHORT_VARSTRING structure. For example, if you want to display the command string received in the example above, you will code:

   ```
   printf("The command string is: %.*s\n", command->short_length,
                          command->short_string);
   ```

   Moreover, because you will be receiving all the parameters in one string, your ILE/C program must do the parsing of the parameters. If one of the parameters consists of a string of separate words, make sure that your REXX program has it delimited by characters which will be known by your ILE/C program (for instance, quotation marks).

2. `argv[2]` will also contain a pointer to a SHORT_VARSTRING structure. The string member of this structure is preallocated to a length of 500 characters and set to 0. The members of this structure must be reset by the ILE/C program to return values for the special variable RC if any problems were encountered.

The interpreter will automatically update the REXX variable RC with what you put in this structure. RC may then be used by your REXX program to figure out what went wrong.

3. `argv[3]` is a pointer to a two byte integer which is initialized by the interpreter to 0. If not altered by your ILE/C program, this will indicate to REXX that your ILE/C program ran successfully. You may, however, assign a 1 or a 2 to this variable, indicating to the interpreter that some problems have been encountered. A 1 would tell the interpreter to raise the ERROR condition and a 2, the FAILURE condition.

Calling ILE/C Programs with the CL CALL Command

As already seen, all REXX parameters received at an ILE/C program through the CL command environment are valid C strings that end with null characters. The following ILE/C program illustrates how the variables sent by the REXX program in "Calling ILE/C with the CL CALL Command" on page 178 are received:

```
main(int argc, char *argv[])
{

    printf("First parm received is ++%s++\n",argv[1]); /* See below.*/
    printf("Second parm received is ++%s++\n",argv[2]);/* See below.*/
    printf("Third parm received is ++%s++\n",argv[3]); /* See below.*/
    return;
}
```

The output for this program will be:

```
First parm received is ++This is a string++
Second parm received is ++123456++
Third parm received is ++++
```

Receiving Parameters from the REXX External Data Queue

Apart from the ways in which ILE/C programs can receive parameters from REXX that have already been discussed, they can also receive any data that the ILE/C program expects through the REXX external data queue. This functions as a common buffer between REXX and any other language that REXX communicates with. Data can be placed in and retrieved from the queue by any program at any time.

Although nothing prohibits the programmer from using the REXX external data queue to pass parameters, this is not normally done. The external data queue is more commonly used to send data records between the two participating programs.

An example of how the REXX external data queue is used is given in "Example Using the REXX External Data Queue" on page 194, after it is discussed in more detail under "Returning Results in the REXX External Data Queue" on page 193.

Returning Results and Return Codes from ILE/C Programs

When REXX programs call other programs, they generally expect returned values that indicate success or failure of that program, or that are the results that program has obtained. The following is a summary of what REXX expects in each situation:

Calling an External Subroutine: When control is returned to REXX after an external subroutine has been called, the special variable RESULT may or may not have a value. This is because it is not mandatory for the called external subroutine to return anything. However, if it does, it must make use of the SHVEXTFN function of the variable pool interface to return the information. This is discussed in "Returning Results with the Variable Pool Interface" on page 185.

The RC special variable is not used in this case.

Calling an External Function: The expression that specifies the function to be called is replaced by the function's result. This is also done by using the SHVEXTFN function of the variable pool interface. Neither RESULT nor RC are used.

Sending Commands to a User-defined Command Environment: The special variable RC will always contain a return code while the special variable RESULT is not used. As seen in "Calling ILE/C Programs as Command Environments" on page 182, an ILE/C program that is specified as the current command environment is always called with three parameters. The string part of the SHORT_VARSTRING structure to which the second parameter points (the return code buffer) is preallocated to a length of 500 characters and set to a value of 0. The two-byte integer that the third parameter is pointing to, is initialized to 0. If the ILE/C program runs successfully, these values remain as is and the 0 in the buffer is assigned to the special RC variable by the interpreter. If, however, an ERROR or FAILURE condition should arise in the ILE/C program, it is the programmer's responsibility to set the two-byte integer, to which the third parameter is pointing, to a 1 or a 2 respectively. The programmer must also assign an appropriate error indication string, with a maximum length of 500 characters, to the string part of the return code buffer structure and update the length part to indicate the length of the string.

Note: If the interpreter receives an escape message from the command environment program, the return code buffer is ignored, and the RC variable is set to the message ID of the received escape message.

Using the CL CALL Command: The CL command environment will always set RC before returning control to REXX. The value will be a zero (0) if the called program ran successfully. If problems occurred while running the program, the program should issue an escape message. The RC variable will be set to the escape message ID. For information on how error codes are handled, see the *REXX/400 Reference*.

In addition, the C program can assign values to parameters that were specified as pseudo-CL variables. The REXX program can then use the contents of those variables.

Returning Results with the Variable Pool Interface

You should be familiar with the description of the "Shared-Variable Request Block" in the *REXX/400 Reference* before starting to read this section. That description will not be repeated here, but is required to refer to when you want to learn how to use the function.

Whenever a REXX program is started, a space which will contain information about all variables used in the program is created and maintained by the interpreter. The interpreter sees to it that all data that makes up the value of a specific variable is linked to that variable's name. Any changes made to the contents of a variable in a REXX program, are recorded by the interpreter so that when the variable is accessed by name again, the new value will be available. The space where this information is kept is known as the variable pool.

QREXVAR is a program in the QSYS library that may be called by a program written in another language to access this variable pool. With this interface, the program can read, change, create, or delete any data contained in there. This called program must have been called by a REXX program, directly or indirectly, before this function becomes available. Also, only the variable pool of the currently active REXX program will be accessible to this program.

Only languages that make use of pointers are able to make use of this function. This is because QREXVAR expects a pointer to a list of pointer-linked *request blocks* as one of its parameters when it is called, and because these request blocks contain RXSTRING structures, each of which contains a pointer. The only other parameter QREXVAR expects is a pointer to a two-byte integer in which it will place a return code to indicate what happened while it was running.

QREXVAR is called as follows:

```
#pragma linkage(QREXVAR,OS) /* OS/400 linkage conventions */

extern void QREXVAR (SHVBLOCK *, short int *); /*Prototype */

   QREXVAR(shvblock_ptr, &return_code);  /*Calling the API  */
```

The `shvblock_ptr` parameter is a pointer to the list of request blocks mentioned above. The layout of a request block in ILE/C is as follows:

```
typedef struct shvnode
{  struct shvnode *shvnext;    /*Pointer to next request block.*/
   RXSTRING       shvname ;    /*Pointer to variable name.     */
   RXSTRING       shvvalue;    /*Pointer to value buffer.      */
   unsigned char  shvcode ;    /*Individual function code.     */
   unsigned char  shvret  ;    /*Individual return code flags. */
} SHVBLOCK;
```

The first field of this request block, `shvnext`, may contain a pointer to another request block so that a chain of request blocks can be submitted to QREXVAR in a single call. QREXVAR will process these request blocks one after the other until it finds a null pointer (0) in a `shvnext` field. This indicates that the last request block has been reached.

The second field, `shvname`, describes the name, in a RXSTRING structure format, of the variable on which an operation must be performed. See "Calling External Subroutines and Functions" on page 179 for a description of the RXSTRING

structure format. This name, and its length, must be supplied by the ILE/C program when operations on a particular variable are to be performed. For the fetch next variable function, this RXSTRING must identify the location and length of a buffer that will receive the name of the variable. QREXVAR will fill this buffer with the name and update the `rxstrlen` field to reflect the actual length of the variable name.

The third field, `shvvalue`, describes the value in a RXSTRING structure format of the variable on which an operation must be performed. The ILE/C program must update this field with the value it wants to assign to an existing or new variable. If QREXVAR is requested to return the value of a variable, this RXSTRING must identify the location and length of a buffer that will receive the value of the variable. QREXVAR will fill this buffer with the value and update the `rxstrlen` field to reflect the actual length of the value.

The fourth field, `shvcode`, is an unsigned one character field that contains the code of the function to be performed by QREXVAR. Currently, 10 functions are available and the decimal numbers 0 through 9 have been assigned to them. These numbers are used in this field to identify the required function to QREXVAR. In this book a name is assigned to each of these codes, by means of the `#define` preprocessor command, to enhance the readability of the ILE/C programs and to establish a common naming convention.

The fifth and last field, `shvret`, is an unsigned one character field that will contain a return code for that individual request block that is set by QREXVAR. There are currently 7 different possible return codes: 0,1,2,4,8,16, and 128. Just as there is a name for each function code, in this book a name is assigned to each of these codes, by means of the `#define` preprocessor command, to enhance the readability of the ILE/C program and to establish a common naming convention.

When QREXVAR returns control to the calling program, the individual return codes of all the requests that were submitted on this call are joined using logical ORs to form the final return code which is returned in the second QREXVAR parameter.

See the *REXX/400 Reference* for a complete description of all the sub-functions of QREXVAR as well as the meanings of the individual return codes from the request block(s).

To illustrate the use of this function, two examples are given:

1. The first shows a call from a REXX program to an ILE/C program which does nothing but change the value of a REXX variable. The name of the variable to be changed and its new value are passed as parameters to the ILE/C program. The REXX program does not expect any results.

2. The second example shows a REXX program which expects the ILE/C program it calls to do the following:

   - To drop the variable named in the first parameter

   - To change the value of the variable named in the second parameter to the value which appears in the third parameter

   - To create a new variable with the name that appears in the fourth parameter with a value which appears in the fifth parameter

   - To do the three actions mentioned above with a single call to the QREXVAR API

- To return the string SUCCESS to the REXX program if the operation was successful, otherwise to return the string FAILURE.

Note: Although the code in the examples constitute runable programs, many steps, such as receiving all parameters and testing validities, have been omitted in order to keep the examples as concise as possible.

Example 1: The REXX program for this example is as follows:

```
/* REXX Program calling an ILE/C program called CCHGVAR to change the  */
/* value of one of its existing variables in the shared variable pool. */

p1 = 'parm one'               /*Create the required existing variable. */
p2 = 'New value for p1'       /*The value that p1 must be changed to.  */

SAY "p1's value before change is:" p1 /*Displays 'parm two'.           */
call 'CCHGVAR' 'P1', p2   /*Ask C to change  p1's value through VPI.   */
                              /*Note that 'P1' is in capital letters    */
                              /*because SHVSET is used in ILE/C.        */

SAY "p1's value after the change is:" p1 /*Displays 'New value for p1'.*/
```

The ILE/C program for this example is as follows.

```
/****************************************************************************/
/*This program is expecting only the name of the REXX variable to          */
/*be changed and the value to which it must be changed in argv[1].         */
/*No attempt is made to receive the other parameters in argv[].            */
/****************************************************************************/

    #include <stdio.h>

/*The following code declares the structure of an RXSTRING - This is a good*/
/*candidate for an include header file.                                    */
    typedef struct                 /* RXSTRING structure definition      */
    {  char          * rxstrptr ;    /* Pointer to data                  */
       unsigned long rxstrlen ;      /* Length of data                   */
    } RXSTRING ;

/*The following code declares a structure for a request block to the       */
/*QREXVAR system API - Also a good candidate for an include header file.   */
    typedef struct shvnode         /* Shared V.P request block definition*/
    {  struct shvnode      *shvnext; /* Pointer to the next block.       */
       RXSTRING             shvname;  /* Pointer to the name buffer.      */
       RXSTRING             shvvalue; /* Pointer to the value buffer.     */
       unsigned char        shvcode;  /* Sub-function code for this block. */
       unsigned char        shvret;   /* Individual Return Code Flag.     */
    } SHVBLOCK;

/*The following code defines meaningful names to the sub-functions and     */
/*return codes of QREXVAR. Only those that will be used need to be defined.*/
/*A better idea might be to include the definitions for all sub-functions  */
/*and return codes in an include header file.                              */
    #define SHVSET          0x00     /* Request to set a variable's value. */
    #define SHVCLEAN        0x00     /* The request ran OK (Return code).  */

    #pragma linkage(QREXVAR,OS)      /* Pragma for the V.P API.            */
    void QREXVAR(SHVBLOCK *, short int *);  /*Prototype for V.P API.       */
```

```
          main(int argc, char *argv[])
          {
              RXSTRING *args;              /*Array of pointers to received parms.*/
              SHVBLOCK block;              /*Name of request block structure.   */
              short int retcode = SHVCLEAN; /*Expected return code from API.     */

              args    = (RXSTRING *) argv[1]; /*Receive parameters passed.       */
      /*Note that the other parameters received are ignored in this example.     */


      /*Now build the request block:                                            */
              block.shvnext = (SHVBLOCK *)0;/*Only one request block required (0) */
              block.shvname = args[0];             /*Name of variable to be changed. */
              block.shvvalue= args[1];             /*New value for this variable.    */
              block.shvcode = SHVSET;              /*Sub-function request.           */
              block.shvret  = SHVCLEAN;            /*Initialized to no errors.       */

              QREXVAR(&block,&retcode);            /* Call variable pool interface. */

      /*If the API has encountered any errors, 'retcode' will contain something  */
      /*different than the SHVCLEAN it was initialized with. In such a case       */
      /*the short int error flag that argv[4] is pointing to,                     */
      /*can be set to a nonzero  value in which case the interpreter will raise   */
      /*error 40 and set the SYNTAX condition in the calling REXX program.        */

              return;
          }
```

Example 2: The following is the REXX program for this example:

```
/*REXX program illustrating the actions to be taken to achieve the       */
/*requirements for the second example above.                             */
parm1 = '1stparm'                      /*Variable to be dropped.         */
parm2 = '2ndparm'                      /*Variable to get the new value.  */
p3 = 'Now changed to 1stparm'          /*Value to be assigned to 'parm2'. */
p4 = 'PARM3'                           /*Name of new variable - Note caps. */
p5 = 'This is now parm2'               /*Value for new variable.         */
say "The values of parm1, parm2, and parm3 before VPI changes are:"       */
say "parm1 = "parm1                    /*Displays '1stparm'.             */
say "parm2 = "parm2                    /*Displays '2ndparm'.             */
say "parm3 = "parm3                    /*Displays 'PARM3' - Still unknown. */

/*Call the QREXVAR API using the ILE/C program CCHGVAR and test the result.*/
call 'CCHGVAR' 'PARM1', 'PARM2', p3, p4, p5
say "The values of parm1, parm2, and parm3 after VPI changes are:"
say "parm1 = "parm1                    /*Displays 'PARM1' - Now unknown.  */
say "parm2 = "parm2                    /*Displays 'Now changed to 1stparm'. */
say "parm3 = "parm3                    /*Displays 'This is now parm2'.    */
ret_cde = Result                       /*Receive the result.             */
Say "Operation was a "ret_cde          /*Displays 'SUCCESS' or 'FAILURE'. */
```

Here is the ILE/C program CCHGVAR for this example:

```
/****************************************************************************/
/*This program expects pointers to five RXSTRINGS in argv[1]: The first one */
/*to the name of a REXX variable to be dropped from the VPI; the second to  */
/*the name of a variable of which the value must be changed to the value    */
/*pointed to by the third pointer; the fourth to the name of a new variable */
/*to be created with the value pointed to by the fifth pointer. Either      */
/*"SUCCESS" or "FAILURE" is returned to the REXX variable RESULT.           */
/****************************************************************************/

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

/*The following code declares the structure of an RXSTRING - This is a good */
/*candidate for an include header file.                                     */
    typedef struct                    /* RXSTRING structure definition    */
    { char         * rxstrptr ;       /* Pointer to data                  */
      unsigned long rxstrlen ;        /* Length of data                   */
    } RXSTRING ;

/*The following code declares a structure for a request block to the QREXVAR */
/*system API - Also a good candidate for an include header file.            */
    typedef struct shvnode            /* Shared V.P request block def.    */
    { struct shvnode     *shvnext;    /* Pointer to the next block        */
      RXSTRING           shvname;     /* Pointer to the name buffer       */
      RXSTRING           shvvalue;    /* Pointer to the value buffer      */
      unsigned char      shvcode;     /* Sub-function code for this block */
      unsigned char      shvret;      /* Individual Return Code Flag      */
    } SHVBLOCK;

/*The following code defines meaningful names to the sub-functions and      */
/*return codes of QREXVAR. Only those that will be used need to be defined.  */
/*A better idea might be to include the definitions for all sub-functions and*/
/*return codes in an include header file.                                   */
    #define SHVSET        0x00        /* Request to set a variable's value*/
    #define SHVDROPV      0x02        /* Drop a variable from the VPI.    */
    #define SHVEXTFN      0x09        /* Set exit code (value for RESULT).*/

    #define SHVCLEAN      0x00        /* The request ran OK (Return code).*/
    #define SHVNEWV       0x01        /* New variable - did not exist.    */

    #pragma linkage(QREXVAR,OS)       /* Pragma for the V.P API.          */

    RXSTRING empty_str = {(char *)0,0L}; /* Used when no value is needed.  */
    void QREXVAR(SHVBLOCK *, short int *);  /*Prototype for V.P API.       */
    RXSTRING crte_rx(char *);         /*Prototype to create a RXSTRING.   */
    short int result_via_vp(char *);  /*Prototype to create a req. block. */

    main(int argc, char *argv[])
    {
/*Only two of the parameters are received because the others are not used.   */

        RXSTRING *args;                       /*Pointers to received parms.  */
        short int *errflag;                   /*Success/Failure error flag.  */

/*The SHVNEWV return code is expected with the creation of the new variable  */
/*and the SHVCLEAN with the others.                                          */
        short int retcode = SHVCLEAN|SHVNEWV; /*Expected return code from API.*/
```

```
        short int recvd_retcode;              /*Retcode actually recvd in here*/
        SHVBLOCK drpv,chgv,crtv;              /*Names for blocks to be built. */

        args    = (RXSTRING *) argv[1];      /*Receive parameters passed    */
        errflag = (short int *) argv[4];     /*Pointer to the error flag    */

/*Better coding practice would have been to build the linked list of request */
/*blocks by way of a subroutine but the following method is used for clarity.*/
/*Now build req. blocks in reverse order to obtain pointers needed. First the*/
/*block for the new variable. Note pointer to next block = 0 ==> (last block)*/
        crtv.shvnext = (SHVBLOCK *)0;         /* This is the last request.   */
        crtv.shvname = args[3];               /* Name of to be created.      */
        crtv.shvvalue= args[4];               /* Value for new variable.     */
        crtv.shvcode = SHVSET;                /* Sub-function request.       */
        crtv.shvret  = SHVNEWV;               /* Expected return code.       */

/*The next code will build a request block to change the existing variable's */
/*value. It is pointing to the next block (the one previously built).        */
        chgv.shvnext = &crtv;                 /* Points to next block.       */
        chgv.shvname = args[1];               /* Name of var. to be changed. */
        chgv.shvvalue= args[2];               /* New value.                  */
        chgv.shvcode = SHVSET;                /* Sub-function request.       */
        chgv.shvret  = SHVCLEAN;              /* Expected return code.       */

/*The next code creates a request block to drop an existing variable.        */
        drpv.shvnext = &chgv;                 /* Points to next block.       */
        drpv.shvname = args[0];               /*Name of variable to be dropped*/
        drpv.shvvalue= empty_str;             /* No value needed.            */
        drpv.shvcode = SHVDROPV;              /* Sub-function request.       */
        drpv.shvret  = SHVCLEAN;              /* Expected return code.       */



        QREXVAR(&drpv,&recvd_retcode);        /* Call variable pool interface */

        if (retcode == recvd_retcode)         /* if aggregate retcode is bad  */
            *errflag = result_via_vp("SUCCESS"); /*Set RESULT variable in REXX*/
        else
            *errflag = result_via_vp("FAILURE"); /*Set RESULT variable in REXX*/
        return;
    }

/*The next subroutine builds a request block to place a value in the REXX    */
/*variable RESULT and submits the request to QREXVAR.                        */

    short int result_via_vp(char *result)
    { SHVBLOCK blk;                           /* Name the request block.     */
      short int expected_rc = SHVCLEAN;       /* Set expected return code.   */

/*Now build the request block ...                                            */
        blk.shvnext = (SHVBLOCK *)0;          /* This is the only block.     */
        blk.shvname = empty_str;              /* RESULT automatically set.   */
        blk.shvvalue= crte_rx(result);        /* RXSTRING format needed.     */
        blk.shvcode = SHVEXTFN;               /* Sub-function request.       */
        blk.shvret  = SHVCLEAN;               /* Expected return code.       */

        QREXVAR(&blk,&expected_rc);           /* Call the API.               */
```

```
            return expected_rc;                    /* Return return code.      */
/*If API was not successful, a nonzero code will be returned to errflag.      */
    }


/*The next subroutine creates a RXSTRING from the value its arg is pointed to*/
    RXSTRING crte_rx(char *str)
        { RXSTRING rx;                           /*Name a RXSTRING structure.  */
          rx.rxstrlen = strlen(str);             /*Store its length            */
          rx.rxstrptr = str;                     /*and the pointer to the string.*/
          return rx;                             /*Return the RXSTRING.        */
        }
```

Returning Results from the CL Command Environment

When an ILE/C program, called by the CL command environment on behalf of REXX, needs to return results to the REXX program, there are three ways it can be done:

- Using the variable pool interface (VPI)

- Using the REXX external data queue

- Setting values into the parameters with which the ILE/C program was called that were specified as pseudo-CL variables.

The following is an example of REXX code which calls an ILE/C program through the CL command environment. The REXX program passes two parameters: the name of a variable in the REXX program, and a character string which the ILE/C program is to assign to the variable named in the first parameter by using the variable pool interface.

```
/* REXX calling ILE/C through CL requesting VPI services.*/

parm1 = 'This is a string'      /* Creating an existing variable.   */
parm2 = 'PARM1'                 /* Store name of existing variable. */
parm3 = 'New value for parm1'   /* New value for existing variable. */

say "PARM1's value before change: "parm1 /*Displays 'This is a string'.*/

'call CCLCHG parm(&Parm2 &Parm3)' /* Command to CL command environment.*/
say 'Return code is: 'RC          /* RC contains 0 if successful.      */
say "PARM1's value after change: "parm1/*Displays 'New value for parm1'*/
```

The ILE/C program shown below is called from REXX using the CALL command. It expects the two parameters mentioned above and will call the QREXVAR API to change the value of the REXX variable PARM1.

```
/****************************************************************************/
/*This program is expecting only the name of the REXX variable to           */
/*be changed and the value to which it must be changed in argv[1]           */
/*and argv[2]. It is called through the CL command environment.             */
/****************************************************************************/

    #include <stdio.h>

    typedef struct                      /* RXSTRING structure definition.   */
    { char         * rxstrptr ;         /* Pointer to data.                 */
      unsigned long rxstrlen ;          /* Length of data.                  */
    } RXSTRING ;

    typedef struct shvnode              /* Shared V.P request block definition*/
    {   struct shvnode    *shvnext;     /* Pointer to the next block.       */
        RXSTRING          shvname;      /* Pointer to the name buffer.      */
        RXSTRING          shvvalue;     /* Pointer to the value buffer.     */
        unsigned char     shvcode;      /* Sub-function code for this block. */
        unsigned char     shvret;       /* Individual Return Code Flag.     */
    } SHVBLOCK;

    #define SHVSET          0x00        /* Request to set a variable's value. */
    #define SHVCLEAN        0x00        /* The request ran OK (Return code). */

    #pragma linkage(QREXVAR,OS)         /* Pragma for the V.P API.          */
    void QREXVAR(SHVBLOCK *, short int *); /*Prototype for V.P API.         */
    RXSTRING crte_rx(char *str);        /*Prototype for internal function.  */

    main(int argc, char *argv[])
    {
        SHVBLOCK block;                     /*Name of request block structure.   */
        short int retcode = SHVCLEAN;       /*Expected return code from API.     */

        block.shvnext = (SHVBLOCK *)0;      /*Only one request block required (0).*/
        block.shvname = crte_rx(argv[1]);   /*Name of variable to be changed.    */
        block.shvvalue= crte_rx(argv[2]);   /*New value for this variable.       */
        block.shvcode = SHVSET;             /*Sub-function request.              */
        block.shvret  = SHVCLEAN;           /*Initialized to no errors.          */

        QREXVAR(&block,&retcode);           /* Call variable pool interface.     */

        if (retcode != SHVCLEAN)            /* If API was not successful...      */
           printf("Variable pool failed with %hd\n",retcode); /* Notify User     */
        return;
    }
/*The next subroutine creates a RXSTRING from the value its arg is pointed to. */
    RXSTRING crte_rx(char *str)
    { RXSTRING rx;                      /*Name a RXSTRING structure.        */
      rx.rxstrlen = strlen(str);        /*Store its length.                 */
      rx.rxstrptr = str;                /*and the pointer to the string.    */
      return rx;                        /*Return the RXSTRING.              */
    }
```

Returning Results in the REXX External Data Queue

The REXX external data queue is nothing more than a large buffer (the maximum size is about 15.5MB) in which REXX places any data it chooses to transmit to other programs or from which it retrieves any data placed there by those programs. It may of course store data temporarily on the queue and retrieve it from there itself later when needed.

The REXX external data queue is created when a job begins and remains active until the job has finished. All programs running in the same job have access to the REXX external data queue which was created for that job. The queue for a particular job is not visible to any other job on the system.

In order to extend the flexibility of REXX programs, the queue may be subdivided into a number of logical buffers. The new CL command ADDREXBUF has been provided for this purpose. Also, the new CL command RMVREXBUF is provided to remove buffers created by the ADDREXBUF command, or to clear all data from the queue.

The maximum size of a single data item on the REXX external data queue is limited to 32,767 bytes. No character placed on the queue has any special meaning or effect—data items are placed and retrieved as complete lines. The data in the REXX external data queue can be regarded as variable length records.

REXX places data on the REXX external data queue with its PUSH and QUEUE instructions and retrieves it from there with its PULL instruction. See the *REXX/400 Reference* for a description of these instructions.

ILE/C accesses the REXX external data queue by way of the supplied system API named QREXQ in the QSYS library, which expects five parameters. They are:

1. An indication of the service required (like Add or Pull)

2. A data buffer containing the data to be added or to receive the data pulled

3. Additional information needed by the API (mainly used to indicate the length of the data string to be added to or pulled from the queue)

4. An operation flag

5. The address of a one byte character in which QREXQ will place a return code to indicate success or failure of its operation

For a detailed description of the parameters required by this API, see the *REXX/400 Reference.*

An example of how the services of the REXX external data queue are used is provided under the heading "Example Using the REXX External Data Queue" on page 194.

Example Using the REXX External Data Queue

This appendix is concluded with a practical example of how to use the REXX external data queue as communication vehicle between REXX and ILE/C.

Problem Description: You want to be automatically reminded of your valuable customers' (or maybe your wife's) birthday every morning when you sign on, if it falls within the next 7 calendar days. You also want a choice to skip the function if you cannot spare the time for it on any specific day. You also want to add names and birthdays to the file where the information is kept.

Limitations of the Program: To limit the size of this example, the following restrictions will be used:

- In all prompts where a yes or no answer is required, a `y` or `Y` will indicate the affirmative while anything else will indicate a no answer.

- Names must be two words (no more, no less) and will be truncated without warning if more than 20 characters in total are used.

- Birth dates must be given in the format MMDD (for example, `0214` means the 14th of February).

Data is stored in a physical file with 25-byte records. The REXX program could be called from an initial program as specified in the AS/400 User Profile. The ILE/C program called by REXX only reads records from the file and places them on the REXX external data queue or writes the records placed there by the REXX program. However, it returns the string `updated with your input` if the database file was successfully updated with the records supplied by the REXX program. It also displays any error codes received from the QREXQ API. It is called as an external subroutine and receives its commands (for example, whether to read or to write, the file's name, and whether to add the records read to the head or the tail of the queue) from a parameter list specified on the REXX CALL instruction. Most logic and decision making will be done in the REXX program.

The following is an example of the REXX code that may be used:

```
/* */
say "Want to leave?"          /*Escape opportunity because program was started  */
                              /*automatically.                                  */
pull ans; if ans = 'Y' then exit  /*Input in uppercase automatically            */
signal on SYNTAX              /*Branch to syntax: if retcode ¬= 0.              */
p2 = '0'                      /*4th parameter required by QREXQ API.            */
p3 = "REXXLIB/BIRTHF"         /*Database file containing names and birthdays.   */
say "Do you want to add more entries to your Birthday File? (Y/N)"
pull ans; if ans = 'Y' then do /*If User wants to add records...               */
   p1 = 'P'                   /*First parameter required by QREXQ API.          */
   bf. = ''                   /*Buffer to hold new records.                     */
   do forever                 /*Do not leave this loop until a valid answer     */
       say "How many entries do you want to add?"   /* is received.            */
       pull ans; if datatype(ans) = 'NUM' then leave /*Note free format: 2      */
       else say "Numeric value required: Please try again." /*clauses per line*/
   end                        /*End forever loop.                               */
   do i = 1 to ans            /*Repeat loop for every addition.                 */
      do forever              /*Secondary loop to obtain name.                  */
         say 'Please give the first and last name of the person to be added'
         parse pull name      /*Parse, do not translate input to uppercase.     */
         if words(name) = 2 then leave /*Two and only 2 names required.         */
         else say'Two separate words are required: Please try again.'
      end                     /*End forever loop.                               */
      do forever              /*Secondary loop to obtain birthday.              */
         say "Pse supply the person's birthday - (MMDD only)"
         pull date            /*Numeric data expected, 'parse' not involved.    */
         mon = substr(date,1,2)   /*First 2 characters are the month.           */
         day = substr(date,3,2)   /*Last 2 characters are the day.              */
         if datatype(date) = 'NUM' & length(date) = 4 & day > 0 &, /*Note       */
            day < 32 & mon > 0 & mon < 13 then leave  /*2 lines per clause.     */
         else do
              say "Input not valid - Input must be in the format MMDD"
              say "(for example, 0214 for the 14nd of February)"
              say "Please try again"
         end                  /*End else.                                       */
      end                     /*End forever.                                    */
   bf.i = left(name,20)||left(mon'/'day,5) /*Format input record.               */
   end                        /*End prompting loop.                             */
   do i = 1 to ans            /*Place input records one by one on the           */
      queue bf.i              /*  External Data Queue.                          */
   end
   call 'CREAD' P1, P2, P3   /*Call ILE/C program as subrtn. to write records. */
   say p3 result             /*Displays 'REXLIB/BIRTHF updated with your input'.*/
end                          /*End input loop.                                 */

/* Now go and read the file so that the required records can be obtained.       */
p1 = 'A'                      /*First parameter required by QREXQ API.          */
call 'CREAD' p1, p2, p3      /*Call ILE/C program to read the file.            */
leap = 0                      /*Leap year switch set to off.                    */
days = 0 31 59 90 120 151 181 212 243 273 304 334 /*Cumulative number of days
                          each month for a normal year.                         */
ref = date(d)                /*Number of days so far this year.                */
yr = substr(date(s),1,4)    /*Current year; i.e. 1990.                        */
if (yr // 4 = 0 & yr // 100 ¬= 0) | yr // 400 = 0 then /*Is this a leap year? */
do                           /*If so...                                        */
  days = 0 31 60 91 121 152 182 213 244 274 305 335 /*Redefine cumulative
```

```
                                      number of days each month for a leap year.       */
      leap = 1                        /*Set leap year switch on.                        */
   end
   bf. = ''; j = 1                    /*Initialize compound variable bf. to hold the
                                          records we are interested in and j as a qualifier
                                          for bf.                                       */
   do queued()                        /*Repeat loop for number of entries in queue.     */
     parse pull rec                   /*Pull a record from the queue - no uppercase     */
                                      /*   translation.                                 */
     bdat = subword(rec,3,1)          /*Birthdate is third word in record (MM/DD).      */
     bdatd = subword(days,substr(bdat,1,2),1) + substr(bdat,4,2) /*Calculate the
                                          number of days from the beginning of the year. */
     if ref - leap > 358 & bdatd < 7 then bdatd = bdatd + 365 + leap /*Make
                                          provision for a year end                      */
     if bdatd >= ref & bdatd <= ref+7 then /*Is birthday within 7 days from now? */
     do                               /*If so...                                        */
       bf.j = rec                     /*Place records in successive elements of bf.     */
        j = j + 1                     /*Step bf.'s qualifier.                           */
     end                              /*End simple do-loop.                             */
   end                                /*End queued() do-loop.                           */
   If bf.1 = '' then                  /*If first element of bf. is still empty,         */
     say "There are no birthdays in the next 7 days." /*notify the User.               */
   else do                            /*Else ....                                       */
     say '   Birthdays today and for the next 7 days:' /*Display report heading. */
     do i = 1 to j - 1                /*Display qualifying                              */
        say bf.i                      /*                  records                        */
     end                              /*                            one by one.         */
   end                                /*End else loop.                                  */
   return                             /*Exit program under normal conditions.           */
   syntax:                            /*Subroutine to be run if interpreter raised      */
                                      /*the syntax condition. Recovery actions taken here*/
                                      /*will depend on how the ILE/C program is coded    */
                                      /*In this example the User is notified and a       */
                                      /*graceful exit is made.                           */
   say "The ILE/C program experienced an unknown error condition."
   say "Activities are ended."
   exit
```

The following is an example of the ILE/C code that may be used:

```
/*****************************************************************************/
/*This program receives the following parameters from the BDTE REXX pgm:     */
/*1. An 'A' if it must read a file or a 'P' if it must write to a file        */
/*2. A '1' if data must be added to the head of the Q; a '0' for all others   */
/*3. The name of the file member concerned                                   */
/*****************************************************************************/
    #include <stdio.h>
    #include <string.h>

    typedef struct                    /* RXSTRING structure definition.      */
    { char          * rxstrptr ;      /* Pointer to data.                    */
      unsigned long rxstrlen ;        /* Length of data.                     */
    } RXSTRING ;                                                            /*    */

    typedef struct shvnode            /* Shared V.P request block def.       */
    { struct shvnode    *shvnext;     /* Pointer to the next block.          */
      RXSTRING          shvname;      /* Pointer to the name buffer.         */
```

```
        RXSTRING          shvvalue;   /* Pointer to the value buffer.    */
        unsigned char     shvcode;    /* Sub-function code for this block. */
        unsigned char     shvret;     /* Individual Return Code Flag.    */
   } SHVBLOCK;

   #define SHVEXTFN        0x09        /* Set exit code (value for RESULT). */
   #define SHVCLEAN        0x00        /* The request ran OK (Return code). */
   #define BUFLEN 26                   /* Expected length of records + 1.   */
   #define DO_FOREVER for(;;)          /* Just for fun.                   */

   #pragma linkage(QREXVAR,OS)         /* Pragma for the V.P API.         */
   #pragma linkage(QREXQ,OS)           /* Pragma for the Ext. Data Q. API.  */

   RXSTRING empty_str = {(char *)0,0L}; /* Used when no value is needed.   */

   void QREXVAR(SHVBLOCK *, short int *);  /* Prototype for V.P API.       */
   void QREXQ(char *, char *,unsigned long int *, char *, unsigned char *);
   RXSTRING crte_rx(char *);           /* Prototype to create a RXSTRING.  */
   short int result_via_vp(char *);    /* Prototype to create a req. block. */
   main(int argc, char *argv[])
   {  RXSTRING *args;                  /* Points to parameters received.   */
      short int *errflag;              /* Points to error flag.           */
      char line[BUFLEN];               /* Buffer to hold file records.    */
      unsigned long int rec_len;       /* Length of record read from file.  */
      unsigned char ret_cde;           /* Space for QREXQ's return code.   */
      FILE *fle;                       /* File pointer.                   */
      char fname[35];                  /* Maximum length for qualified fname.*/
      unsigned char function;          /* Read or write indicator.        */
      unsigned char flag;              /* Data to head or tail of Q.      */

/*Note that only the arguments to be used are received.                   */
      args    = (RXSTRING *) argv[1]; /* Receive parameters passed.       */
      errflag = (short int *) argv[4]; /* Pointer to interpreter's err flag*/

      function = *(args[1].rxstrptr); /* Must we read or write.           */
      flag = *(args[1].rxstrptr);     /* Data to head or tail of Q.       */
      if (flag == '0')                /* Change to acceptable             */
          flag = '\0';                /*                    format.       */
      memcpy(fname,args[2].rxstrptr,args[2].rxstrlen); /* Get file name.   */
      fname[args[2].rxstrlen] = '\0'; /* Add null character to file name.  */
      if (function == 'A')            /* If file read is requested ...    */
      {  if (( fle=fopen(fname,"rb, type=record")) == NULL) /* Open file.  */
         {  printf("Cannot open input file\n");  /* Notify User if        */
            return;                   /* problems are encountered.        */
         }                            /* End fopen-if.                    */
         DO_FOREVER                   /* Keep on reading to EOF.          */
         {  rec_len = fread(line,1,BUFLEN,fle); /* Record read into 'line'.*/
            if (!feof(fle))           /* If not yet EOF, place 'line' on   */
            {  QREXQ(&function, line, &rec_len, &flag, &ret_cde); /* the Q.*/
               if (ret_cde != 0)      /* If bad return code from QREXQ,   */
               {  fclose(fle);        /* close file.                     */
                  printf("QREXQ return code = %c : Push operation failed\n",
                        ret_cde);     /* Notify User.                    */
                  return;             /* Return to REXX.                 */
               }                      /* End bad-return-code-if           */
            }                         /* End EOF if                       */
            else                      /* If EOF is reached ...            */
            {  fclose(fle);           /* close file.                     */
```

```
                          return;                 /* Return to REXX.                    */
                    }                              /* End else.                          */
                }                                  /* End DO_FOREVER.                    */
            }                                      /* End function = 'A'.                */
       else if (function == 'P')      /*If file write operation is requested*/
         {  if ((fle = fopen(fname, "ab, type=record")) == NULL) /* Open file */
            {   printf("Cannot open output file\n"); /* Notify User if                   */
              return;                              /* problems are encountered.          */
            }                                      /* End fopen-if.                      */
         DO_FOREVER                                /* Else do until buffer is empty.     */
         {  rec_len = BUFLEN;                      /* Expected record length.            */
            QREXQ(&function, line, &rec_len, &flag, &ret_cde); /* Pull rec.*/
            if (ret_cde == 0)          /* If successful ...                   */
               fwrite(line, rec_len, 1, fle); /* Write record in 'line'       */
            else if (ret_cde == 2)     /* Q became empty                      */
               {  *errflag = result_via_vp("updated with your input");
                  fclose(fle);         /* Close the file.                     */
                  return;              /* Return to REXX.                     */
               }                        /* End else-if                         */
            else                        /* If errors were encountered          */
                                        /*                                     */
            {  fclose(fle);             /* Close file.                         */
               printf("QREXQ return code = %c : Pull operation failed\n",
                       ret_cde);        /* Notify User.                        */
               return;                  /* Return to REXX.                     */
            }                            /* End else                            */
         }                               /* end DO_FOREVER                      */
      }                                  /* end function = 'P'                  */
   }                                     /* end main                            */
/*The next subroutine builds a request block to place a value in the REXX     */
/*variable RESULT and submits the request to QREXVAR.                         */

    short int result_via_vp(char *result)
    {  SHVBLOCK blk;                     /* Name the request block.            */
       short int expected_rc = SHVCLEAN; /* Set expected return code.          */

/*Now built the request block ...                                             */
       blk.shvnext = (SHVBLOCK *)0;      /* This is the only block             */
       blk.shvname = empty_str;          /* RESULT automatically set.          */
       blk.shvvalue= crte_rx(result);    /* RXSTRING format needed.            */
       blk.shvcode = SHVEXTFN;           /* Sub-function request.              */
       blk.shvret  = SHVCLEAN;           /* Expected return code.              */

       QREXVAR(&blk,&expected_rc);       /* Call the API                       */
       return expected_rc;               /* return Return code                 */
/*If API was not successful, a nonzero code will be returned to errflag.      */
    }

/*The next subroutine creates a RXSTRING from the value its arg is pointed to*/
    RXSTRING crte_rx(char *str)
    {  RXSTRING rx;                       /*Name a RXSTRING structure.         */
       rx.rxstrlen = strlen(str);         /*Store its length                   */
       rx.rxstrptr = str;                 /*and the pointer to the string.     */
       return rx;                         /*Return the RXSTRING.               */
    }
```

Appendix H. Communication Between REXX/400 and Other Languages

Using the REXX External Data Queue API

The REXX external data queue API (QREXQ) can also be used by programs that are not written in ILE/C. In this section, sample programs in OPM (Original Program Module) RPG and OPM COBOL are used to illustrate the use of QREXQ.

For more information about QREXQ, see the *REXX/400 Reference*.

Pushing Data from RPG into the Queue

RPGPSHQ is a small RPG program which reads records from the file BIRTHF, and pushes them into the REXX external data queue through QREXQ.

The following is the source listing of RPGPSHQ:

```
H*   This RPG program reads records from a file and then use
H*    QREXQ to pushes the data into REXX external data queue.
H*
FBIRTHF  IF  E                    DISK
I* This file only has two fields : NAME and BDATE
I*
IBFLDS      DS
I                              B   1   40BUFLEN
I                              B   5   60FLAG
I                              B   7   80RCODE
C*
C                READ BIRTHF                      90
C        *IN90   DOWEQ'0'
C                MOVELNAME    BUF    25
C                MOVE BDATE   BUF
C                MOVE 'A'     FUNT   1       * Push into q
C                Z-ADD25      BUFLEN
C                Z-ADD0       FLAG          * FIFO
C*               Z-ADD1       FLAG          * LIFO
C                Z-ADD0       RCODE
C*
C                CALL 'QREXQ'
C                PARM         FUNT
C                PARM         BUF
C                PARM         BUFLEN
C                PARM         FLAG
C                PARM         RCODE
C*
C                READ BIRTHF                      90
C                END
C*
C                SETON                    LR
C*
        * * * * *  E N D   O F   S O U R C E   * * * * *
```

This is the DDS of the file BIRTHF:

```
A               R BIRTHREC
A                 NAME          20          COLHDG('Name')
A                 BDATE          5          COLHDG('Date MM/DD')
```

The logic of this program is very simple. After reading a record from the file, it sets up the parameter list and then pushes the record into the REXX external data queue using QREXQ. This process is repeated until all records are pushed into the queue.

However, it should be noted that the buffer length *(3rd parameter)*, the operation flag *(4th parameter)*, and the return code *(5th parameter)* should be defined as binary fields in the Input specification.

Note: The REXX program in "Example Using the REXX External Data Queue" on page 194 can be changed to call RPGPSHQ instead of the ILE/C program CREAD to read the BIRTHF file and place its contents in the REXX external data queue.

Updating the File from the Queue by RPG

RPGUPD uses QREXQ to pull data from the REXX external data queue, and then updates the BIRTHF file according to the action code which is added to the end of each pulled data record.

The logic of this RPG program is as follows:

1. Use QREXQ to pull the first record from the queue, which contains the number of updated entries in the queue.

2. Use QREXQ to pull a data line from the REXX queue

3. Look at the action code which is added to the end of each data record, and then add, delete, or write the record back to the file.

4. Repeat steps 2 & 3 until all updated entries are pulled out.

The following is the source listing of RPGPUPD:

```
H*  This RPG program uses the data in the REXX external data
H*  queue to update the BIRTHF file.
F*
FBIRTHF  UF  E                    DISK                          A
I* This file only has two fields : NAME and BDATE
I*
IBFLDS       DS
I                                 B   1   40BUFLEN
I                                 B   5   60FLAG
I                                 B   7   80RCODE
IBUF         DS
I                                     1  20 NWNAME
I                                    21  25 NWDATE
I                                    26  26 ACTCDE
C*
C                 Z-ADD0        FLAG            * Not used
C                 Z-ADD0        RCODE           * Reset RCODE
C                 MOVE 'P'      FUNT     1      * Pull from q
C*
C                 Z-ADD10       BUFLEN
C                 CALL 'QREXQ'
```

```
C                          PARM            FUNT
C                          PARM            ENTNOA 10
C                          PARM            BUFLEN
C                          PARM            FLAG
C                          PARM            RCODE
C*
C                          MOVELENTNOA     ENTNO   40      * Ent. to pull
C*
C                          DO    ENTNO
C                          Z-ADD30         BUFLEN          * Set buf length
C                          CALL 'QREXQ'
C                          PARM            FUNT
C                          PARM            BUF
C                          PARM            BUFLEN
C                          PARM            FLAG
C                          PARM            RCODE
C*
C          ACTCDE    IFEQ 'A'                              * If add
C                          MOVE NWNAME     NAME
C                          MOVE NWDATE     BDATE
C                          WRITEBIRTHREC
C                          ELSE
C                          READ BIRTHF                     90
C          ACTCDE    IFEQ 'D'                              * If delete
C                          DELETBIRTHREC
C                          ELSE                            * Otherwise
C                          MOVE NWNAME     NAME
C                          MOVE NWDATE     BDATE
C                          UPDATBIRTHREC
C                          END
C                          END
C                          MOVE *BLANK     ACTCDE
C*
C                          END
C*
C                          SETON                           LR
C*
       * * * * *  E N D  O F  S O U R C E  * * * * *
```

Note: Since the buffer length (3rd parameter) is changed by QREXQ to the actual length of the data being placed in the data buffer after each call, this parameter should be reset each time before QREXQ is called.

Moreover, it is a good practice to set a larger buffer length for the QREXQ call than you expect, and then only use the number of bytes indicated by the buffer length parameter.

The following is another example of a REXX program which prompts the users for changes that should be made to the BIRTHF file. It calls RPGPSHQ to read the file and put it into the REXX external data queue, then later calls RPGUPD to update the file.

```
                 /*******************************************************************/
                 /*  REXX program which allows the user to change or add            */
                 /*  records to the data file, BIRTHF.  It calls RPGPSHQ program    */
                 /*  to get all the data from the file, and then uses RPGUPD to     */
                 /*  update the file from the REXX external data queue.             */
                 /*                                                                 */
                 /*******************************************************************/

                 /* Setup ERROR, FAILURE, and SYNTAX condition traps.*/
                 signal on error name command_error
                 signal on failure name command_error
                 signal on syntax name syntax_error

                 /* Save number of existing entries in the queue.*/
                 prev_entries = queued()

                 /* Create a new buffer for this program.*/
                 bufno = 0
                 'ADDREXBUF BUFFER(&bufno)'

                 /* Use RPGPSHQ to get all records into the queue.*/
                 'CALL RPGPSHQ'

                 /* Put all queue entries into compound variable DATA.*/
                 total_recs = queued() - prev_entries
                 do count = 1 to total_recs
                    pull data.count
                    say count') 'data.count
                 end

                 /* Prompt for changing or deleting records */
                 do until answer = 'N'
                    say 'Do you want to change/delete any record? (Y/N)'
                    parse upper linein answer
                    if answer = 'Y' then do
                       do until rec_no = ''
                          say 'Change/delete which record?'
                          parse linein rec_no
                          if rec_no ¬= ''  then
                             call change_rec
                       end
                    end
                 end
                 /* Prompt for adding new records.*/
                 answer = ''
                 do until answer = 'N'
                    say 'Do you want to add a new record? (Y/N)'
                    parse upper linein answer
                    if answer = 'Y' then do
                       say 'Please enter Name","Birthday(mm/dd):'
                       parse upper linein name ',' birth_date .
                       total_recs = total_recs +1
                       data.total_recs = overlay(name, data.total_recs,1,20)
                       data.total_recs = overlay(birth_date,data.total_recs,21,5)
                       /*  Add the action code 'A' to the end of data */
                       data.total_recs = data.total_recs||'A'
                    end
                 end
```

```
                    /* Right-justified total_recs into a 4-byte field */
                    lead_blanks = 4 - length(total_recs)
                    total_recs = copies(' ',lead_blanks)||total_recs
                    data.0 = total_recs

                    /* Push records back to queue, starting from last record.*/
                    do count = total_recs to 0 by -1
                       if data.count ¬= ''  then
                          queue data.count
                    end

                    /* Call RPGUPD to update the file.*/
                    'CALL RPGUPD'

                    /* Remove the buffer created.*/
                    'RMVREXBUF BUFFER(&bufno)'

                    /* End of Program. */
                    exit


                    /*******************************************************************/
                    /*   change_rec: Change a record in the REXX queue                 */
                    /*******************************************************************/
                    change_rec:

                      if rec_no > total_recs then
                         say 'Record does not exist, please try again.'
                      else do
                         say rec_no') ' data.rec_no
                         say 'Enter DELETE or changes: Name","Birthday(mm/dd)'
                         parse upper linein name ',' birth_date
                         if name = 'DELETE' then
                            /*  Add the action code 'D' to the end of data.*/
                            data.rec_no = data.rec_no||'D'
                         else do
                            if name ¬= '' then
                               data.rec_no = overlay(name,data.rec_no,1,20)
                            if birth_date ¬= ' ' then
                               data.rec_no = overlay(birth_date,data.rec_no,21,5)
                            /*  Add the action code 'C' to the end of data.*/
                            data.rec_no = data.rec_no||'C'
                         end
                      end

                    return

                    /*******************************************************************/
                    /*   command_error : ERROR & FAILURE condition trap                */
                    /*******************************************************************/
                    command_error:

                      parse source system start srcmbr srcfile srclib
                      say 'Unexpected error at line 'sigl' of REXX program 'srcmbr',
                          ' in 'srclib'/'srcfile'.  The exception id is 'rc'.'
                      'RMVREXBUF BUFFER(&bufno)'
```

```
         exit(rc)

         /*******************************************************************/
         /*  syntax_error : Syntax condition trap                          */
         /*******************************************************************/
         syntax_error:

            parse source system start srcmbr srcfile srclib
            say 'Syntax error at line 'sigl' of REXX program 'srcmbr,
                ' in 'srclib'/'srcfile'.  The error code is 'rc'.',
                ' The description of the error is :'
            say errortext(rc)
           'RMVREXBUF BUFFER(&bufno)'

         exit(rc)
```

Pushing Data from COBOL into the Queue

CBLPSHQ is a simple COBOL program which has the same function as
RPGPSHQ in "Pushing Data from RPG into the Queue" on page 199. CBLPSHQ
reads records from the BIRTHF file, and then pushes them into the REXX external
data queue using QREXQ.

It should be noted that when defining the parameter list for QREXQ, the buffer
length (3rd parameter), the operation flag (4th parameter), and the return code (5th
parameter) should be defined as COMP-4 (for example, binary field).

This is the source listing of CBLPSHQ:

```
      PROCESS  XREF APOST.
       IDENTIFICATION DIVISION.
       PROGRAM-ID.  CBLPSHQ.
       AUTHOR.  ITSC-ROCHESTER.
      ****************************************************************
      *                                                            *
      *  PROGRAM FOR USING QREXQ TO PUSH DATA INTO REXX EXTERNAL    *
      *  DATA QUEUE.                                                *
      *                                                            *
      ****************************************************************
       ENVIRONMENT DIVISION.
       CONFIGURATION SECTION.
       SOURCE-COMPUTER.    IBM-AS400.
       OBJECT-COMPUTER.    IBM-AS400.
      *
       INPUT-OUTPUT SECTION.
       FILE-CONTROL.
          SELECT  BIRTH-FILE
                  ASSIGN TO        DATABASE-BIRTHF
                  ORGANIZATION IS  SEQUENTIAL.
      /
       DATA DIVISION.
       FILE SECTION.
       FD  BIRTH-FILE
           LABEL RECORDS ARE OMITTED.
       01  BIRTH-REC.
      *  This file only has two fields : NAME and BDATE.
           COPY DDS-ALL-FORMATS  OF  BIRTHF.
      /
```

```
           WORKING-STORAGE SECTION.
           01  PARM-LIST.
               05  FUNCTION-CODE          PIC X.
               05  BUFFER.
                   10  NAME               PIC X(20).
                   10  BDATE              PIC X(5).
               05  BUF-LENGTH             PIC 9(5) COMP-4.
               05  FLAG                   PIC 99   COMP-4.
                   88 FIFO                VALUE '0'.
                   88 LIFO                VALUE '1'.
               05  RETURN-CODE            PIC 99   COMP-4.
       *
        01  STATUS-FLAG                   PIC X.
               88 END-OF-FILE             VALUE 'Y'.
       /
        PROCEDURE DIVISION.
       **************************************************************
        MAINLINE-ROUTINE.
       **************************************************************
           OPEN    INPUT    BIRTH-FILE.
       *
           PERFORM READ-FILE.
           PERFORM PUSH-INTO-QUEUE
               UNTIL  END-OF-FILE.
       *
        END-PGM.
           CLOSE            BIRTH-FILE.
           STOP RUN.
       *
       /****************************************************************
        PUSH-INTO-QUEUE.
       ****************************************************************
       *
           MOVE    'A'               TO     FUNCTION-CODE.
           MOVE    CORR BIRTHREC     TO     BUFFER.
           MOVE    25                TO     BUF-LENGTH.
           SET     FIFO              TO     TRUE.
           MOVE    ZEROS             TO     RETURN_CODE.
       *
       *    When queuing data into the REXX queue
       *    SET    LIFO              TO     TRUE.
       *
           CALL    'QREXQ'                USING  FUNCTION-CODE
                                          BUFFER BUF-LENGTH FLAG RETURN-CODE.
       *
           PERFORM  READ-FILE.


       /****************************************************************
        READ-FILE.
       ****************************************************************
       *
           READ    BIRTH-FILE   AT   END
             SET END-OF-FILE         TO      TRUE.
       *
             * * * * *  E N D   O F   S O U R C E   * * * * *
```

Overriding STDIN and STDOUT

In a REXX program, the display is usually used as the input and output device. In fact, you can also have direct input from a database file and output to a spooled output file in a REXX program. All these can be done by using the CL override commands to override the standard streams STDIN and STDOUT.

DSPUNUSE is an example of a REXX program which displays a list of objects in a library that have not been used after a certain specified date. It uses the DSPOBJD command to output all the objects in the library to a database file (OBJDPF), and then select those expired objects for display or print out.

Because all the fields in OBJDPF are not needed, STDIN is overridden to a logical file (OBJDLF) of this physical file, in which only those fields we are interested in are selected. If you are working with packed decimal fields in a file, you can also make use of the logical file to do the conversion for you instead of converting the packed fields in the REXX program.

Similarly, DSPUNUSE produces output to a printer file by using the OVRPRTF command to override STDOUT to QPRINT.

Because of the way standard streams are opened and committed to the REXX program call, it should be noted that these CL override commands must be issued before the first use of each stream in the REXX program, and these overrides cannot be removed until that call to the interpreter ends. In other words, STDIN must be overridden before the first PULL instruction that tries to read from the display and before any PARSE LINEIN instructions, and STDOUT must be overridden before any SAY instructions are run. If this causes a problem, the override commands and the corresponding SAY or PARSE LINEIN statements can be included in a separate REXX program, which will then be called using STRREXPRC or as a command from the main REXX program.

Moreover, you should also be aware that the program will not work properly when using interactive trace, because STDIN cannot be overridden when a REXX program is being interactively traced.

For more information about STDIN and STDOUT, see the *REXX/400 Reference*.

The following is the REXX program DSPUNUSE:

```
/********************************************************************/
/*  REXX program as the CPP of DSPUNUSE command.                    */
/*  DSPUNUSE displays a list of unused objects in a library after   */
/*  a particular date.                                              */
/*                                                                  */
/*     Parameter passed:                                            */
/*                                                                  */
/*         Lib         :  Library name                              */
/*         Type        :  Object Type                               */
/*         Date        :  Last used date (yymmdd)                   */
/*         Output      :  *PRINT - Output to listing                */
/*                        *     - Output to display                 */
/*                                                                  */
/********************************************************************/

/* Parse out the library value from the CDO 'LIBRARY' keyword.*/
arg 'LIBRARY('lib')'

/* Parse out the type value from the CDO 'TYPE' keyword.*/
arg 'TYPE('type')'

/* Parse out the last_date value from the CDO 'LASTUSE' keyword.*/
arg 'LASTUSE('last_used_date')'

/* Parse out the output type value from the CDO 'OUTPUT' keyword.*/
arg 'OUTPUT('output')'

/* Check if the library exists.*/
'CHKOBJ OBJ(QSYS/'lib') OBJTYPE(*LIB)'

if pos('CPF98',rc) ¬= 0  then do
   message = 'Library:' lib 'not found'
   'SNDUSRMSG MSG(&message)'
    exit(rc)
end

/* Setup ERROR, FAILURE, and SYNTAX condition traps.*/
signal on error name command_error
signal on failure name command_error
signal on syntax name syntax_error

/* Put all the objects information into a file.*/
'DSPOBJD OBJ('lib'/*all)  OBJTYPE(*ALL)  DETAIL(*FULL)',
'OUTPUT(*OUTFILE)  OUTFILE(QGPL/OBJDPF)'

/*  Get the number of records in QGPL/OBJDPF.*/
reccnt = 0
'RTVMBRD FILE(QGPL/OBJDPF) MBR(*FIRST) NBRCURRCD(&reccnt)'

/*  Override STDIN to the logical file QGPL/OBJDLF.         */
/*  This must be done before the first use of STDIN,       */
/*  for example, PARSE LINEIN, or PULL.                    */
'OVRDBF FILE(STDIN) TOFILE(QGPL/OBJDLF)'

/*  Select output device - default is display.             */
/*  This must be done before the first use of STDOUT, for  */
```

```
           /*  example SAY                                              */
           if output = '*PRINT'  then
              'OVRPRTF  FILE(STDOUT)  TOFILE(QPRINT)'
           /* Set variables to initial values.*/
              line_count = 24
              page_no = 0
              rec_count = 0
              detail_line = ''

           /* Parse out the record from the queue into the following variables.*/
           do reccnt
             parse  linein              obj_name    ,    /* Object name.    */
                                11      obj_type    ,    /* Object type.    */
                                19      obj_atb     ,    /* Object attribute.*/
                                29      used_mdy    ,    /* Last used date.  */

              used_date = substr(used_mdy,5,2)||substr(used_mdy,1,4)
              if ((type = '*ALL') | (obj_type = type)) &,
                 (used_date < last_used_date) then
                 call write_line
           end

           /* Write total number of objects listed.*/
           call write_total

           /* End Program.*/
           Exit

           /*******************************************************************/
           /*  write_line : write a line of the selected object              */
           /*******************************************************************/
           write_line:

             /* Check for new page.*/
             if line_count ¬< 24  then do
                call write_heading
             end

             line_count = line_count + 1
             rec_count = rec_count + 1

             detail_line = overlay(obj_name,detail_line,4,10)
             detail_line = overlay(obj_type,detail_line,19,8)
             detail_line = overlay(obj_atb,detail_line,34,10)
             detail_line = overlay(used_date,detail_line,54,6)
             say detail_line

           return

           /*******************************************************************/
           /*  write_heading : write a new page heading                      */
           /*******************************************************************/
           write_heading:

             page_no = page_no + 1
             line_count = 3
             say,
             '                    Display Unused Objects           Page : 'page_no
```

```
 say,
 '   Object name    Object Type    Object Attribute    Last Used Date'
 say,
 '   -----------    -----------    ----------------    --------------'

return

/********************************************************************/
/*  write_total : write the total no. of object listed             */
/********************************************************************/
write_total:

  say ' '
  say rec_count' objects with type 'type' in library 'lib' were',
      ' unused since 'last_used_date'.'
  say ' '

return

/********************************************************************/
/*  command_error : ERROR & FAILURE condition trap                 */
/********************************************************************/
command_error:

  parse source system start srcmbr srcfile srclib
  say 'Unexpected error at line 'sigl' of REXX program 'srcmbr,
      ' in 'srclib'/'srcfile'.  The exception id is 'rc'.'

exit(rc)

/********************************************************************/
/*  syntax_error : Syntax condition trap                           */
/********************************************************************/
syntax_error:

  parse source system start srcmbr srcfile srclib
  say 'Syntax error at line 'sigl' of REXX program 'srcmbr,
      ' in 'srclib'/'srcfile'.  The error code is 'rc'.',
      ' The description of the error is :'
  say errortext(rc)

exit(rc)
```

The following is the CDO which calls this REXX program:

```
/********************************************************************/
/*                                                                  */
/* DSPUNUSE - Display unused objects in a library after a           */
/*            particular date.                                      */
/*                                                                  */
/* The CPP is the REXX program DSPUNUSE.                            */
/*                                                                  */
/********************************************************************/

            CMD         PROMPT('Display Unused Objects')

            PARM        KWD(LIBRARY) TYPE(*NAME) LEN(10) MIN(1) +
                          PROMPT('Library name')
```

```
PARM       KWD(TYPE) TYPE(*CHAR) LEN(10) DFT(*ALL) +
             PROMPT('Object type')

PARM       KWD(LASTUSE) TYPE(*CHAR) LEN(6) PROMPT('Last +
             Used Date        (yymmdd)')

PARM       KWD(OUTPUT) TYPE(*CHAR) LEN(6) RSTD(*YES) +
             DFT(*) SPCVAL((*) (*PRINT)) PROMPT('Output +
             Type')
```

Note: *REXX should be specified as the "program to process command" when creating this command.

Appendix I. String Manipulation in REXX versus CL

REXX provides extensive capabilities for processing character strings through its
built-in functions and parsing techniques. In this section, we will look at some
common examples of string manipulation in CL program, and compare it with those
written in REXX.

Searching for a String Pattern

The following CL statements find the first sentence, which is delimited by a period,
in a 50-character variable &INPUT and place any remaining text in the variable
&REMAINDER:

```
        DCL &INPUT  *CHAR  LEN(50)
        DCL &REMAINDER *CHAR LEN(50)
        DCL &X *DEC LEN(2 0)  VALUE(01)
        DCL &L *DEC LEN(2 0)         /*  REMAINING LENGTH  */
        :
        :
SCAN:   IF ((%SUBSTRING(&INPUT &X 1) *NE '.')  *AND  +
          (&X *LT 50))  THEN(DO)
          CHGVAR  &X (&X+1)
          GOTO SCAN
        ENDDO
        CHGVAR VAR(&L) VALUE(50-&X)
        CHGVAR VAR(&X) VALUE(&X+1)
        CHGVAR VAR(&REMAINDER) VALUE(%SUBSTRING(&INPUT &X &L))
```

This is the REXX statement that performs the same function:

```
parse var input . '.' remainder
```

In fact, this is just a simple example of finding a single character pattern in a string.
If we are searching for a literal pattern that consists of several characters (for
example, `'QPGMR'`), the CL program will become much more complex, because it
treats the whole string as an array of single characters. However, the only changes
for the REXX statement are as follows:

```
parse var input . 'QPGMR' remainder
```

Extracting Words from a String

The following CL statements shows how three words can be extracted, with leading
and trailing blanks removed, from a 30-characters field, and assign them to the
variables &LIB, &FILE, and &MBR, respectively.

```
            DCL &INPUT  *CHAR  LEN(30)
            DCL &LIB *CHAR LEN(10)
            DCL &FILE *CHAR LEN(10)
            DCL &MBR *CHAR LEN(10)
            DCL &S *DEC LEN(2 0)                /*  Starting position.  */
            DCL &E *DEC LEN(2 0)                /*  Ending position.    */
            DCL &L *DEC LEN(2 0)                /*  Length of parameter.*/
            :
            :

            CHGVAR &S 1         /* Remove leading blanks for &LIB.*/
   LIBSTR: IF (%SST(&LIB &S 1) *EQ ' ')  THEN(DO)
                CHGVAR &S (&S+1)
                GOTO LIBSTR
            ENDDO
            CHGVAR &E (&S+1)     /* Find end of &LIB.*/
   LIBEND: IF (%SST(&LIB &E 1) *NE ' ')  THEN(DO)
                CHGVAR &E (&E+1)
                GOTO LIBEND
            ENDDO
            CHGVAR &L (&E-&S)
            CHGVAR &LIB (%SST(&LIB &S &L))

            CHGVAR &S (&E+1)     /* Remove leading blanks for &FILE.*/
   FILSTR: IF (%SST(&FILE &S 1) *EQ ' ')  THEN(DO)
                CHGVAR &S (&S+1)
                GOTO FILSTR
            ENDDO
            CHGVAR &E (&S+1)      /* Find end of &FILE.*/
   FILEND: IF (%SST(&FILE &E 1) *NE ' ')  THEN(DO)
                CHGVAR &E (&E+1)
                GOTO FILEND
            ENDDO
            CHGVAR &L (&E-&S)
            CHGVAR &FILE (%SST(&FILE &S &L))

            CHGVAR &S (&E+1)     /* Remove leading blanks for &MBR.*/
   MBRSTR: IF (%SST(&MBR &S 1) *EQ ' ')  THEN(DO)
                CHGVAR &S (&S+1)
                GOTO MBRSTR
            ENDDO
            CHGVAR &E (&S+1)     /* Find end of &MBR.*/
   MBREND: IF (%SST(&MBR &E 1) *NE ' ')  THEN(DO)
                CHGVAR &E (&E+1)
                GOTO MBREND
            ENDDO
            CHGVAR &L (&E-&S)
            CHGVAR &MBR (%SST(&MBR &S &L))
            :
            :
```

The following is the REXX statement that performs the same function:

```
  :
parse var input lib file mbr
  :
```

Concatenation with Numeric Variables

When trying to concatenate values in numeric variables with other character strings using CL, the Change Variable (CHGVAR) command is used. This is because CL supports the concatenation of character variables and character strings only.

The following CL example shows how several numeric and character variables can be concatenated to produce the message below for a workstation operator:

```
Customer ABC COMPANY, Account Number 12345, is overdue by 4 days.
```

It assumes that the variables for the account number (&ACTNUM) and the overdue days (&ODDAY) were declared as numeric variables.

```
DCL &MSG *CHAR LEN(50)
DCL &ODDAYA *CHAR LEN(3)
DCL &ACTNUMA *CHAR  LEN(6)
 :
 :
CHGVAR &ODDAYA  &ODDAY
CHGVAR &ACTNUMA &ACTNUM
CHGVAR &MSG('Customer' *BCAT &CUSNAME *CAT +
          ', Account Number' *BCAT &ACTNUMA *CAT +
          ', is overdue by' *BCAT &ODDAYA *BCAT 'days.')
 :
 :
```

As REXX regards all data as character strings, it makes no difference when concatenating numeric variables instead of character variables. This is the REXX statement which prepares the same message:

```
 :
msg = 'Customer' cusname', Account Number' actnum', is overdue by',
      odday 'days.'
 :
```

From these examples, we can see that REXX is really a handy and powerful tool in string manipulation. Moreover, as the syntax of the REXX clauses are simple and the instructions are quite self-explanatory, it also helps to increase the readability and maintainability of the programs.

The following is a command SBMCALL, which is designed to assist the users in building a CALL command with parameters and then submit it to batch. A REXX program and a CL program, which have the same functions, are developed to be the CPP of this command. The reader can compare these two programs in their techniques in string handling, and also their relative readability.

The way that parameters are passed to a REXX CPP is different from those to a CL CPP, as shown below. Therefore, the logic of these two programs in retrieving values from the passed parameters will be somewhat different.

If we type in the following command:

```
SBMCALL PGM(TESTLIB/PGMA) PARM(HELLO 'GOOD MORNING' 123)
```

The following character string will be passed to the REXX CPP:

```
PGM(TESTLIB/PGMA) PARM(HELLO 'GOOD MORNING' 123)
```

On the other hand, the following will be passed to a CL CPP:
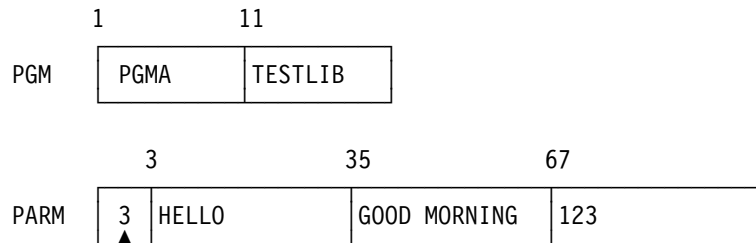
```
                  1              11
         PGM  | PGMA        | TESTLIB        |

                  3              35              67
         PARM | 3 | HELLO        | GOOD MORNING | 123         |
                  ↑
              A 2-byte binary value indicating the number of values
              passed.
```

Note: The SBMCALL command is changed from the BLDCALL command in QUSRTOOL. The only difference is that a SBMJOB command is issued in the CPP of SBMCALL, but this command is issued by the calling program of the BLDCALL command.

This is the REXX CPP of SBMCALL:

```
/*********************************************************************/
/*  REXX program as the CPP of SMBCALL command.                      */
/*  SBMCALL build the parameter list for a CALL command and submit   */
/*  it to batch.                                                     */
/*                                                                   */
/*     Parameter passed:                                            */
/*                                                                   */
/*         Pgm          :  Program name                              */
/*         Parm_list    :  A list of parameters                      */
/*                                                                   */
/*********************************************************************/
/* Setup ERROR, FAILURE, and SYNTAX condition traps.*/
signal on error name command_error
signal on failure name command_error

/* Parse out the program value from the CDO 'PGM' keyword.*/
arg 'PGM(' lib '/' pgm ')'

/* Parse out the parameter list from the CDO 'PARM' keyword.*/
arg 'PARM(' parmlist ')'

/*  Format the program name for the CALL command.*/
if lib = '*LIBL'
  then outpgm = pgm
  else outpgm = lib'/'pgm

rqsdta = 'Call' outpgm ' PARM('

/*  Format the parameter list for the CALL command.*/
do while (parmlist ¬= ' ') & (length(rqsdta) < 256)
  parse var parmlist parm parmlist
  /* Check if the parameter is in apostrophes (more than 1 word).*/
  if left(parm,1) = "'"  then do
     outparm = parm
      /* Get the next word until closing apostrophe is found.*/
      do while right(parm,1) ¬= "'"
```

```
            parse var parmlist parm parmlist
            outparm = outparm parm
          end
        /* Put a space after each parameter.*/
        outparm = outparm' '
    end
      /* Add apostrophes to those unquoted parameters.*/
    else outparm = "'"parm"' "
    rqsdta = rqsdta||outparm
  end

if length(rqsdta) ¬< 256 then do
  msg = 'The RQSDTA area being assembled has exceeded 256 bytes'
  'SNDPGMMSG MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) MSGDTA(&msg)'
end
else do
  rqsdta = rqsdta||')'
  'SBMJOB RQSDTA(&rqsdta)'
end
/* End of Program.*/
exit


/*********************************************************************/
/*  command_error : ERROR & FAILURE condition trap                   */
/*********************************************************************/
command_error:

  parse source system start srcmbr srcfile srclib
  say 'Unexpected error at line 'sigl' of REXX program 'srcmbr,
      ' in 'srclib'/'srcfile'.  The exception id is 'rc'.'


exit(rc)
```

This is the listing of CL program SBMCALL:

```
/* CL Program as the CPP of SBMCALL.                                 */
          PGM        PARM(&FULLPGM &LIST)
          DCL        &FULLPGM *CHAR LEN(20)
          DCL        &PGM *CHAR LEN(10)
          DCL        &LIB *CHAR LEN(10)
          DCL        &LIST *CHAR LEN(1602)
          DCL        &RQSDTA *CHAR LEN(256)
          DCL        &C *DEC LEN(3 0)            /* Index to RQSDTA area.*/
          DCL        &W *DEC LEN(3 0)            /* Index to Parm.*/
          DCL        &X *DEC LEN(3 0)            /* Count of nbr.*/
          DCL        &Z *DEC LEN(5 0) VALUE(3)   /* Parm loc.*/
          DCL        &P *DEC LEN(5 0) VALUE(11) /* Index to Pgm.*/
          DCL        &L *DEC LEN(5 0) VALUE(11) /* Index to Lib.*/
          DCL        &TEMP *CHAR LEN(50)
          DCL        &LSTCNT *DEC LEN(5 0)
          DCL        &PARM *CHAR LEN(32)
          MONMSG     MSGID(CPF0000) EXEC(GOTO ERROR)

          /* Extract &PGM and &LIB from &FULLPGM .*/
          CHGVAR     &PGM %SST(&FULLPGM 1 10)
          CHGVAR     &LIB %SST(&FULLPGM 11 10)

          /* Determine program name length */
```

```
PGMLEN:     CHGVAR     &P (&P - 1)
            IF         (%SST(&PGM &P 1) *EQ ' ') GOTO PGMLEN
            IF         (&LIB *EQ '*LIBL') DO       /* LIBL specified */
            CHGVAR     &RQSDTA ('CALL ' *CAT &PGM *TCAT ' PARM(')
            /* Next location in cmd work.*/
            CHGVAR     &C (12 + &P)
            GOTO       PARMKWD
            ENDDO                                   /* LIBL specified.*/


            /* Determine library name length.*/
LIBLEN:     CHGVAR     &L (&L - 1)
            IF         (%SST(&LIB &L 1) *EQ ' ') GOTO LIBLEN
            CHGVAR     &RQSDTA ('CALL ' *CAT &LIB *TCAT '/' *CAT +
                         &PGM *TCAT ' PARM(')
            /* Next location in cmd work.*/
            CHGVAR     &C (13 +&P + &L)
            /* Get the number of parameters in the list.*/
PARMKWD:    CHGVAR     &LSTCNT %BINARY(&LIST 1 2)


            /* Begin loop for each parm.*/
LOOP:       CHGVAR     &X (&X + 1)              /* Next parm.*/
            CHGVAR     &PARM %SST(&LIST &Z 32)  /* Extract parm.*/
            IF         (&PARM *EQ '     ') DO    /* Blank parm.*/
            CHGVAR     &W 1                     /* Provide 1 blank.*/
            GOTO       BLDPARM
            ENDDO                               /* Blank parm.*/
            CHGVAR     &W 33


            /* Determine parameter length.*/
PARMLEN:    CHGVAR     &W (&W - 1)
            IF         (%SST(&PARM &W 1) *EQ ' ') GOTO PARMLEN


            /* Format the parameter list for the CALL command.*/
BLDPARM:    /* Add apostrophes to parameter.*/
            CHGVAR     &TEMP ('''' *CAT %SST(&PARM 1 &W) *CAT '''')
            /* Check if length of parameter list exceeds 256.*/
            CHGVAR     &W (&W + 2)      /* Parm value len plus  */
                                         /* apostrophes.*/
            CHGVAR     %SST(&RQSDTA &C &W) &TEMP
            MONMSG     MSGID(MCH0603) EXEC(DO)    /* Exceeds 256.*/
            RCVMSG     MSGTYPE(*EXCP)             /* Remove MCH message.*/
            SNDPGMMSG  MSGID(CPF9898) MSGF(QCPFMSG) MSGTYPE(*ESCAPE) +
                         MSGDTA('The RQSDTA area being assembled +
                         has exceeded 256 bytes')
            ENDDO                                 /* Exceeds 256.*/
            /* Next loc in cmd work. */
            CHGVAR     &C (&C + &W + 2)
            /* Increase for next parm.*/
            CHGVAR     &Z (&Z + 32)


            /* Loop back until all parms are processed.*/
TESTLOOP:   IF         (&X *LT &LSTCNT) GOTO LOOP /* Loop back.*/
                         /* Begin the end of all lists processing.*/
            CHGVAR     &C (&C - 2) /* No need for extra blanks.*/
            CHGVAR     %SST(&RQSDTA &C 1) ')'    /* Final right paren.*/


SUBMIT:     SBMJOB     RQSDTA(&RQSDTA)
            GOTO       END
```

```
            /* Run time error handling.*/
 ERROR:     SNDMSG     MSG('Unexpected error in CL Program SBMCALL') +
                       TOUSR(*REQUESTER)


 END:       ENDPGM
```

The following is the CDO of the SBMCALL command:

```
/*****************************************************************/
/*                                                               */
/* The Submit call command builds the command and the parameter  */
/*  list and then submits it to batch using SBMJOB command.       */
/*                                                               */
/* The CPP is the REXX program SBMCALL.                          */
/*                                                               */
/*****************************************************************/

            CMD        PROMPT('Submit Call command to Batch')

            PARM       KWD(PGM) TYPE(QUAL1) MIN(1) +
                         PROMPT('Program name')

            PARM       KWD(PARM) TYPE(*CHAR) MIN(1) MAX(50) +
                         PROMPT('Parameters        (32 or less)')

QUAL1:      QUAL       TYPE(*NAME) LEN(10) EXPR(*YES)
            QUAL       TYPE(*NAME) LEN(10) DFT(*LIBL) SPCVAL(*LIBL) +
                         EXPR(*YES) PROMPT('Library name')
```

Note: If the REXX program is used as the CPP of SBMCALL, *REXX should be
specified as the 'Program to process command' when creating this command.

Glossary

A

absolute positional pattern. The part of a parsing template that allows a string to be split by the specification of numeric positions. A positional pattern has no sign or has an equal sign.

abuttal operator. When two terms in an expression are adjacent and are not separated by an operator, they are said to abut. The effect of this operation is that the two terms are concatenated without a blank.

arithmetic operator. An operator used to perform arithmetic operations on character strings that are valid numbers. The arithmetic operators include addition (+), subtraction (-), multiplication (*), exponentiation (**), division (/), integer division (%), remainder (//), prefix + and prefix -.

array. An arrangement of data in one or more dimensions, such as a list, a table, or a multidimensional arrangement of items. Arrays are implemented using compound symbols.

B

binary string. A literal string expressed using a binary (base 2) representation of a value. The binary representation is a sequence of zero or more binary digits (the characters 0 or 1), enclosed in quotation marks and followed by the character b.

bit. A contraction of binary digit. Either of the binary digits, 0 or 1. Compare with *byte*.

Boolean operator. An operator each of whose operands and whose result take one of two values (0 or 1).

byte. (1) The smallest unit of storage that can be addressed directly. (2) A group of 8 adjacent bits. In the EBCDIC coding system, 1 byte can represent a character. In the double-byte coding system, 2 bytes represent a character.

C

CCSID. See *coded character set identifier (CCSID)*.

character. Any letter, number, or other symbol in the data character set that is part of the organization, control, or representation of data.

character format. A format that is used in the REXX conversion functions to indicate that data is in a textual form as opposed to machine-readable form.

CL. See *control language (CL)*.

C language. A language used to develop application programs in compact, efficient code that can be run on different types of computers with minimal change.

clause. The fundamental grouping of REXX syntax. A clause is composed of zero or more blanks, a sequence of tokens, zero or more blanks, and the semicolon delimiter.

coded character set identifier (CCSID). A 16-bit number identifying a specific set of encoding scheme identifiers, character set identifiers, code page identifiers, and other relevant information that uniquely identifies the coded graphic character representation used.

compound symbol. A symbol that permits the substitution of variables within its name, when referred to. A compound symbol contains at least one period, and at least two other characters. It cannot start with a digit or period, and if there is only one period in the compound symbol, it cannot be the last character. The compound symbol begins with a stem (that part of the symbol up to and including the first period). The stem is followed by the tail (the parts of the name, delimited by periods, that are constant symbols, simple symbols, or null). Compound symbols allow the construction of arrays, associative tables, lists, and so on.

computing environment. Type of computer system. The AS/400 system together with OS/400 software is one computing environment. The Personal System/2* (PS/2*) together with OS/2 is another computing environment.

condition. A specific event, or state, that can be trapped by the REXX CALL ON or SIGNAL ON instruction.

condition trap. The method by which the explicit flow of processing in a REXX program can be modified. Condition traps are enabled or disabled using the ON or OFF subkeywords of the CALL and SIGNAL instructions.

control language (CL). The set of all commands with which a user requests system functions.

control language (CL) program. A program that is created from source statements consisting entirely of control language commands.

control language (CL) variable. A program variable that is declared in a control language program and is available only to the CL program.

control structure. A REXX instruction that determines if, when and how often part of a program gets processed.

controlled repetitive loop. A repetitive DO loop in which the repetitive phrase specifies a control variable. The variable is given an initial value before the first run of the instruction list and is then stepped (by adding the result of an optional expression) before the second and subsequent times that the instruction list is run.

D

derived name. The stem of the symbol, in uppercase, followed by the tail in which all simple symbols have been replaced by their value. It is also the default value of a compound symbol.

double-byte character set (DBCS). A set of characters in which each character is represented by 2 bytes. Languages such as Japanese, Chinese, and Korean, which contain more symbols that can be represented by 256 code points, require double-byte character sets. Because each character requires 2 bytes, the typing, displaying, and printing of DBCS characters requires hardware and programs that support DBCS. Four double-byte character sets are supported by the system: Japanese, Korean, Simplified Chinese, and Traditional Chinese. Contrast with *single-byte character set*.

E

embedded blank. A space between characters within a unit of data.

execute. To perform the actions specified by a program or a portion of a program (to carry out an instruction).

extended characters. Double-byte characters that are stored in a DBCS font file, not in the hardware of a DBCS-capable work station. When displaying or printing extended characters, the work station receives them from the DBCS font table under control of the extended character processing function of the operating system.

F

facility. A service provided by an operating system for a particular purpose.

FIFO. See *first-in first-out (FIFO)*.

first-in first-out (FIFO). In REXX, a queuing technique in which the next item to be retrieved is the item that has been on the queue for the longest time. Contrast with *last-in first-out (LIFO)*.

fixed-point notation. A REXX number that is written without exponentiation.

floating-point notation. A REXX number that is written using exponentiation.

function. A series of instructions that a REXX procedure calls to perform a specific task and to return a value. The three types of routines that can be called as functions are internal, built-in, and external.

function invocation. A term in an expression which invokes a routine that carries out some procedure and then returns a string.

H

halt. To cease or to stop, as in halt execution of your program.

hexadecimal string. In REXX, any sequence of zero or more hexadecimal digits (0-9, a-f, A-F), optionally separated by blanks, delimited by apostrophes or quotation marks, and immediately followed by the symbol x or X.

I

imbedded blank. See *embedded blank*.

implied semicolon. In REXX, an assumed semicolon at the end of each line.

invalid. Being logically unsupported and thereby not allowed.

iteration. The process of repeatedly running a set of computer instructions until some condition is satisfied.

K

keyword instruction. One or more clauses, the first of which starts with a keyword that identifies the instruction. Some keyword instructions affect the flow of control, while others provide services to the programmer. CALL, DO, and PARSE are examples of keyword instructions.

L

last-in first-out (LIFO). In REXX, a queuing technique in which the next item to be retrieved is the item most recently placed in the queue. Contrast with *first-in first-out (FIFO)*.

LIFO. See *last-in first-out (LIFO)*. technique in which the next item to be retrieved is the item most recently placed in the queue. Contrast with *first-in first-out (FIFO)*.

literal string. In REXX, a sequence including any characters that are delimited by apostrophes or quotation marks.

M

mantissa. In floating-point format, the number that precedes the E. The value represented is the product of the mantissa and the power of 10 specified by the exponent.

N

nonzero. A value that is not zero.

numeric pattern. A pattern that specifies, by column number, how input data is to be parsed.

O

operand. In REXX, a term or expression that is operated on by an operator.

operator precedence. In programming languages, an order relationship that defines the sequence of the application of operators within an expression.

P

placeholder. The symbol, consisting of a single period in a REXX parsing template, that can be replaced by a value while running a REXX program. A placeholder has the same effect as a variable name, except that no variable is set.

prefix operation. An operation on one value, specified by writing by writing a prefix operator in front of the value. The only prefix operators are + and -. Example: Say -(1)

pseudo-CL variable. A variable used in CL commands, whose name conforms to the CL programming rules for variables, but actually refers to a REXX variable. The name must begin with an ampersand, but it is stripped off when determining the name of the actual REXX variable that is to be used. Pseudo-CL variables must be valid REXX variable names and valid CL variable names.

Q

queue. A list of messages, jobs, file, or requests waiting to be read, processed, printed, or distributed in a predetermined order.

R

RC. A REXX special variable set to the return code from any executed host command or subcommand. It is also set to the return code when the conditions ERROR, FAILURE, and SYNTAX are trapped.

reiterate or reiterated. To repeat a loop.

relative positional pattern. The part of a parsing template that uses a plus or minus sign to indicate movement relative to a previous pattern match.

REstructured eXtended eXecutor (REXX) language. A general-purpose programming language, particularly suitable for CL commands, or programs for personal computing. Procedures and programs written in this language can be interpreted by the REXX/400 interpreter. See also *REXX/400*.

RESULT. A REXX special variable that is set by the RETURN instruction in a called routine. The RESULT special variable is dropped if the called routine does not return a value.

return code. For printer files, display files, and ICF files, a value sent by the system to a program to indicate the results of an operation by that program.

REXX language. See *REstructured eXtended eXecutor (REXX) language*.

REXX/400. The Operating System/400 implementation of the Systems Application Architecture Procedures Language. REXX/400 is a programming language that is supported by an interpreter provided as part of the OS/400 licensed program. See also *REstructured eXtended eXecutor (REXX)*.

S

SAA. See *Systems Application Architecture (SAA)*.

simple repetitive loop. A repetitive DO loop in which the repetitive phrase is an expression that evaluates to a count of iterations.

single-byte character set (SBCS). A character set in which each character is represented by a one-byte code. Contrast with *double-byte character set*.

stem. That part of a compound symbol up to and including the first period. It contains just one period, which is the last character. It cannot start with a digit or a period. A reference to a stem can also be used to manipulate all variables sharing that stem.

step. To cause a computer to run one operation.

string. A sequence of elements of the same nature, such as characters considered as a whole; for example, character string, binary string, and hexadecimal string.

string concatenation. An operation that joins two characters or strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two characters or strings.

structured programming. Programming using blocks of instructions where flow is controlled by instructions such as DO, IF THEN ELSE, and SELECT, instead of direct branching instructions.

subkeyword. A symbol reserved by the language processor within the clause of individual instructions. For example, the symbol FOREVER is a subkeyword of the DO instruction.

subroutine. An internal, built-in, or external routine called by the CALL instruction that may or may not return a result string. If a subroutine returns a result string, a subroutine can also be called by a function call, in which case it is being called as a function.

swapped. When using the REXX REVERSE function, pertaining to a process that exchanges the values in the input string by reversing their positions.

Systems Application Architecture (SAA). Pertaining to an architecture defining a set of rules for designing a common user interface, programming interface, application programs, and communications support for strategic operating systems such as the OS/2, OS/400, VM, and MVS operating systems.

T

tail. The part of a compound symbol that follows the stem. A tail can consist of constant symbols, simple symbols, and periods.

term. A string, symbol, or function call contained within a REXX expression.

terminate. To stop execution of a program.

terminating error. An error in a program that causes its execution to stop.

token. The unit of low-level syntax from which REXX clauses are built. Tokens include literal strings, operator characters, and special characters.

traceback. Trace output that shows a failing instruction when a syntax error occurs.

translation table. An object that contains a set of hexadecimal characters used to translate one or more characters of data. The table can be used for translation of data being moved between the system and a device. for example, data stored in one national language character set may need to be displayed or entered on display devices that support a different national language character set. The table can also be used to specify an alternative collating sequence or field translation functions. The system-recognized identifier for the object type is *TBL.

trap. In REXX, to recognize that a currently enabled condition occurred and to perform the CALL or SIGNAL instruction specified when the condition trap was enabled.

trigger point. A threshold or boundary limit used in the REXX FORMAT function.

U

uninitialized value. When a REXX variable is used before a value is assigned to it, it is given an uninitialized value, which is the same as its name, with all alphabetic characters in uppercase.

V

variable. A name used to represent data whose value can be changed while the program is running by referring to the name of the variable.

variable pool interface. An application program interface that allows programs written in other languages to access variables being used by or contained in an active REXX program.

W

word. A sequence of characters that do not include any blanks. Words may be used as units for manipulation during parsing and by many built-in functions.

4

400/REXX. See *REXX/400*.

Bibliography

The manuals below are listed with their full title and base order number.

For more information about REXX, see:

REXX/400 Reference, SC41-5729

This manual provides detail on all REXX instructions, functions, input and output, parsing, and application interfaces.

You may want to refer to other AS/400 manuals for more specific information about a particular topic. The *AS/400 Advanced Series Handbook*, GA19-5486, provides an introduction to the system characteristics and software offerings available for the AS/400 system.

For information about operating the AS/400 system and its display stations, see:

System Operation, SC41-4203.

This manual provides general information about how to run the system, how to send and receive messages, and use the display station function keys.

For more information about programming, see:

Backup and Recovery, SC41-5304.

This manual provides information about the different media available to save and protect system data.

DB2 for AS/400 Database Programming, SC41-5701.

This manual provides a detailed discussion of the AS/400 database structure, including information on how to create, describe, and manipulate database files.

Data Management, SC41-5710.

This manual provides information about using files in application programs. Information includes: spooling support, copying files, and tailoring a system using double-byte data.

Security – Reference, SC41-5302.

This manual discusses general security concepts and planning for security on the system. It also includes information for all users about resource security.

Work Management, SC41-5306.

This manual provides information about creating and changing the work management environment, working with system values, collecting and using performance data to improve system performance.

For detailed information about CL commands, see:

Programming Reference Summary, SX41-5720.

This manual provides quick reference information about the structure of the AS/400 commands, including syntax diagrams and error messages that can be monitored. Summary charts, system values, and DDS keywords for the AS/400 system are also included.

CL Programming, SC41-5721.

This manual provides a comprehensive discussion of AS/400 programming topics. Topics such as program communication, working with objects and libraries, creating CL programs and commands, and developing applications are discussed.

CL Reference, SC41-5722.

This manual provides a description of the AS/400 control language (CL) commands. Each command is described, including its syntax diagram, parameters, default values, and keywords.

For more information about AS/400 utilities mentioned in this guide, see:

DDS Reference, SC41-5712.

This manual provides a detailed description of the entries and keywords needed to externally describe database files and certain device files.

ADTS/400: Character Generator Utility, SC09-1769.

This manual provides information about using the character generator utility (CGU) to create and maintain a double-byte character set on the AS/400 system.

ADTS/400: Programming Development Manager, SC09-1771.

This manual provides information about using the programming development manager (PDM) to work with lists of libraries, objects, members, and user-defined options.

ADTS/400: Screen Design Aid, SC09-1768.

This manual provides information about using the screen design aid (SDA) to design, create, and maintain display formats and menus.

ADTS/400: Source Entry Utility, SC09-1774.

This manual provides information about using the source entry utility (SEU) to create and edit source members.

For additional information on the SAA Procedures Language, see:

SAA Common Programming Interface REXX Level 2 Reference, SC24-5549.

225

This manual may be useful to more experienced REXX users who may want to code portable programs. This manual defines the SAA Procedures Language. Descriptions include the use and syntax of the language as well as explanations on how the language processor interprets the language as a program is running.

SAA Common Programming Interface Communications Reference, SC26-4399.

This manual will help you program with the CPI Communications interface. It contains general-use programming interfaces which let you write programs that use the services of CPI Communications.

National Language Support, SC41-5101.

This manual will help you plan and use the national language support (NLS) function on the AS/400 system. It contains information on how to understand the national language support concepts, how to use national language support in a multilingual environment, and how to write internationalized applications in a multilingual environment.

DB2 for AS/400 SQL Reference, SC41-5612.

This manual provides information about SQL/400 statements and their parameters. It also includes an appendix describing the SQL communications area (SQLCA) and SQL description area (SQLDA).

DB2 for AS/400 SQL Programming, SC41-5611.

This manual provides information about the EXECSQL environment.

Index

Special Characters

- tracing flag 128
+++ tracing flag 128
>.> tracing flag 128
>>> tracing flag 128
>C> tracing flag 129
>F> tracing flag 129
>L> tracing flag 129
>O> tracing flag 129
>P> tracing flag 129
>V> tracing flag 129

A

ABBREV function 74
Add REXX Buffer (ADDREXBUF) command 116, 119, 162, 170, 193, 202
ADDRESS function 80, 105
ADDRESS instruction 81
ADDREXBUF command 116, 119, 162, 170, 193, 202
AND operator 42
application program interfaces (APIs)
  QREXQ 115, 180, 193, 199—205
  QREXVAR 101, 185—192
  QREXX 10, 12, 86
ARG instruction 19
arithmetic operators 32, 143
array
  description 23
  two dimensional 25
  using compound symbols 23, 26
assignment
  description 6

B

batch mode, SAY and PULL in 16
binary strings 111
book, purpose ix
books to read ix
branches 45
built-in functions
  See functions, built-in

C

CALL command 79, 92, 178, 180, 183, 184, 191, 213
CALL instruction 26, 45, 79, 92, 100, 132, 135, 175
Call Program (CALL) command 79, 92, 178, 180, 183, 184, 191, 213

CCSID (coded character set identifier)
  See coded character set identifier (CCSID)
CENTER function 74
CENTRE function 74
Change Command (CHGCMD) command 11
Change Variable (CHGVAR) command 59, 211, 212, 213, 215
CHGCMD command 11
CHGVAR command 59, 211, 212, 213, 215
CL (Control Language)
  command environment 86
  commands
    See commands, CL
  conditions 85
  program boundaries, sensitivity to 91
  replacing CL programs with REXX programs 59
  variables, pseudo-CL
    See pseudo-CL variables
clause
  assignments 6
  commands 8
  instructions 6
  interpretation 79
  labels 7
  null 5
clause interpretation 79
COBOL, using with REXX
  pushing data to the external data queue 204
coded character set identifier (CCSID)
  DBCS 141
  SBCS 141
  using TRACE 123
  within REXX source file 17, 18
command environments
  CL command environment 79, 86
  CPICOMM 80, 95
  description 79—95
  user-defined command environments 80
commands, CL
  *RTNVAL attribute 87
  *VARY attribute 87
  Add REXX Buffer (ADDREXBUF) 116, 119, 162, 170, 193, 202
  Call Program (CALL) 79, 92, 178, 180, 183, 184, 191, 213
  Change Command (CHGCMD) 11
  Change Variable (CHGVAR) 59, 211, 212, 213, 215
  command parameters 87
  Create Command (CRTCMD) 11
  Create Library (CRTLIB) 8
  Create Source Physical File (CRTSRCPF) 8

Reader Comments—We'd Like to Hear from You!

AS/400 Advanced Series
REXX/400 Programmer's Guide
Version 4

Publication No. SC41-5728-00

Overall, how would you rate this manual?

	Very Satisfied	Satisfied	Dissatisfied	Very Dissatisfied
Overall satisfaction				

How satisfied are you that the information in this manual is:

Accurate				
Complete				
Easy to find				
Easy to understand				
Well organized				
Applicable to your tasks				
T H A N K Y O U !				

Please tell us how we can improve this manual:

_____
_____
_____
_____
_____
_____

May we contact you to discuss your responses? __ Yes __ No
Phone: (____) _____ Fax: (____) _____ Internet: _____

To return this form:

- Mail it
- Fax it
  United States and Canada: **800+937-3430**
  Other countries: **(+1)+507+253-5192**
- Hand it to your IBM representative.

Note that IBM may use or distribute the responses to this form without obligation.

Name   Address

Company or Organization
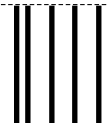

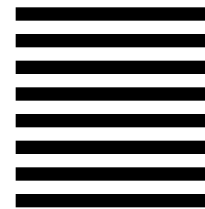
Phone No.

Cut or Fold
Along Line

Fold and Tape   **Please do not staple**   Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST-CLASS MAIL   PERMIT NO. 40   ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

ATTN DEPT 542 IDCLERK
IBM CORPORATION
3605 HWY 52 N
ROCHESTER MN 55901-9986

Fold and Tape   **Please do not staple**   Fold and Tape

Cut or Fold
Along Line

SC41-5728-00

IBM ®

IBM   AS/400 Advanced Series   REXX/400 Programmer's Guide   *Version 4*