ARexx User's Reference Manual

The
REXX Language
for the Amiga

ARexx User's Reference Manual

Version 1.0

The REXX Language for the Amiga

Copyright Notice

Disclaimer

This product is offered for sale "as is" with no representation of fitness for any particular purpose. The user assumes all risks and responsibilities related to its use. The material within is believed to be accurate, but the author reserves the right to make changes to the software or documentation without notice.

Distribution

ARexx software and documentation are available from:

William S. Hawes
P.O. Box 308
Maynard, MA 01754
(508) 568-8695

Please direct orders or inquiries about this product to the above address. Site licenses are available; write for further information.

About ...

ARexx was developed on an Amiga 1000 computer with 512K bytes of memory and two floppy disk drives. The language prototype was developed in C using Lattice C, and the production version was written in assembly-language using the Metacomco Assembler. The documention was created using the TxEd editor, and was set in TEX using AmigaTEX. This is a 100% Amiga product.

Trademarks

Amiga, Amiga WorkBench, and Intuition are trademarks of Commodore-Amiga, Inc.

Table of Contents

ARexx User's Reference Manual

Introduction

Welcome to ARexx, an implementation of the REXX language for the Amiga computer. ARexx is a powerful programming tool, but one which by virtue of its clean syntax and sparse vocabulary is also easy to learn and easy to use.

1 Organization of this Document

This document will attempt to fill the roles of User's Manual, Language Reference, and Programmer's Guide. The chapters that follow have been organized to provide a gentle introduction to the language.

- Chapter 1, *What is ARexx?*, gives an overview of the ARexx language and its implementation on the Amiga.
- Chapter 2, *Getting Acquainted*, tells how to install ARexx on your Amiga and presents several example programs to illustrate the features of the language.
- Chapter 3, *Elements of the Language*, introduces the language structure and syntax.
- Chapter 4, *Instructions*, describes the action statements of ARexx.
- Chapter 5, *Commands*, describes the program statements used to communicate with external programs.
- Chapter 6, *Functions*, explains how functions are called and documents the Built-In Function library.
- Chapter 7, *Tracing and Interrupts*, describes the source-level debugging features useful for developing and testing programs.
- Chapter 8, *Parsing and Templates*, describes the instructions used to extract words or fields from strings.
- Chapter 9, *The Resident Process*, describes the capabilities of the global communications and resources manager.
- Chapter 10, *Interfacing to ARexx*, describes how to design and implement an interface between ARexx and an external program.
- Appendix A, *Error Messages*, lists the error messages issued by the interpreter.
- Appendix B, *Limits and Compatibility*, discusses the compatibility of ARexx with the language standard.
- Appendix C, *The ARexx Systems Library*, documents the functions in the ARexx systems library.
- Appendix D, *The Support Library*, documents the library of Amiga-specific functions.
- Appendix E, *Distribution Files*, lists the files on the distribution disk.

Finally, a Glossary and an Index are provided.

Using this Manual

If you are new to the REXX language, or perhaps to programming itself, you should review chapters 1 through 4 and then play with ARexx by running some of the sample programs given in chapter 2. Further examples are available in the `:rexx` directory of the distribution disk.

If you are already familiar with REXX you may wish to skip directly to chapter 5, which begins to present some of the system-dependent features of this implementation. A summary of the compatibility of ARexx with the language definition is contained in Appendix B.

Typographic Conventions

Describing a language is sometimes difficult because of the multiple and changing contexts involved. To help clarify the presentation here, a simple typographic convention has been adopted throughout the document. All of the terms and words specific to the REXX language, as well as the program examples and computer input and output, have been set `in typewriter font like this`. This should help to distinguish the language keywords and examples from the surrounding text.

2 Future Directions

ARexx, like most software products, will probably evolve somewhat over the next few years as new features are added, old bugs are removed, and market imperatives become more apparent. While the core language will probably undergo few modifications, many capabilities will be added to the function libraries supported by ARexx. Your comments and suggestions for improvements to ARexx are most welcome.

The author sincerely hopes that other software developers will consider using ARexx with their products. The advantages of having a rich variety of software products sharing a common user interface and a common procedural interface cannot be overstated. This is the underlying promise of the Amiga's multitasking capability, and that which most sets it apart from other inexpensive computers.

Example Programs. One of the best ways to learn a computer language is to study examples written by more experienced programmers. The ARexx distribution disk includes a few example programs in the `:rexx` directory, and more programs will be added in future releases.

If you have written a REXX language program (for any computer) that you think would be of interest to a more general audience, please send it to the author for consideration. Programs should be of interest either in terms of their specific functionality or as an example of programming technique. Each program submitted should include an author credit and a few lines of commentary on its intended function.

Chapter 1

What is ARexx?

ARexx is a high-level language useful for prototyping, software integration, and general programming tasks. It is an implementation of the REXX language described by M. F. Cowlishaw in *The REXX Language: A Practical Approach to Programming* (Prentice-Hall, 1985), and follows the language definition closely. ARexx is particularly well suited as a command language. Command programs, sometimes called "scripts" or "macros", are widely used to extend the predefined commands of an operating system or to customize an applications program.

As a programming language, ARexx can be useful to a wide cross section of users. For the novice programmer, ARexx is an easy-to-learn yet powerful language that serves as a good introduction to programming techniques. Its source-level debugging facilities will help take some of the mystery out of how programs work (or don't work, as is more frequently the case.)

For the more sophisticated user, ARexx provides the means to build fully integrated software packages, combining different applications programs into an environment tailored to their needs. A common command language among applications that support ARexx will bring uniformity to procedural interfaces, much as the Amiga's Intuition provides uniformity in the graphical interface.

Finally, for the software developer, ARexx offers a straightforward way to build fully programmable applications programs. Developers can concentrate their efforts on making the basic operations of their programs fast and efficient, and let the end user add the frills and custom features.

1-1 Language Features

Some of the important features of the language are:

Typeless Data. Data are treated as typeless character strings. Variables do not have to be declared before being used, and all operations dynamically check the validity of the operands.

Command Interface. ARexx programs can issue commands to external programs that provide a suitable command interface. Any software package that implements the command interface is then fully programmable using ARexx, and can be extended and customized by the end user.

Tracing and Debugging. ARexx includes source-level debugging facilities that allow the programmer to see the step-by-step actions of a program as it runs, thereby reducing the time required to develop and test programs. An internal interrupt system permits special handling of errors that would otherwise cause the program to terminate.

Interpreted Execution. ARexx programs are run by an interpreter, so separate compilation and linking steps are not required. This makes it especially useful for prototyping and as a learning tool.

Function Libraries. External function libraries can be used to extend the capabilities of the language or as bridges to other programs. Libraries also allow ARexx programs to be used as "test drivers" for software development and testing.

Automatic Resource Management. Internal memory allocation related to the creation and destruction of strings and other data structures is handled automatically.

1-2 ARexx on the Amiga

ARexx was designed to run on the Amiga, and makes use of many of the features of its multitasking operating system. ARexx programs run as separate tasks and may communicate with each other or with external programs. The interpreter follows the design guidelines expected of well-behaved programs in a multitasking environment: specifically, it uses as little memory as possible and is careful to return resources to the operating system when they are no longer needed. Memory requirements were minimized by implementing the entire ARexx system as a shared library, so that only one copy of the program code must be loaded.

1-3 Further Information

The aforementioned book by M. F. Cowlishaw is highly recommended to those interested in further information about REXX. It presents an interesting discussion of the design and development of the language.

Chapter 2

Getting Acquainted

This chapter explains how to install ARexx on your Amiga computer and shows some example programs.

2-1 Installing ARexx

ARexx requires an Amiga computer with at least 256k of memory, and will operate under V1.1 or V1.2 of the operating system. It uses the double-precision math library called "`mathieeedoubbas.library`" that is supplied with the Amiga WorkBench disk, so make sure that this file is present in your `LIBS:` directory. The distribution disk includes the language system, some example programs, and a set of the INCLUDE files required for integrating ARexx with other software packages. The distribution files are listed in Appendix E.

ARexx and WorkBench

ARexx can be installed and loaded from within the icon-based environment provided by the Amiga WorkBench. However, it is a primarily a text-oriented language system and requires a good text editor and file management environment to be most effective. Unless you purchased ARexx as part of an applications package that includes an integrated editor, you'll probably find it useful to become familiar with the Command Line Interface (CLI) environment on the Amiga.

Installation

The ARexx language system consists of a shared library, a resident program, and several command utilities. All of the required files are contained in the `:c` and `:libs` directories of the distribution disk. ARexx may be installed on any of the system disks with which it will be used, but first check the `:c` and `:libs` directories of each disk to make sure that there are no naming conflicts. The following steps will then install ARexx on the system disk, provided that two disk drives are available:

1. Activate a CLI window.
2. Copy the ARexx `:libs` directory to the system `LIBS:` directory with the command "`copy df1:libs to libs:`".
3. Copy the ARexx `:c` directory to the system `C:` directory with the command "`copy df1:C to c:`".

Single-Drive Systems. Installing software in a single-drive system can be very confusing, so an installation utility has been provided with the ARexx distribution disk. It copies the `:c` and `:libs` directories of the distribution disk into memory, and then prompts the user to insert each disk that is to receive the files. Follow these steps to run the installation utility:

1. Activate a CLI window.
2. Insert the distribution disk into drive 0 and type "`df0:rxinstall`".
3. At the program prompt, insert the system disk on which ARexx is to be installed into drive 0.
4. Repeat step 3 as required.

Starting the Resident Process

ARexx programs are launched by a background program called the *resident process.* It can be started by issuing the command `rexxmast` and must be active before any ARexx programs can be run. The `rexxmast` program briefly displays a small window to announce itself, and then disappears into the background to await your next request. If you will be using ARexx frequently, you can place the `rexxmast` command in the "`startup-sequence`" file that resides in the system `S:` directory. This will start the resident process automatically when you reboot the computer.

After the resident process has been loaded, ARexx programs can be run from the CLI by typing the command `rx` followed by the program name and any arguments. For example, the sample program `calc.rexx`, which evaluates an expression, could be run by typing "`rx :rexx/calc 1+1.`"

You may not need to start up the resident process if you are using a software package that starts it automatically. Applications that use ARexx can test whether the resident process is active by checking for a public message port named "REXX." If the port hasn't been opened, the program can issue the `rexxmast` command directly.

The resident process can be closed using the command `rxc`; it will then exit as soon as the last ARexx program finishes execution. Unless you are very short on memory space, there is usually no reason to close ARexx, as it simply waits in the background for the next program to run.

Naming Conventions

ARexx programs can be named anything, but adopting a simple naming convention will make managing the programs much easier. Programs to be run from the CLI are usually given the file extension `.rexx` to distinguish them from programs written in other languages. Programs written as "macros" or "scripts" for a particular applications program should be given a file extension specific to that program. For example, a macro written for a communications program called "`MyComm`" might be named "`download.myc`". ARexx uses this file extension when it searches for a program file to be executed.

The `REXX:` Directory

You can designate one directory as the system-wide source for ARexx programs by defining a `REXX:` "device" with the `assign` command. This directory should reside on a volume that is usually mounted, such as `SYS:` or a hard disk. For example, the command "`assign rexx: sys:rexx`" defines the `REXX:` device as the `:rexx` directory on the system disk. Once defined, the `REXX:` device is searched after the current directory when looking for an ARexx program.

2-2 Program Examples

Before introducing the structure and syntax of the language, let's look at a few examples of ARexx programs. Readers familiar with other high-level programming languages should find many points of similarity between ARexx and other languages. In the examples that follow, new terms are highlighted in the text as they are introduced, and will be covered in depth in the next few chapters.

These short programs can be created using any text editor and then run from the Command Line Interface (CLI), or may simply be read as samples of the language. If the examples are to be run, first complete the installation procedures outlined in the previous section, and then start the ARexx resident process. Example programs can then be run by entering, for example, "`rx age`" at the CLI prompt.

We'll begin with a "Hello, World" program that simply displays a message on the console screen.

```
/* A simple program */
say 'Hello, World'
```

This program consists of a *comment* line that describes the program and an *instruction* that displays text on the console. For historical reasons, ARexx programs begin with a comment line; the initial "`/*`" says "I'm an ARexx program" to the interpreter when it searches for a program.

Instructions are language statements that denote a certain action to be performed, and always start with a *symbol*, in this case the word `say`. Symbols are translated to uppercase when the program is run, so the symbol `say` here is equivalent to `SAY`. Following `say` is an example of a *string*, which is a series of characters surrounded by quotes (`'`). Double quotes (`"`) could also have been used to define the string.

In the next program we'll display a prompt for input and then read some information from the user.

```
/* Calculate age in days */
say 'Please enter your age'
pull age
say 'You are about' age*365 'days old'
```

This program uses the `pull` instruction to read a line of input into a *variable* called `age`, which is then used with a `say` instruction. Variables are symbols that may be assigned a value. The words following `say` form an *expression* in which strings are joined and an arithmetic calculation is performed.

Note that the variable `age` did not have to be declared as a number; instead, its value was checked when it was actually used in the expression. To see what would happen if `age` wasn't a number, try rerunning the program with a non-numeric entry for the age. The resulting error message shows the line number and type of error that occurred, after which the program ends.

The next program introduces the do instruction, which allows program statements to be executed repeatedly. It also illustrates the *exponentiation* operator, which is used to raise a number to an integral power.

```
/* Calculate some squares and cubes     */
do i = 1 to 10          /* 10 iterations*/
   say i i**2 i**3      /* calculations */
   end                  /* end of loop  */
say 'all done'
```

The do instruction causes the statements between the do and end instructions to be executed 10 times. The variable i is the *index variable* for the loop, and is incremented by 1 for each iteration. The number following the symbol to is the *limit* for the do instruction, and could have been a full expression rather than just the constant 10. Note that the statements within the loop have been indented. This is not required by the language, but it makes the program more readable and is therefore good programming practice.

The subject of the next example is the if instruction, a often-used control statement that allows statements to be conditionally executed. The numbers from 1 to 10 are classified as even or odd by dividing them by 2 and then checking the remainder.

```
/* Even or odd? */
do i = 1 to 10
   if i//2 = 0 then type = 'even'
               else type = 'odd'
   say i 'is' type
   end
```

This example introduces the // arithmetic operator, which calculates the remainder after a division operation. The if instruction tests whether the remainder is 0 and executes the then branch if it is, thereby setting the variable type to "even." If the remainder was not 0, the alternative else branch is executed and type is set to "odd."

The next example introduces the concept of a *function*, which is a group of statements that can be executed by mentioning the function name in a suitable context. Functions are an important part of most programming languages, as they allow large, complex programs to be built from smaller modules. Functions are specified in an expression as a name followed by an open parenthesis. One or more expressions called *arguments* may follow the parenthesis; these are used to pass information to the function for processing.

```
/* Defining and calling a function      */
do i = 1 to 5
   say i square(i)      /* call square  */
   end
exit                /* all done          */
square:             /* function name     */
arg x               /* get the ''argument'' */
return x**2         /* square it and return */
```

The function `square` is defined in the lines following the *label* `square:` up through the `return` instruction. Two new instructions are introduced here: `arg` retrieves the value of the argument string, and `return` passes the function's result back to the point where the function was called.

One final example will suffice for now. A new instruction called `trace` is used here to activate the tracing features of ARexx.

```
/* Demonstrate "results" tracing */
trace results
sum=0;sumsq=0;
do i = 1 to 5
   sum = sum + i
   sumsq = sumsq + i**2
   end
say 'sum=' sum 'sumsq=' sumsq
```

When this program is run, the console displays the source lines as they are executed, and shows the final results of expressions. This makes it easy to tell what the program is really doing, and helps reduce the time required to develop and test a new program. One minor point is illustrated here: the third line shows two distinct statements separated by a semicolon (`;`). The semicolon is an example of a *special character*, characters that have particular meanings within ARexx programs.

The following chapters will present further information on the language statements illustrated here and will introduce others that have not been shown. Take heart, though; ARexx is a "small" language and there are relatively few words and rules to learn.

Chapter 3

Elements of the Language

This chapter introduces the rules and concepts that make up the REXX language. The intent is not to present a formalized definition, but rather to convey a practical understanding of how the langauge elements "fit together" to form programs.

3-1 Format

ARexx programs are composed of ASCII characters and may be created using any text editor. No special formatting of the program statements is required or imposed on the programmer.

3-2 Tokens

The smallest distinct entities or "words" of the langauge are called *tokens*. A token may be a series of characters, as in the symbol `MyName`, or just a single character like the "`+`" operator. Tokens can be categorized into *comments*, *symbols*, *strings*, *operators*, and *special characters*. Each of these groups are described below.

Comment Tokens

Any group of characters beginning with the sequence "`/*`" and ending with "`*/`" defines a *comment* token. Comments may be placed anywhere in a program and cost little in terms of execution speed, since they are stripped (removed) when the program is first scanned by the interpreter. Comments may be "nested" within one another, but each "`/*`" must have a matching "`*/`" in the program.
Examples:

```
/* Your basic comment          */
/* a  /* nested! */  comment   */
```

Symbol Tokens

Any group of the characters `a`–`z`, `A`–`Z`, `0`–`9`, and `.!?$_` defines a *symbol* token. Symbols are translated to uppercase as the program is scanned by the interpreter, so the symbol `MyName` is equivalent to `MYNAME`. Four types of symbols are recognized:

- *Fixed* symbols begin with a digit (`0`-`9`) or a period (`.`).
- *Simple* symbols do not begin with a digit, and do not contain any periods.
- *Stem* symbols have exactly one period at the end of the symbol name.
- *Compound* symbols include one or more periods in the interior of the name.

Stems and compound symbols have special properties that make them useful for building arrays and lists.

Symbol Values. The value used for a fixed symbol is always the symbol name itself (as translated to uppercase.) Simple, stem, and compound symbols are called *variables* and may be assigned a value during the course of the program execution. A variable is *uninitialized* if it has not yet been assigned a value; the value used for an uninitialized variable is just the variable name itself.
Examples:

```
123.45        /* a fixed symbol       */
MyName        /* same as MYNAME       */
a.            /* a stem symbol        */
a.1.Index     /* a compound symbol    */
```

String Tokens

A group of characters beginning and ending with a quote (') or double quote (") delimiter defines a *string* token. The delimiter character itself may be included within the string by a double-delimiter sequence ('' or ""). The number of characters in the string is called its length, and a string of length zero is called a *null string*. A string is treated as a *literal* in an expression; that is, its value is just the string itself.

Strings followed immediately by an "X" or "B" character that is not part of a longer symbol are classifed as *hex* or *binary* strings, respectively, and must be composed of hexadecimal digits (0-9,A-F) or binary digits (0,1). Blanks are permitted at byte boundaries for added readability. Hex and binary strings are convenient for specifying non-ASCII characters and for machine-specific information like addresses in a program. They are converted immediately to the "packed" internal form.
Examples:

```
"Now is the time"    /* a simple example     */
""                   /* a null string        */
'Can''t you see??'   /* Can't you see??      */
'4A 3B C0'X          /* a hex string         */
'00110111'b          /* binary for '7'       */
```

Operators

The characters ~+-*/=><&|^ may be combined in the sequences shown in Table 3.1 to form *operator* tokens. Operator sequences may include leading, trailing, and embedded blanks, all of which are removed when the program is scanned. In addition to the above characters, the *blank* character is treated as a concatenation operator if it follows a symbol or string and is not adjacent to an operator or special character.

Each operator has an associated priority that determines the order in which operations will be performed in an expression. Operators with higher priorities are performed before those with lower priorites.

Table 3.1 Operator Sequences

Sequence	Priority	Operator Definition
~	8	Logical NOT
+	8	Prefix Conversion
-	8	Prefix Negation
**	7	Exponentiation
*	6	Multiplication
/	6	Division
%	6	Integer Division
//	6	Remainder
+	5	Addition
-	5	Subtraction
||	4	Concatenation
(blank)	4	Blank Concatenation
==	3	Exact Equality
~==	3	Exact Inequality
=	3	Equality
~=	3	Inequality
>	3	Greater Than
>=, ~<	3	Greater Than or Equal To
<	3	Less Than
<=, ~>	3	Less Than or Equal To
&	2	Logical AND
|	1	Logical Inclusive OR
^, &&	1	Logical Exclusive OR

Special Character Tokens

The characters : ( ) ; , are each treated as a separate *special character* token and have particular meanings within an ARexx program. Blanks adjacent to these special characters are removed, except for those preceding an open parenthesis or following a close parenthesis.

Colon (:). A colon, if preceded by a symbol token, defines a *label* within the program. Labels are locations in the program to which control may be transferred under various conditions.

Opening and Closing Parentheses (()). Parentheses are used in expressions to group operators and operands into subexpressions, in order to override the normal operator priorities. An open parenthesis also serves to identify a *function call* within an expression; a symbol or string followed immediately by an open parenthesis defines a function name. Parentheses must always be balanced within a statement.

Semicolon (;). The semicolon acts as a program statement terminator. Several statements may be placed on a single source line if separated by semicolons.

Comma (,). A comma token acts as the continuation character for statements that must be entered on several source lines. It is also used to separate the argument expressions in a function call.

3-3 Clauses

Tokens are grouped together to form *clauses*, the smallest language unit that can be executed as a statement. Every clause in ARexx can be classified as either a *null*, *label*, *assignment*, *instruction*, or *command* clause. The classification process is very simple, since no more than two tokens are required to classify any clause. Assignment, instruction, and command clauses are jointly termed *statements*.

Clause Continuation. The end of a source line normally acts as the implicit end of a clause. A clause can be continued on the next source line by ending the line with a comma (,). The comma is then removed, and the next line is considered as a continuation of the clause. There is no limit to the number of continuations that may occur. String and comment tokens are automatically continued if a line ends before the closing delimiter has been found, and the "newline" character is not considered to be part of the token.

Multiple Clauses. Several clauses can be placed on a single line by separating them with semicolons (;).

Null Clauses

Lines consisting only of blanks or comments are called *null* clauses. They have no function in the execution of a program, except to aid its readability and to increment the source line count. Null clauses may appear anywhere in a program.
Example:

```
/* perform annuity calculations     */
```

Label Clauses

A symbol followed immediately by a colon defines a *label* clause. A label acts as a place-marker in the program, but no action occurs with the "execution" of a label. The colon is considered as an implicit clause terminator, so each label stands as a separate clause. Label clauses may appear anywhere in a program.
Examples:

```
start:      /* begin execution     */
syntax:     /* error processing    */
```

Assignment Clauses

Assignments are identified by a variable symbol followed by an "=" operator. In this context the operator's normal definition (an equality comparison) is overridden, and it becomes an assignment operator. The tokens to the right of the "=" are evaluated as an expression, and the result is *assigned to* (becomes the value of) the variable symbol.

Examples:

```
when = 'Now is the time'
answ = 3.14 * fact(5)
```

Instruction Clauses

Instructions begin with certain keyword symbols, each of which denotes a particular action to be performed. Instruction keywords are recognized as such only at the beginning of a clause, and may otherwise be used freely as symbols (although such use may become confusing at times.) The ARexx instructions are described in detail in Chapter 4.
Examples:

```
drop a b c            /* reset variables       */
say 'please'          /* a polite program      */
if j > 5 then leave;  /* several instructions  */
```

Command Clauses

Commands are any ARexx expression that can't be classified as one of the preceding types of clauses. The expression is evaluated and the result is issued as a command to an external *host*, which might be the native operating system or an application program. Commands are discussed in Chapter 5, and the details of the host command interface are given in Chapter 10.
Examples:

```
'delete' 'myfile'    /* a DOS command        */
'jump' current+10    /* an editor command?   */
```

Clause Classification

The process by which program lines are divided into clauses and then classified is important in understanding the operation of an ARexx program. The language interpreter splits the program source into groups of clauses as the program is read, using the end of each line as a clause separator and applying the continuation rule as required. These groups of one or more clauses are then tokenized, and each clause is classified into one of the above types. Note that seemingly small syntactic differences may completely change the semantic content of a statement. For example,

```
SAY 'Hello, Bill'
```

is an instruction clause and will display "`Hello, Bill`" on the console, but

```
''SAY 'Hello, Bill'
```

is a command clause, and will issue "`SAY Hello, Bill`" as a command to an external program. The presence of the leading null string changes the classification from an instruction clause to a command clause.

3-4 Expressions

Expression evaluation is an important part of ARexx programs, since most statements include at least one expression. Expressions are composed of strings, symbols, operators, and parentheses. Strings are used as literals in an expression; their value in an operation is just the string itself. Fixed symbols are also literals (remember that symbols are always translated to uppercase,) but variable symbols may have an assigned value. Operator tokens represent the predefined operations of ARexx; each operator has an associated priority that determines the order in which operations will be performed. Parentheses may be used to alter the normal order of evaluation in the expression, or to identify *function* calls. A symbol or string followed immediately by an open parenthesis defines the function name, and the tokens between the opening and (final) closing parenthesis form the *argument list* for the function.

For example, the expression "`J 'factorial is' fact(J)`" is composed of a symbol `J`, a blank operator, the string `'factorial is'`, another blank, the symbol `fact`, an open parenthesis, the symbol `J` again, and a closing parenthesis. `FACT` is a function name and `(J)` is its argument list, in this case the single expression `J`.

Symbol Resolution

Before the evaluation of an expression can proceed, the interpreter must obtain a value for each symbol in the expression. For fixed symbols the value is just the symbol name itself, but variable symbols must be looked up in the current symbol table. In the example above, the expression after symbol resolution would be "`3 'factorial is' FACT(3)`," assuming that the symbol `J` had the value `3`.

Suppose that the example above had been "`FACT(J) 'is' J 'factorial'`." Would the second occurrence of symbol `J` still resolve to `3` in this case? In general, function calls may have "side effects" that include altering the values of variables, so the value of `J` might have been changed by the call to `FACT`. In order to avoid ambiguities in the values assigned to symbols during the resolution process, ARexx guarantees a strict left-to-right resolution order. Symbol resolution proceeds irrespective of operator priority or parenthetical grouping; if a function call is found, the resolution is suspended while the function is evaluated. Note that it is possible for the same symbol to have more than one value in an expression.

Order of Evaluation

After all symbol values have been resolved, the expression is evaluated based on operator priority and subexpression grouping. Operators of higher priority are evaluated first. ARexx does not guarantee an order of evaluation among operators of equal priority, and does not employ a "fast path" evaluation of boolean operations. For example, in the expression

```
(1 = 2) & (FACT(3) = 6)
```

the call to the `FACT` function will be made, although it is clear that the final result will be `0`, since the first term of the AND operation is `0`.

3-5 Numbers and Numeric Precision

An important class of operands are those representing numbers. Numbers consist of the characters `0-9`, `.+-`, and blanks; an `e` or `E` may follow a number to indicate *exponential notation*, in which case it must be followed by a (signed) integer.

Both string tokens and symbol tokens may be used to specify numbers. Since the language is typeless, variables do not have to be declared as "numeric" before being used in an arithmetic operation. Instead, each value string is examined when it is used to verify that it represents a number. The following examples are all valid numbers:

```
33
"  12.3  "
0.321e12
' +  15.'
```

Note that leading and trailing blanks are permitted, and that blanks may be embedded between a "+" or "-" sign and the number body (but not within the body.)

Boolean Values

The numbers 0 and 1 are used to represent the boolean values **False** and **True**, respectively. The use of a value other than 0 or 1 when a boolean operand is expected will generate an error. Any number equivalent to 0 or 1, for example "`0.000`" or "`0.1E1`," is also acceptable as a boolean value.

Numeric Precision

ARexx allows the basic precision used for arithmetic calculations to be modified while a program is executing. The number of significant figures used in arithmetic operations is determined by the Numeric Digits environment variable, and may be modified using the `NUMERIC` instruction.

The number of decimal places used for a result depends on the operation performed and the number of decimal places in the operands. Unlike many languages, ARexx preserves trailing zeroes to indicate the precision of the result. If the total number of digits required to express a value exceeds the current Numeric Digits setting, the number is formatted in *exponential notation*. Two such formats are provided:

- In `SCIENTIFIC` notation, the exponent is adjusted so that a single digit is placed to the left of the decimal point.
- In `ENGINEERING` notation, the number is scaled so that the exponent is a multiple of 3 and the digits to the left of the decimal point range from 1 to 999.

The numeric precison and format can be set using the `NUMERIC` instruction.

3-6 Operators

Operators can be grouped into four categories:

- *Arithmetic* operators require one or two numeric operands, and produce a numeric result.
- *Concatenation* operators join two strings into a single string.
- *Comparison* operators require two operands, and produce a boolean (0 or 1) result.
- *Logical* operators require one or two boolean operands, and produce a boolean result.

Arithmetic Operators

The arithmetic operators are listed in Table 3.2 below. Note the inclusion of the integer division (%) and remainder (//) operators, along with the usual arithmetic operations. The result of an arithmetic operation is always formatted based on the current Numeric Digits setting, and will never have leading or trailing blanks.

Table 3.2 Arithmetic Operators

Sequence	Priority	Operation
+	8	Prefix Conversion
-	8	Prefix Negation
**	7	Exponentiation
*	6	Multiplication
/	6	Division
%	6	Integer Division
//	6	Remainder
+	5	Addition
-	5	Subtraction

Prefix Conversion (+). This unary operator converts the operand to and internal numeric form and formats the result based on the current Numeric Digits settings. This causes any leading and trailing blanks to be removed, and may result in a loss of precision.
Examples:

```
'   3.12  '          ==> 3.12
1.5001               ==> 1.500  /* If digits = 3 */
```

Prefix Negation (-). This unary operator negates the operand. The result is formatted based on the current Numeric Digits setting.

Examples:

```
-'   3.12  '          ==> -3.12
-1.5E2                ==> -150
```

Exponentiation (**). The left operand is raised to the power specified by the right operand, which must be an integer. The number of decimal places for the result is the product of the exponent and the number of decimal places in the base.
Examples:

```
2**3                  ==> 8
3**-1                 ==> .333333333
0.5**3                ==> 0.125
```

Multiplication (*). The product of two numbers is computed. The number of decimal places for the result is the sum of the decimal places of the operands.
Examples:

```
12 * 3                ==> 36
1.5 * 1.50            ==> 2.250
```

Division (/). The quotient of two numbers is computed. The number of decimal places for the result depends on the current setting of the numeric `DIGITS` variable; the number is formatted to the maximum precision required.
Examples:

```
6 / 3                 ==> 2
8 / 3                 ==> 2.66666667
```

Integer Division (%). The quotient of two numbers is computed, and the integer part of the quotient is used as the result.
Examples:

```
5 % 3                 ==> 1
-8 % 3                ==> -2
```

Remainder (//). The result is the remainder after the two operands are divided. The remainder for "`a//b`" is calculated as "`a-(a%b)*b`." If both operands are positive integers, this operation yields the usual "modulo" result.

Examples:

```
5 // 3                ==> 2
-5 // 3               ==> -2
5.1 // 0.2            ==> 0.1
```

Addition (+). The sum of two numbers is computed. The number of decimal places for the result is the larger of the decimal places of the operands.
Examples:

```
12 + 3                ==> 15
3.1 + 4.05            ==> 7.15
```

Subtraction (-). The difference of two numbers is computed. As in the case of addition, the number of decimal places for the result is the larger of the decimal places of the operands.
Examples:

```
12 - 3                ==> 9
5.55 - 1.55           ==> 4.00
```

Concatenation Operators

ARexx defines two concatenation operators, both of which require two operands. The first, identified by the operator sequence "||", joins two strings into a single string with no intervening blank. The second concatenation operation is identified by the blank operator, and joins the two operand strings with one intervening blank.

An implicit concatenation operator is recognized when a symbol and a string are directly abutted in an expression. Concatenation by abuttal uses the "||" operator, and behaves exactly as though the operator had been provided explicitly.
Examples:

```
'why me,' || 'Mom?'   ==> why me,Mom?
'good' 'times'        ==> good times
one'two'three         ==> ONEtwoTHREE
```

Comparison Operators

Comparisons are performed in one of three modes, and always result in a boolean value (0 or 1.)

- *Exact* comparisons proceed character-by-character, including any leading blanks that may be present.
- *String* comparisons ignore leading blanks, and pad the shorter string with blanks if necessary.

- *Numeric* comparisons first convert the operands to an internal numeric form using the current Numeric Digits setting, and then perform a standard arithmetic comparison.

Except for the exact equality and exact inequality operators, all comparison operators dynamically determine whether a string or numeric comparison is to be performed. A numeric comparison is performed if both operands are valid numbers; otherwise, the operands are compared as strings.

Table 3.3 Comparison Operators

Sequence	Priority	Operation	Mode
==	3	Exact Equality	Exact
~==	3	Exact Inequality	Exact
=	3	Equality	String/Numeric
~=	3	Inequality	String/Numeric
>	3	Greater Than	String/Numeric
>=,~<	3	Greater Than or Equal	String/Numeric
<	3	Less Than	String/Numeric
<=,~>	3	Less Than or Equal	String/Numeric

Logical (Boolean) Operators

ARexx defines the four logical operations NOT, AND, OR, and Exclusive OR, all of which require boolean operands and produce a boolean result. Boolean operands must have values of either 0 (**False**) or 1 (**True.**) An attempt to perform a logical operation on a non-boolean operand will generate an error.

Table 3.4 Logical Operators

Sequence	Priority	Operation
~	8	NOT (Inversion)
&	2	AND
|	1	OR
^, &&	1	Exclusive OR

3-7 Stems and Compound Symbols

Stems and compound symbols have special properties that allow for some interesting and unusual programming. A compound symbol can be regarded as having the structure `stem.`n_1`.`n_2`.`$n_3 \ldots n_k$ where the leading name is a stem symbol and each node $n_1 \ldots n_k$ is a fixed or simple symbol. Whenever a compound symbol appears in a program, its name is *expanded* by replacing each node with its current value as a (simple) symbol. The value string may consist of any characters, including embedded blanks, and is not converted to uppercase. The result of the expansion is a new name that is used in place of the compound symbol. For example, if `J` has the value 3 and `K` has the value 7, then the compound symbol `a.j.k` will expand to `A.3.7`.

Stem symbols provide a way to initialize a whole class of compound symbols. When an assignment is made to a stem symbol, it assigns that value to all possible compound

symbols derived from the stem. Thus, the value of a compound symbol depends on the prior assignments made to itself or its associated stem.

Compound symbols can be regarded as a form of "associative" or "content-addressable" memory. For example, suppose that you needed to store and retrieve a set of names and telephone numbers. The conventional approach would be to set up two arrays `NAME` and `NUMBER`, each indexed by an integer running from one to the number of entries. A number would be "looked up" by scanning the name array until the given name was found, say in `NAME.12`, and then retrieving `NUMBER.12`. With compound symbols, the symbol `NAME` could hold the name to be looked-up, and `NUMBER.NAME` would then expand to `NUMBER.Bill` (for example), which be the corresponding number.

Of course, compound symbols can also be used as conventional indexed arrays, with the added convenience that only a single assignment (to the stem) is required to initialize the entire array.

3-8 The Execution Environment

The ARexx interpreter provides a uniform execution environment by running each program as a separate task (actually, as a DOS *process*) in the Amiga's multitasking operating system. This allows for a flexible interface between an external host program and the interpreter, as the host can either proceed concurrently with its operations or can simply wait for the interpreted program to finish.

The External Environment

The external environment of a program includes its task (process) structure, input and output streams, and current directory. When each ARexx task is created, it inherits the input and output streams and current directory from its *client*, the external program that invoked the ARexx program. The current directory is used as the starting point in a search for a program or data file.

External Programs. The external environment usually includes one or more external programs with which the ARexx program may communicate. Any program that supports a suitable interface can receive commands from ARexx programs. The command interface is discussed in Chapter 5.

The Internal Environment

The internal environment of an ARexx program consists of a static global structure and one or more storage environments. The global data values are fixed at the time the program is invoked, and include the argument strings, program source code, and static data strings. The storage environment includes the symbol table used for variable values, the numeric options, trace option, and host address strings. While the global environment is unique, there may be many storage environments during the course of the program execution. Each time an internal function is called a new storage environment is activated and initialized. The initial values for most fields are inherited from the previous environment, but values may be changed afterwards without affecting the caller's environment. The new environment persists until control returns from the function.

Argument Strings. A program may receive one or more argument strings when it is first invoked. These arguments persist for the duration of the program and are never altered. The number of arguments a program receives depends in part on the mode of invocation. ARexx programs invoked as commands normally have only one argument string, although the "command tokenization" option may provide more than one. A program invoked as a function can have any number of arguments if called as an internal function, but external functions are limited to a maximum of 15 arguments.

The argument strings can be retrieved using either the `ARG` instruction or the `ARG()` Built-In function. `ARG()` can also return the total number of arguments, or the status (as "exists" or "omitted") of a particular argument.

The Symbol Table. Every storage environment includes a *symbol table* to store the value strings that have been assigned to variables. This symbol table is organized as a two-level *binary tree*, a data structure that provides an efficient look-up mechanism. The primary level stores entries for simple and stem symbols, and the secondary level is used for compound symbols. All of the compound symbols associated with a particular stem are stored in one tree, with the root of the tree held by the entry for the stem.

Symbols are not entered into the table until an assignment is made to the symbol. Once created, entries at the primary level are never removed, even if the symbol subsequently becomes uninitialized. Secondary trees are released whenever an assignment is made to the stem associated with the tree.

For the most part ARexx programmers need not be concerned with the details of storage environments except to understand what values are saved when a function is called. Applications developers who need to manipulate environment values should refer to the structure definitions in the INCLUDE files provided on the ARexx distribution disk.

Input and Output

Most computer programs require some means of communicating with the outside world, either to accept input data or to pass along results. The REXX language includes only a minimal specification of input and output (I/O) operations, leaving the choice of additional functionality to the language implementor. This is in keeping with the design of many computer languages. For instance, the "C" langauge has no statements dedicated to I/O, but instead relies on a standardized set of I/O functions.

ARexx extends the I/O facilities of REXX by providing Built-In functions to manipulate external files. Files are referenced by a *logical name* associated with the file when it is first opened. The initial input and output streams are given the names `STDIN` and `STDOUT`.

ARexx maintains a list of all of the files opened by a program and automatically closes them when the program finishes. There is no limit to the number of files that may be open simultaneously.

Resource Tracking

ARexx provides complete tracking for all of the dynamically-allocated resources that it uses to execute a program. These resources include memory space, DOS files and related structures, and the message port structures supported by ARexx. The tracking system was designed to allow a program to "bail out" at any point (perhaps due to an execution error) without leaving any hanging resources.

It is possible to go outside of the interpreter's resource tracking net by making calls directly to the Amiga's operating system from within an ARexx program. In these cases it is the programmer's responsibility to track and return all of the allocated resources. ARexx provides a special interrupt facility so that a program can retain control after an execution error, perform the required cleanup, and then make an orderly exit. Chapter 7 has information on the ARexx interrupt system.

Chapter 4

Instructions

Instruction clauses are identified by an initial keyword symbol that is not followed by a colon (:) or an equals (=) operator. Each instruction signifies a specific action, and may be followed by one or more subkeywords, expressions, or other instruction-specific information. Instruction keywords and subkeywords are recognized only in this specific context, and are therefore not "reserved words" in the usual sense of the term. Keywords may be used freely as variables or function names, although such usage may become confusing at times.

In the descriptions that follow, keywords are shown in uppercase and optional parts of the instruction are enclosed in brackets. Alternative selections are separated by a vertical bar (|), and required alternatives are enclosed in braces ({}).

4-1 ADDRESS

Usage: ADDRESS *[symbol | string | [[*VALUE*] [expression]]*
This instruction specifies a *host address* for commands issued by the interpreter. A host address is the name associated with an external program to which commands can be sent; external hosts are described in Chapter 5. ARexx maintains two host addresses: a "current" and a "previous" value. Whenever a new host address is supplied, the "previous" address is lost, and the "current" address becomes the "previous" one. These host addresses are part of a program's storage environment and are preserved across internal function calls. The current address can be retrieved with the Built-In function ADDRESS(). There are four distinct forms for the ADDRESS instruction:

- ADDRESS *{string | symbol} expression.* The expression is evaluated and the result is issued to the host specified by the string or symbol, which is taken as a literal. No changes are made to the current or previous address strings. This provides a convenient way to issue a single command to an external host without disturbing the current host addresses. The return code from the command is treated as it would be from a command clause.
- ADDRESS *{string | symbol}*. The string or symbol, taken as a literal, specifies the new host address. The current host address becomes the previous address.
- ADDRESS *[*VALUE*] expression.* The result of the expression specifies the new host address, and the current address becomes the previous address. The VALUE keyword may be omitted if the first token of the expression is not a symbol or string.
- ADDRESS. This form interchanges the current and previous hosts. Repeated execution will therefore "toggle" between the two host addresses.

Examples:

```
address edit            /* set an new host address      */
address edit 'top'      /* move to the top              */
address VALUE edit n    /* compute a new host address   */
address                 /* swap current and previous    */
```

4-2 ARG

Usage: ARG *[template] [,template ...]*
ARG is a shorthand form for the PARSE UPPER ARG instruction. It retrieves one or more of the argument strings available to the program, and assigns values to the variables in the template. The number of argument strings available depends on the whether the program was invoked as a command or a function. Command invocations normally have only one argument string, but functions may have up to 15. The argument strings are not altered by the ARG instruction.

The structure and processing of templates is described briefly with the PARSE instruction, and in greater depth in Chapter 8.
Example:

```
arg first,second /* fetch arguments */
```

4-3 BREAK

Usage: BREAK
The BREAK instruction is used to exit from the range of a DO instruction or from within an INTERPRETed string, and is valid only in these contexts. If used within a DO statement, BREAK exits from the innermost DO statement containing the BREAK. This contrasts with the otherwise similar LEAVE instruction, which exits only from an iterative DO.
Example:

```
do                      /* begin block          */
   if i>3 then break    /* all done?            */
   a = a + 1
   y.a = name
   end                  /* end block            */
```

4-4 CALL

Usage: CALL *{symbol | string} [expression] [,expression,...]*
The CALL instruction is used to invoke an internal or external function. The function name is specified by the symbol or string token, which is taken as a literal. Any expressions that follow are evaluated and become the arguments to the called function. The value returned by the function is assigned to the special variable RESULT. It is not an error if a result string is not returned; in this case the variable RESULT is DROPped (becomes uninitialized.)

The linkage to the function is established dynamically at the time of the call. ARexx follows a specific search order in attempting to locate the called function; this process is described in Chapter 6.
Example:

```
call center name,length+4,'+'
```

4-5 DO

Usage: DO *[var=exp [*TO *exp] [*BY *exp]] [*FOR *exp] [*FOREVER*] [*WHILE *exp* | UNTIL *exp]*
The DO instruction begins a group of instructions to be executed as a block. The range of the DO instruction includes all statements up to and including an eventual END instruction. There are two basic forms of the instruction:

- The DO keyword by itself defines a block of instructions to be executed once.
- If any iteration specifiers follow the DO keyword, the block of instructions is executed repeatedly until a termination condition occurs.

An iterative DO instruction is sometimes called a "loop", since the interpreter "loops back" to perform the instruction repeatedly. The various parts of the DO instruction are described below.

Initializer expression. An initializer expression of the form "`variable=expression`" defines the *index variable* of the loop. The expression is evaluated when the DO range is first activated, and the result is assigned to the index variable. On subsequent iterations an expression of the form "`variable = variable + increment`" is evaluated, where the increment is the result of the BY expression. If specified, the initializer expression must precede any of the other subkeywords.

BY expression. The expression following a BY symbol defines the increment to be added to the index variable in each subsequent iteration. The expression must yield a numeric result, which may be positive or negative and need not be an integer. The default increment is 1.

TO expression. The result of the TO expression specifies the upper (or lower) limit for the index variable. At each iteration the index variable is compared to the TO result. If the increment (BY result) is positive and the variable is greater than the limit, the DO instruction terminates and control passes to the statement following the END instruction. Similarly, the loop terminates if the increment is negative and the index variable is less than the limit.

FOR expression. The FOR expression must yield a positive whole number when evaluated, and specifies the maximum number of iterations to be performed. The loop terminates when this limit is reached irrespective of the value of the index variable.

FOREVER. The FOREVER keyword can be used if an iterative DO instruction is required but no index variable is necessary. Presumably the loop will be terminated by a LEAVE or BREAK instruction contained within the loop body.

WHILE expression. The WHILE expression is evaluated at the beginning of each iteration and must result in a boolean value. The iteration proceeds if the result is 1; otherwise, the loop terminates.

UNTIL expression. The UNTIL expression is evaluated at the end of each iteration and must result in a boolean value. The instruction continues with the next iteration if the result is 0, and terminates otherwise.

The initializer, BY, TO, and FOR expressions are evaluated only when the instruction is first activated, so the increment and limits are fixed throughout the execution. Note that a limit need not be supplied; for example, the instruction "DO i=1" will simply count away forever. Note also that only one of the WHILE or UNTIL keywords can be specified.
Example:

```
do i=1 to limit for 5 while time < 50
   y.i = i*time
   end
```

4-6 DROP

Usage: DROP *variable [variable...]*
The specified variable symbols are reset to their uninitialized state, in which the value of the variable is the variable name itself. It is not an error to DROP a variable that is already uninitialized. DROPping a stem symbol is equivalent to DROPping the values of all possible compound symbols derived from that stem.
Example:

```
a = 123                 /* assign a value      */
drop a b                /* drop some           */
say a b                 /* ==> A B             */
```

4-7 ECHO

Usage: ECHO *[expression]*
The ECHO instruction is a synonym for the SAY instruction. It displays the expression result on the console.
Example:

```
echo "You don't SAY!"
```

4-8 ELSE

Usage: ELSE *[;] [conditional statement]*
The ELSE instruction provides the alternative conditional branch for an IF statement. It is valid only within the range of an IF instruction, and must follow the conditional statement of the THEN branch. If the THEN branch wasn't executed, the statement following the ELSE clause is performed.

Binding. ELSE clauses always bind to the nearest (preceding) IF statement. It may be necessary to provide "dummy" ELSE clauses for the inner IF ranges of a compound IF statement in order to allow alternative branches for the outer IF statements. In this case it is not sufficient to follow the ELSE with a semicolon or a null clause. Instead, the NOP (no-operation) instruction can be used for this purpose.

Example:

```
if 1 > 2 then say 'really?'
         else say 'I thought so'
```

4-9 END

Usage: `END` *[variable]*
The `END` instruction terminates the range of a `DO` or `SELECT` instruction. If the optional variable symbol is supplied, it is compared to the index variable of the `DO` statement (which must therefore be iterative). An error is generated if the symbols do not match, so this provides a simple mechanism for matching the `DO` and `END` statements.
Example:

```
do i=1 to 5              /* index variable is I  */
   say i
   end i                 /* end ''I'' loop       */
```

4-10 EXIT

Usage: `EXIT` *[expression]*
The `EXIT` instruction terminates the execution of a program, and is valid anywhere within a program. The evaluated expression is passed back to the caller as the function or command result.

Results Processing. The processing of the `EXIT` result depends on whether a result string was requested by the calling program, and whether the current invocation resulted from a command or function call. If a result string was requested, the expression result is copied to a block of allocated memory and a pointer to the block is returned as the secondary result of the call.

If the caller did not request a result string, and the program was invoked as a command, then an attempt is made to convert the expression result to an integer. This value is then returned as the primary result, with 0 as the secondary result. This allows the `EXIT` expression to be interpreted as a "return code" by the caller. Refer to Chapter 10 for further information on the data structures used to return the result string.
Examples:

```
exit                     /* no result needed     */
exit 12                  /* an error return?     */
```

4-11 IF

Usage: `IF` *expression [*`THEN`*] [;] [conditional statement]*
The `IF` instruction is used in conjunction with `THEN` and `ELSE` instructions to conditionally execute a statement. The result of the expression must be a boolean value. If the result is 1 (**True**), the statement following the `THEN` symbol is executed; otherwise, control passes to the next statement (which might be an `ELSE` clause.) The `THEN` keyword need not immediately follow the `IF` expression, but may appear as a separate clause. The instruction

is actually analyzed as "IF expression; THEN; statement;." In essence, the IF statement begins a syntactic range and establishes the test condition that determines whether subsequent THEN or ELSE clauses will be performed.

Any valid statement may follow the THEN symbol; in particular, a "DO; ... END;" group allows a series of statements to be performed conditionally.
Example:

```
if result < 0 then exit  /* all done? */
```

4-12 INTERPRET

Usage: INTERPRET *expression*
The expression is evaluated and the result is executed as one or more program statements. The statements are considered as a group, as though surrounded by a "DO; ...;END" combination. Any statements can be included in the INTERPRETed source, including DO or SELECT instructions.

An INTERPRET instruction activates a control range when it is executed, which serves as a "fence" for LEAVE and ITERATE instructions. These instructions can therefore be used only with DO-loops defined within the INTERPRET. The BREAK instruction can be used to terminate the processing of INTERPRETed statements. While it is not an error to include label clauses within the interpreted string, only those labels defined in the original source code are searched during a transfer of control.

The INTERPRET instruction can be used to solve programming problems in interesting and novel ways. Programs can be constructed dynamically and then executed using this instruction, or program fragments may be passed as arguments to functions, which then INTERPRET them.
Example:

```
inst = 'say'             /* an instruction       */
interpret inst hello     /* ... "say HELLO"      */
```

4-13 ITERATE

Usage: ITERATE *[variable]*
The ITERATE instruction terminates the current iteration of a DO instruction and begins the next iteration. Effectively, control passes to the END statement and then (depending on the outcome of the UNTIL expression) back to the DO statement. The instruction normally acts on the innermost iterative DO range containing the instruction. An error results if the LEAVE instruction is not contained within an iterative DO instruction.

The optional variable symbol specifies which DO range is to be exited, in the event that several nested ranges exist. The variable is taken as a literal and must match the index variable of a currently active DO instruction. An error results if no such matching DO instruction is found.

Example:

```
do i=1 to 3
   if i = j then iterate i
   end
```

4-14 LEAVE

Usage: LEAVE *[variable]*
LEAVE forces an immediate exit from the iterative DO range containing the instruction. An error results if the LEAVE instruction is not contained within an iterative DO instruction.

The optional variable symbol specifies which DO range is to be exited, in the event that several nested ranges exist. The variable is taken as a literal and must match the index variable of a currently active DO instruction. An error results if no such matching DO instruction is found.
Example:

```
do i = 1 to limit
   if i > 5 then leave   /* maximum iterations   */
   end
```

4-15 NOP

Usage: NOP
The NOP or "no-operation" instruction does just that: nothing. It is provided to control the binding of ELSE clauses in compound IF statements.
Example:

```
if i = j then           /* first (outer) IF     */
   if j = k then a = 0  /* inner IF             */
            else nop    /* binds to inner IF    */
   else a = a + 1       /* binds to outer IF    */
```

4-16 NUMERIC

Usage: NUMERIC {DIGITS | FUZZ} *expression*
or: NUMERIC FORM {SCIENTIFIC | ENGINEERING}
This instruction sets options relating to the numeric precision and format. The valid forms of the NUMERIC instruction are:

- NUMERIC DIGITS *expression.* Specifies the number of digits of precision for arithmetic calculations. The expression must evaluate to a positive whole number.
- NUMERIC FUZZ *expression.* Specifies the number of digits to be ignored in numeric comparison operations. This must be a positive whole number that is less than the current DIGITS setting.
- NUMERIC FORM SCIENTIFIC. Specifies that numbers that require exponential notation be expressed in SCIENTIFIC notation. The exponent is adjusted so that the mantissa (for non-zero numbers) is between 1 and 10. This is the default format.

- NUMERIC FORM ENGINEERING. Selects ENGINEERING format for numbers that require exponential notation. ENGINEERING format normalizes a number so that its exponent is a multiple of three and the mantissa (if not 0) is between 1 and 1000.

The numeric options are preserved when an internal function is called.
Examples:

```
numeric digits 12       /* precision          */
numeric form scientific /* format             */
```

4-17 OPTIONS

Usage: OPTIONS *[*FAILAT *expression]*
or: OPTIONS *[*PROMPT *expression]*
or: OPTIONS *[*RESULTS*]*
The OPTIONS instruction is used to set various internal defaults. The FAILAT expression sets the limit at or above which command return codes will be signalled as errors, and must evaluate to an integer value. The PROMPT expression provides a string to be used as the prompt with the PULL (or PARSE PULL) instruction. The RESULTS keyword indicates that the interpreter should request a result string when it issues commands to an external host.

The internal options controlled by this instruction are preserved across function calls, so an OPTIONS instruction can be issued within an internal function without affecting the caller's environment. If no keyword is specified with the OPTIONS instruction, all controlled options revert to their default settings.
Example:

```
options failat 10
options prompt "Yes Boss?"
options results
```

4-18 OTHERWISE

Usage: OTHERWISE *[;] [conditional statement]*
This instruction is valid only within the range of a SELECT instruction, and must follow the "WHEN ... THEN" statements. If none of the preceding WHEN clauses have succeeded, the statement following the OTHERWISE instruction is executed. An OTHERWISE is not mandatory within a SELECT range. However, an error will result if the OTHERWISE clause is omitted and none of the WHEN instructions succeed.
Example:

```
select
   when i=1 then say 'one'
   when i=2 then say 'two'
   otherwise say 'other'
   end
```

4-19 PARSE

Usage: PARSE *[*UPPER*] inputsource [template] [,template...]*
The PARSE instruction provides a mechanism to extract one or more substrings from a string and assign them to variables. The input string can come from a variety of sources, including argument strings, an expression, or from the console. The *template* provides both the variables to be given values and the way to determine the value strings. The template may be omitted if the instruction is intended only to create the input string. The different options of the instruction are described below.

Input Sources

The sources for the input strings are specified by the keyword symbols listed below. When multiple templates are supplied, each template receives a new input string, although for some source options the new string will be identical to the previous one. The input source string is copied before being parsed, so the original strings are never altered by the parsing process.

UPPER. This optional keyword may be used with any of the input sources, and specifies that the input string is to be translated to uppercase before being parsed. It must be the first token following PARSE.

- ARG. This input option retrieves the argument strings supplied when the program was invoked. Command invocations normally have only a single argument string, but functions may have up to 15 argument strings. Multiple templates may be given to retrieve successive argument strings.
- EXTERNAL. The input string is read from the console. If multiple templates are supplied, each template will read a new string. This source option is the same as PULL.
- NUMERIC. The current numeric options are placed in a string in the order DIGITS, FUZZ, and FORM, separated by a single space.
- PULL. Reads a string from the input console. If multiple templates are supplied, each template will read a new string.
- SOURCE. The "source" string for the program is retrieved. This string is formatted as "{COMMAND | FUNCTION} {0 | 1} *called resolved ext host.*" The first token indicates whether the program was invoked as a command or as a function. The second token is a boolean flag indicating whether a result string was requested by the caller. The *called* token is the name used to invoke this program, while the *resolved* token is the final resolved name of the program. The *ext* token is the file extension to be used for searching (the default is "REXX"). Finally, the *host* token is the initial host address for commands.
- VALUE *expression* WITH. The input string is the result of the supplied expression. The WITH keyword is required to separate the expression from the template. The expression result may be parsed repeatedly by using multiple templates, but the expression is not reevaluated.
- VAR *variable*. The value of the specified variable is used as the input string. When multiple templates are provided, each template uses the current value of the variable.

This value may change if the variable is included as an assignment target in any of the templates.

- `VERSION`. The current configuration of the ARexx interpreter is supplied in the form "`ARexx` *version cpu mpu video freq*". The *version* token is the release level of the interpreter, formatted as `V1.0`. The *cpu* token indicates the processor currently running the program, and will be one of the values `68000`, `68010`, or `68020`. The *mpu* token will be either `NONE` or `68881` depending on whether a math coprocessor is available on the system. The *video* token will indicate either `NTSC` or `PAL`, and the *freq* token gives the clock (line) frequency as either `60HZ` or `50HZ`.

Templates

Parsing is controlled by a template, which may consist of symbols, strings, operators, and parentheses. During the parsing operation the input string is split into substrings that are assigned to the variable symbols in the template. The process continues until all of the variables in the template have been assigned a value; if the input string is "used up", any remaining variables are given null values.

Templates are described in depth in Chapter 8, so only a simplified description is presented here. The goal of the parsing operation is to associate a "current" and "next" position with each variable symbol in the template. The substring between these positions is then assigned as the value to the variable. There are three basic methods used to determine the value strings.

Parsing by Tokenization. When a variable in the template is followed immediately by another variable, the value string is determined by breaking the input string into words separated by blanks. Each word is assigned to a variable in the template.

Values determined by tokenization will never have leading or trailing blanks. Normally the last variable in the template receives the untokenized remainder of the input string, since it is not followed by a symbol. A "placeholder" symbol, signified by a period (.), may be used to force tokenization. Placeholders behave like variables in the template except that they are never actually assigned a value.
Example:

```
/* Numeric string is: "9 0 SCIENTIFIC"   */
parse numeric digits fuzz form .
say digits              /* => 9                */
say fuzz                /* => 0                */
say form                /* => SCIENTIFIC       */
```

Parsing by Position. If the fields in the input string have known positions, value strings can be specified by absolute or relative positions. Relative positions are indicated by a number preceded by a "+" or "-" operator. Each positional marker updates the scan position in the string. The value assigned to a variable is the string from the current position up to, but not including, the next position in the string.

Example:

```
/* assume argument is "1234567890"              */
parse arg 1 a 3 b +2 1 c
say a b c              /* ==> 12 34 1234567890 */
```

Parsing with Patterns. Fields in the input string separated by specific characters or strings can be parsed using a pattern, which is matched against the input string. A pattern is specified in the template as a string token, or alternatively as a symbol enclosed in parentheses. The position in the parse string matched by the pattern determines the value strings. The pattern is removed from the input string when a match is found; this is the only parsing operation that modifies the input string.
Example:

```
check = 'one,two,three'
parse var check a ',' b ',' c
say a b c              /* ==> one two three    */
```

4-20 PROCEDURE

Usage: PROCEDURE [EXPOSE *variable [variable ...]]*
The PROCEDURE instruction is used within an internal function to create a new symbol table. This protects the symbols defined in the caller's environment from being altered by the execution of the function. PROCEDURE is usually the first statement within the function, although it is valid anywhere withing the function body. It is an error to execute two PROCEDURE statements within the same function.

Exposing Variables. The EXPOSE subkeyword provides a selective mechanism for accessing the caller's symbol table, and for passing global variables to a function. The variables following the EXPOSE keyword are taken to refer to symbols in the caller's table. Any subsequent changes made to these variables will be reflected in the caller's environment.

The variables in the EXPOSE list may include stems or compound symbols, in which case the ordering of the variables becomes significant. The EXPOSE list is processed from left to right, and compound symbols are expanded based on the values in effect in the new generation. For example, suppose that the value of the symbol J in the previous generation is 123, and that J is uninitialized in the new generation. Then PROCEDURE EXPOSE J A.J will expose J and A.123, whereas PROCEDURE EXPOSE A.J J will expose A.J and J. Exposing a stem has the effect of exposing all possible compound symbols derived from that stem.
Example:

```
fact: procedure          /* a recursive function */
   arg i
   if i <= 1
      then return 1
      else return i*fact(i-1)
```

4-21 PULL

Usage: PULL *[template] [,template...]*
This is a shorthand form of the PARSE UPPER PULL instruction. It reads a string from the input console, translates it to uppercase, and parses it using the template. Multiple strings can be read by supplying additional templates. The instruction will read from the console even if no template is given.

Templates are described briefly with the PARSE instruction and in greater depth in Chapter 8.
Example:

```
pull first last .       /* read names          */
```

4-22 PUSH

Usage: PUSH *[expression]*
The PUSH instruction is used to prepare a stream of data to be read by a command shell or other program. It appends a "newline" to the result of the expression and then stacks or "pushes" it into the STDIN stream. Stacked lines are placed in the stream in "last-in, first-out" order, and are then available to be read just as though they had been entered interactively. For example, after issuing the instructions

```
push line 1
push line 2
push line 3
```

the stream would be read in the order "`line 3`," "`line 2`," and "`line 1`."

There are several restrictions governing the use of the PUSH instruction and its alter ego QUEUE. These instructions use a special I/O mechanism to accomplish their task, and as a result can be used only with an interactive (stream-model) I/O device like a console or pipe. The stream must be managed by with a DOS handler that supports the special ACTION_STACK (for PUSH) or ACTION_QUEUE (for QUEUE) command.

PUSH allows the STDIN stream to be used as a private scratchpad to prepare data for subsequent processing. For example, several files could be concatenated with delimiters between them by simply reading the input files, PUSHing the lines into the stream, and inserting a delimiter where required. Once the stacked lines are exhausted, the stream reverts to its normal source of data.
Example:

```
/* Stack commands for compile and link */
push "blink c.o+main.o library amiga.lib to myprog"
push "cc main"
```

4-23 QUEUE

Usage: QUEUE *[expression]*
The QUEUE instruction is used to prepare a stream of data to be read by a command shell or other program. It is very similar to the preceding PUSH instruction, and differs only in that the data lines are placed in the STDIN stream in "first-in, first-out" order. In this case the instructions

```
queue line 1
queue line 2
queue line 3
```

would be read in the order "line 1," "line 2," and "line 3." The QUEUEd lines always precede all interactivly-entered lines, and always follow any PUSHed (stacked) lines.

The same restrictions noted with the use of the PUSH instruction apply to the QUEUE instruction. The queueing mechanism uses the ACTION_QUEUE command, so the DOS handler associated with the STDIN stream must support this command.

In most cases the choice of whether to use PUSH or QUEUE is just a matter of convenience or personal preference. Each of them provides a "scratch pad" facility similar to that provided by an I/O pipe, but useful within one program or task rather than just for interprocess communications.
Example:

```
/* Queue commands for compile and link */
queue "cc main"
queue "blink c.o+main.o library amiga.lib to myprog"
```

4-24 RETURN

Usage: RETURN *[expression]*
RETURN is used to leave a function and return control to the point of the previous function invocation. The evaluated expression is returned as the function result. If an expression is not supplied, an error may result in the caller's environment. Functions called from within an expression must return a result string, and will generate an error if no result is available. Function invoked by the CALL instruction need not return a result.

A RETURN issued from the base environment of a program is not an error, and is equivalent to an EXIT instruction. Refer to the EXIT instruction for a description of how result strings are passed back to an external caller.
Example:

```
return 6*7              /* the answer          */
```

4-25 SAY

Usage: SAY *[expression]*
The result of the evaluated expression is written to the output console, with a "newline" character appended. If the expression is omitted, a null string is sent to the console.
Example:

```
say 'The answer is ' value
```

4-26 SELECT

Usage: SELECT
This instruction begins a group of instructions containing one or more WHEN clauses and possibly a single OTHERWISE clause, each followed by a conditional statement.

Only one of the conditional statements within the SELECT group will be executed. Each WHEN statement is executed in succession until one succeeds; if none succeeds, the OTHERWISE statement is executed. The SELECT range must be terminated by an eventual END statement.

Example:

```
select
   when i=1 then say 'one'
   when i=2 then say 'two'
   otherwise say 'other'
   end
```

4-27 SHELL

Usage: SHELL *[symbol | string] [expression]*
The SHELL instruction is a synonym for the ADDRESS instruction.
Example:

```
shell edit              /* set host to 'EDIT'   */
```

4-28 SIGNAL

Usage: SIGNAL {ON |OFF} *condition*
or: SIGNAL *[VALUE] expression*
There are two forms of the SIGNAL instruction. The first form illustrated controls the state of the internal interrupt flags. Interrupts allow a program to detect and retain control when certain errors occur, and are discussed in Chapter 7. In this form SIGNAL must be followed by one of the keywords ON or OFF and one of the condition keywords listed below. The interrupt flag specified by the condition symbol is then set to the indicated state. The valid signal conditions are:

- BREAK_C A "control-C" break was detected.
- BREAK_D A "control-D" break was detected.
- BREAK_E A "control-E" break was detected.

- BREAK_F A "control-F" break was detected.
- ERROR A host command returned a non-zero code.
- HALT An external HALT request was detected.
- IOERR An error was detected by the I/O system.
- NOVALUE An uninitialized variable was used.
- SYNTAX A syntax or execution error was detected.

The condition keywords are interpreted as labels to which control will be transferred if the selected condition occurs. For example, if the ERROR interrupt is enabled and a command returns a non-zero code, the interpreter will transfer control to the label ERROR:. The condition label must of course be defined in the program; otherwise, an immediate SYNTAX error results and the program exits.

In the second form of the instruction, the tokens following SIGNAL are evaluated as an expression. An immediate interrupt is generated that transfers control to the label specified by the expression result. The instruction thus acts as a "computed goto."

Interrupts. Whenever an interrupt occurs, all currently active control ranges (IF, DO, SELECT, INTERPRET, or interactive TRACE) are dismantled before the transfer of control. Thus, the transfer cannot be used to jump into the range of a DO-loop or other control structure. Only the control structures in the current environment are affected by a SIGNAL condition, so it is safe to SIGNAL from within an internal function without affecting the state of the caller's environment.

Special Variables. The special variable SIGL is set to the current line number whenever a transfer of control occurs. The program can inspect SIGL to determine which line was being executed before the transfer. If an ERROR or SYNTAX condition causes an interrupt, the special variable RC is set to the error code that triggered the interrupt. For the ERROR condition, this code is usually an error severity level. The SYNTAX condition will always indicate an ARexx error code.
Examples:

```
signal on error          /* enable interrupt     */
signal off syntax        /* disable SYNTAX       */
signal start             /* goto START           */
```

4-29 THEN

Usage: THEN *[;] [conditional statement]*
The THEN instruction must be the next statement following an IF or WHEN instruction, and is valid only in that context. It tests whether the preceding expression evaluated to 1 (**True**), in which case the conditional statement following the THEN is performed. If the expression result was a 0 (**False**), the conditional statement is skipped.

Example:

```
if i = j
   then say 'equal'
   else say 'not equal'
```

4-30 TRACE

Usage: TRACE *[symbol | string | [[VALUE] expression]]*
The TRACE instruction is used to set the internal tracing mode. If a symbol or string is supplied, it is taken as a literal. Otherwise, the tokens following the VALUE keyword are evaluated as an expression. The VALUE keyword can be omitted if the expression doesn't start with a symbol or string token.

In either case the result string is converted to uppercase and checked first for one of the "alphabetic" options. The valid alphabetic options are ALL, COMMANDS, ERRORS, INTERMEDIATES, LABELS, RESULTS, and SCAN. These can be spelled out in full or shortened to the initial character, and are described in Chapter 7. If the result doesn't match any of these options, the interpreter attempts to convert it to an integer. A conversion failure here will be reported as an error.

Prefix Characters. Two special symbol characters may precede any of the alphabetic keywords. The "?" character controls *interactive tracing*, and the "!" character controls *command inhibition*. These characters act as "toggles" to alternatively select and de-select the respective modes. Any number of prefix characters may precede an alphabetic option. Interactive tracing and command inhibition are described in Chapter 7.

Numeric Option. If the specified trace option is a negative whole number, it is accepted as a trace suppression count. The suppression count is the number of clauses (that would otherwise be traced) to be passed over before resuming the tracing display. Suppression counts are ignored except during interactive tracing.
Examples:

```
trace ?r              /* interactive RESULTS  */
trace off
trace -20             /* skip 20 clauses      */
```

4-31 UPPER

Usage: UPPER *variable [variable ...]*
The values of the variables in the list are converted to uppercase. It is not an error to include an uninitialized variable in the list, but it will be trapped if the NOVALUE interrupt has been enabled.

The TRANSLATE() or UPPER() Built-In functions could also be used to convert variables to uppercase, but the instruction form is more concise (and faster) if several variables are being converted.

Example:

```
when = 'Now is the time'
upper when
say when                    /* NOW IS THE TIME      */
```

4-32 WHEN

Usage: WHEN *expression [*THEN *[;] [conditional statement]]*
The WHEN instruction is similar to the IF instruction, but is valid only within a SELECT range. Each WHEN expression is evaluated in turn and must result in a boolean value. If the result is a 1, the conditional statement is executed and control passes to the END statement that terminates the SELECT. As in the case of the IF instruction, the THEN need not be part of the same clause.
Example:

```
select;
   when i<j then say 'less'
   when i=j then say 'equal'
   otherwise     say 'greater'
   end
```

Chapter 5

Commands

The REXX language is unusual in that an entire syntactic class of program statements are reserved for *commands*, statements that have meaning not within the language itself but rather to an external program. When a command clause is found in a program, it is evaluated as an expression and then sent through the *command interface* to an explicit or implicit *host application*, an external program that has announced its ability to receive commands. The host application then processes the command and returns a result code that indicates whether the command was performed successfully. In this manner every host program becomes fully programmable, and with even a limited set of predefined operations can be customized by the end user.

This chapter discusses the ARexx command interface and examines some of the ways in which commands can be used to build programs for an external program. Such programs are often called "macro programs" because they implement a complex ("macro") action from a series of simpler "micro" commands.

Chapter 10 has detailed information on the data structures required to implement a command interface for an applications program.

5-1 Command Clauses

Syntactically, a command clause is just an expression that can't be classified as another type of clause. The actual structure of the command is dictated by the external host to which it is intended, but in most cases will follow the model of a name or letter followed by parameter data. Command names can be given as either a symbol or a string. However, it is generally safer to use a string for the name, since it can't be assigned a value or be mistaken for an instruction keyword. For example, the following might be commands for a text editor:

```
JUMP current+10          /* advance to next      */
'insert' newstring       /* blast it in          */
'TOP'                    /* back to the top      */
```

Since command clauses are expressions, they are fully evaluated before being sent to the host. Any part of the final command string can be computed within the program, so virtually any sort of command structure can be created.

The interpretation of the received commands depends entirely on the host application. In the simplest case the command strings will correspond exactly to commands that could be entered directly by a user. For instance, positional control (up/down) commands for a text editor would probably have identical interpretations whether issued by the user or from a program. Other commands may be valid only when issued from a macro program; a command to simulate a menu operation would probably not be entered by the user.

5-2 The Host Address

The destination for a command is determined by the current *host address*, which is the name of the *public message port* managed by an external program. ARexx maintains two implicit host addresses, a "current" and a "previous" value, as part of the program's storage environment. These values can be changed at any time using the ADDRESS instruction (or its synonym, SHELL,) and the current host address can be inspected with the ADDRESS() Built-In function. The default host address string is "REXX", but this can be overridden when a program is invoked. In particular, most host applications will supply the name of their public port when they invoke a macro program, so that the macro can automatically issue commands back to the host.

One special host address is recognized: the string COMMAND indicates that the command should be issued directly to the underlying DOS. All other host addresses are assumed to refer to a public message port. An attempt to send a command to a non-existent message port will generate the syntax error "Host environment not found."

Single commands can be sent to a specific host without disturbing the host address settings. This is done using the ADDRESS instruction, as the following example illustrates:

```
ADDRESS MYEDIT 'jump top'
```

This example would send the command "`jump top`" to an external host named "MYEDIT."

It is important to note that you cannot send commands to a host application without knowing the name of its public message port. Writing macro programs to communicate with two or more hosts may require some clever programming to determine whether both hosts are active and what their respective host addresses are.

5-3 The Command Interface

ARexx implements its command interface using the message-passing facilities provided by the EXEC operating system. Each host application must provide a public message port, the name of which is referred to as the *host address.* ARexx programs issue commands by placing the command string in a message packet and sending the packet to the host's message port. The program "sleeps" while the host processes the command, and awakens when the message packet returns. The entire process can be regarded as a dialogue between the host application and a macro program: the host initiates the dialogue by invoking the macro, and the macro program replies with one or more command strings. The commands that can be sent are not limited to simple text strings, but might be address pointers or even bit-mapped images.

After it finishes processing a command, the host "replies" the message packet with a *return code* that indicates the status of the command. This return code is placed in the ARexx special variable RC so that it can be examined by the program. A value of zero is assumed to mean that no errors occurred, while positive values usually indicate progressively more severe error conditions. The return code allows the macro program to determine whether the command succeeded and to take action if it failed, so it is important for each applications program to document the meanings of the return codes for its commands.

5-4 Using Commands in Macro Programs

ARexx can be used to write programs for any host application that includes a suitable command interface. Some applications programs are designed with an embedded macro language, and may include many predefined macro commands. With a well-designed macro language interface the user will be usually be unaware of whether a given action is implemented as a primitive operation or as a macro program.

The starting point in designing a macro program is to examine the commands that would be required to perform it manually. The documentation for the host application program should then describe the possible return codes for each command; these codes can be used to determine whether the operation performed by the command was successful. Check also for "shortcut" commands that may be available only to macro programs; some applications programs may include very powerful functions that were implemented specifically for use in macro programs.

5-5 Using ARexx with Command Shells

Although ARexx was designed to work most effectively with programs that support its specific command interface, it can be used with any "command shell" program that uses standard I/O mechanisms to obtain its input stream. There are several ways to use ARexx to prepare a stream of commands for such program.

One obvious technique is to create an actual command file on the "RAM:" disk and then pass it directly to the command shell. For example, you could open a new CLI window to run a standard "execute" script using the following short program:

```
/* Launch a new CLI  */
address command
conwindow = "CON:0/0/640/100/NewOne"

/* create a command file on the fly */
call open out,"ram:$$temp",write
call writeln out,'echo "this is a test"'
call close out

/* open the new CLI window */
'newcli' conwindow "ram:$$temp"
exit
```

Since no disk accesses are required, this method is actually fairly fast, if not very elegant.

Another alternative is to use the command stacking facility provided by the PUSH and QUEUE instructions. These instructions allow an ARexx program to stack an arbitrary stream of commands and data for the command shell or other program to read. Any set of commands that could be "typed ahead" at a command prompt can be prepared in this fashion. After the ARexx program exits, the next program that uses the input stream will read the prepared commands and can process them in the normal fashion.

5-6 Command Inhibition

Sometimes it is necessary to write and test macro programs that issue potentially destructive commands. For instance, a program to find and delete unneeded files would be difficult to test safely, since it might accidentally delete the wrong files and would require a continual source of new files for testing.

To simplify the development and testing of such programs, ARexx provides a special tracing mode called *command inhibition* that suppresses host commands. While in command inhibition mode, command processing proceeds normally except that the command is not actually issued and the variable RC is set to 0. This allows the program logic to be verified before any commands are actually sent to the external program. Chapter 7 has further information on this facility.

Chapter 6

Functions

The basic concept of a function is a program or group of statements that will be executed whenever the function name appears in a certain context. Functions are an important building block of most computer languages in that they allow *modular programming* — the ability to build a large program from a series of smaller, more easily developed modules. In ARexx a function may be defined as part of (internal to) a program, as part of a library, or as a separate external program.

6-1 Syntax and Search Order

Function calls in an expression are defined syntactically as a symbol or string followed immediately by an open parenthesis. The symbol or string (taken as a literal) specifies the *function name*, and the open parenthesis begins the *argument list.* Between the opening and eventual closing parentheses are zero or more argument expressions, separated by commas, that supply the data being passed to the function. For example,

```
CENTER('title',20)
ADDRESS()
'AllocMem'(256*4,1)
```

are all valid function calls. Each argument expression is evaluated in turn and the resulting strings are passed as the argument list to the function. There is no limit to the number of arguments that may be passed to an internal function, but calls to Built-In or external functions are limited to a maximum of 15 arguments. Note that each argument expression, while often just a single literal value, can include arithmetic or string operations or even other function calls. Argument expressions are evaluated from left to right.

Functions can also be invoked using the CALL instruction. The syntax of this form is slightly different, and is described in Chapter 4. The CALL instruction can be used to invoke a function that may not return a value.

Search Order

Function linkages in ARexx are established dynamically at the time of the function call. A specific search order is followed until a function matching the name symbol or string is found. If the specified function cannot be located, an error is generated and the expression evaluation is terminated. The full search order is:

1. *Internal Functions.* The program source is examined for a label that matches the function name. If a match is found, a new storage environment is created and control is transferred to the label.
2. *Built-In Functions.* The Built-In function library is searched for the specified name. All of these functions are defined by uppercase names, and the library has been specially organized to make the search as efficient as possible.

3. *Function Libraries and Function Hosts.* The available function libraries and function hosts are maintained in a prioritized list, which is searched starting at the highest priority until the requested function is found or the end of the list is reached. Each function library is opened and called at a special entry point to determine whether it contains a function matching the given name. Function hosts are called using a message-passing protocol similar to that used for commands, and may be used as gateways for remote procedure calls to other machines in a network.

4. *External ARexx Programs.* The final search step is to check for an external ARexx program file by sending an invocation message to the ARexx resident process. The search always begins in the current directory, and follows the same search path as the original ARexx program invocation. The name matching process is not case-sensitive.

Note that the function name-matching procedure may be case-sensitive for some of the search steps but not for others. The matching procedure used in a function library or function host is left to the discretion of the applications designer. Functions defined with mixed-case names must be called using a string token, since symbol names are always translated to uppercase.

The full search order is followed whenever the function name is defined by a symbol token. However, the search for internal functions is bypassed if the name is specified by a string token. This allows internal functions to usurp the names of external functions, as in the following example:

```
CENTER:                          /* internal "CENTER" */
arg string,length                /* get arguments     */
length = min(length,60)          /* compute length    */
return 'CENTER'(string,length)
```

Here the Built-In function `CENTER()` has been replaced by an internal function of the same name, which calls the original function after modifying the length argument.

Internal Functions

The interpreter creates a new storage environment when an internal function is called, so that the previous (caller's) environment is preserved. The new environment inherits the values from its predecessor, but subsequent changes to the environment variables do not affect the previous environment. The specific values that are preserved are:

- The current and previous host addresses,
- The `NUMERIC DIGITS`, `FUZZ`, and `FORM` settings,
- The trace option, inhibit flag, and interactive flag,
- The state of the interrupt flags defined by the `SIGNAL` instruction, and
- The current prompt string as set by the `OPTIONS PROMPT` instruction.

The new environment does not automatically get a new symbol table, so initially all of the variables in the previous environment are available to the called function. The `PROCEDURE`

instruction can be used to create a new symbol table and thereby protect the caller's symbol values.

Execution of the internal function proceeds until a RETURN instruction is executed. At this point the new environment is dismantled and control resumes at the point of the function call. The expression supplied with the RETURN instruction is evaluated and passed back to the caller as the function result.

Built-In Functions

ARexx provides a substantial library of predefined functions as part of the language system. These functions are always available and have been optimized to work with the internal data structures. In general the Built-In functions execute much faster than an equivalent interpreted function, so their usage is strongly recommended.

The Built-In Function Library is not user-extensible, but additional functions will be included in later releases.

External Function Libraries

External function libraries provide a mechanism with which users and applications developers can extend the functionality of ARexx. A function library is a collection of one or more functions together with a "query" entry point that serves to match a name string with the appropriate function. External function libraries are supported as standard Amiga shared libraries, and may be either memory or disk-resident. Disk-resident libraries are loaded and opened as needed.

The ARexx resident process maintains a list, called the *Library List*, of the currently available function libraries and function hosts. Applications programs can add or remove function libraries as required. The Library List is maintained as a priority-sorted queue, and entries can be added at an appropriate priority to control the function name resolution. Libraries with higher priorities are searched first; within a given priority level, those libraries added first are searched first.

During the search process the ARexx interpreter opens each library and calls its "query" entry point. The query function must then check to see whether the requested function name is in the library. If not, it returns a "function not found" error code and the search continues with the next library in the list. Function libraries are always closed after being checked so that the operating system can reclaim the memory space if required. Once the requested function has been found, it is called with the arguments passed by the interpreter, and must return an error code and a result string.

The ARexx language system includes an external function library in a file called "`rexxsupport.library`." It contains a number of Amiga-specific functions and is described in Appendix D. Chapter 10 provides information on designing and implementing function libraries.

Function Hosts

Function hosts are called by sending a function invocation message packet to the public message port identified by the host's name. No constraints are imposed on the internal design of the host except that it must eventually return the invocation message with an appropriate return code and result string. The function call may result in a new program being loaded and run, or might even be sent to a network handler as a remote procedure call.

The available function hosts, along with the function libraries, are contained in the Library List maintained by the resident process. This list provides a general mechanism for resolving function names in a priority-controlled manner.

The ARexx resident process is an example of a function host. It is added to the Library List at a nominal priority of -60 when the resident process is started, using the same name ("REXX") that is used for command invocations. When it receives a function invocation packet, it searches for an external file matching the function name, just as it would for a command invocation of the same name. In particular, the search begins with the current directory and proceeds to the system REXX: directory. Two names are used in the search: the function name with the current file extension appended, and the name by itself. The name matching process is not case-sensitive, but is affected by the presence of explicit directory specifications or file extensions in the name string. The rules governing the search for external programs are covered in Chapter 9.

External programs are always run as a separate process in the Amiga's multitasking system. The calling program "sleeps" until the called function finishes and the message packet returns. The result string and error code are returned in the packet.

6-2 The Built-In Function Library

This section of the chapter is devoted to descriptions of the individual Built-In functions, which are listed alphabetically. Many of the functions have optional as well as required arguments. The optional arguments are shown in brackets, and generally have a default value that is used if the argument is omitted.

Maximum Arguments. While internal functions can be called with any number of arguments, the Built-In functions (and external functions as well) are limited to a maximum of 15 arguments.

Pad and Option Characters. For functions that accept a "pad" character argument, only the first character of the argument string is significant. If a null string is supplied, the default padding character (usually a blank) will be used. Similarly, where an option keyword is specified as an argument, only the first character is significant. Option keywords may be given in uppercase or lowercase.

I/O Support Functions. ARexx provides functions for creating and manipulating external DOS files. The functions available at the present time are OPEN(), CLOSE(), READCH(), READLN(), WRITECH(), WRITELN(), EOF(), SEEK(), and EXISTS(). Files are referenced by a "logical name," a case-sensitive name that is assigned to a file when it is first opened.

There is no limit to the number of files that may be open simultaneously, and all open files are closed automatically when the program exits.

Bit-Manipulation Functions. The functions BITCHG(), BITCLR(), BITCOMP(), BITSET(), and BITTST() are provided to implement extended bit-testing on character strings. These functions differ from similar string-manipulation functions in that the elementary unit of comparison is the bit rather than the byte. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.

ABBREV()

Usage: ABBREV(*string1,string2,[length]*)
Returns a boolean value that indicates whether *string2* is an abbreviation of *string1* with length greater than or equal to the specified *length* argument. The default length is 0, so the null string is an acceptable abbreviation.
Examples:

```
say abbrev('fullname','ful')     ==> 1
say abbrev('almost','alm',4)     ==> 0
say abbrev('any','')             ==> 1
```

ABS()

Usage: ABS(*number*)
Returns the absolute value of the *number* argument, which must be numeric.
Examples:

```
say abs(-5.35)                   ==> 5.35
say abs(10)                      ==> 10
```

ADDLIB()

Usage: ADDLIB(*name,priority,[offset,version]*)
Adds a function library or a function host to the Library List maintained by the resident process. The *name* argument specifies either the name of a function library or the public message port associated with a function host. The name is case-sensitive, and any libraries thus declared should reside in the system LIBS: directory. The *priority* argument specifies the search priority and must be an integer between 100 and -100, inclusive. The *offset* and *version* arguments apply only to libraries. The *offset* is the integer offset to the library's "query" entry point, and the *version* is an integer specifying the minimum acceptable release level of the library.

The function returns a boolean result that indicates whether the operation was successful. Note that if a library is specified, it is not actually opened at this time; similarly, no check is performed as to whether a specified function host port has been opened yet.
Example:

```
say addlib("rexxsupport.library",0,-30,0) ==> 1
call addlib "EtherNet",-20       /* a gateway */
```

ADDRESS()

Usage: ADDRESS()
Returns the current host address string. The host address is the message port to which commands will be sent. THe SHOW() function can be used to check whether the required external host is actually available.
See Also: SHOW()
Example:

```
say address()                    ==> REXX
```

ARG()

Usage: ARG(*[number],['Exists' | 'Omitted']*)
ARG() returns the number of arguments supplied to the current environment. If the *number* parameter alone is supplied, the corresponding argument string is returned. If a number and one of the keywords **Exists** or **Omitted** is given, the boolean return indicates the status of the corresponding argument. Note that the existence or omission test does not indicate whether the string has a null value, but only whether a string was supplied.
Examples:

```
/* Assume arguments were: ('one',,10)          */
say arg()                        ==> 3
say arg(1)                       ==> one
say arg(2,'O')                   ==> 1
```

B2C()

Usage: B2C(*string*)
Converts a string of binary digits (0, 1) into the corresponding (packed) character representation. The conversion is the same as though the argument string had been specified as a literal binary string (e.g. '1010'B). Blanks are permitted in the string, but only at byte boundaries. This function is particularly useful for creating strings that are to be used as bit masks.
See Also: X2C()
Examples:

```
say b2c('00110011')              ==> 3
say b2c('01100001')              ==> a
```

BITAND()

Usage: BITAND(*string1,string2,[pad]*)
The argument strings are logically ANDed together, with the length of the result being the longer of the two operand strings. If a pad character is supplied, the shorter string is padded on the right; otherwise, the operation terminates at the end of the shorter string and the remainder of the longer string is appended to the result.
Example:

```
bitand('0313'x,'FFF0'x)          ==> '0310'x
```

BITCHG()

Usage: BITCHG(*string,bit*)
Changes the state of the specified bit in the argument string. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.
Example:

```
bitchg('0313'x,4)              ==> '0303'x
```

BITCLR()

Usage: BITCLR(*string,bit*)
Clears (sets to zero) the specified bit in the argument string. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.
Example:

```
bitclr('0313'x,4)              ==> '0303'x
```

BITCOMP()

Usage: BITCOMP(*string1,string2,[pad]*)
Compares the argument strings bit-by-bit, starting at bit number 0. The returned value is the bit number of the first bit in which the strings differ, or -1 if the strings are identical.
Examples:

```
bitcomp('7F'x,'FF'x)           ==> 7
bitcomp('FF'x,'FF'x)           ==> -1
```

BITOR()

Usage: BITOR(*string1,string2,[pad]*)
The argument strings are logically ORed together, with the length of the result being the longer of the two operand strings. If a *pad* character is supplied, the shorter string is padded on the right; otherwise, the operation terminates at the end of the shorter string and the remainder of the longer string is appended to the result.
Example:

```
bitor('0313'x,'003F'x)         ==> '033F'x
```

BITSET()

Usage: BITSET(*string,bit*)
Sets the specified bit in the argument string is 1. Bit numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.
Example:

```
bitset('0313'x,2)              ==> '0317'x
```

BITTST()

Usage: BITTST(*string,bit*)
The boolean return indicates the state of the specified bit in the argument string. Bit

numbers are defined such that bit 0 is the low-order bit of the rightmost byte of the string.

Example:

```
bittst('0313'x,4)              ==> 1
```

BITXOR()

Usage: BITAND(*string1,string2,[pad]*)
The argument strings are logically exclusively-ORed together, with the length of the result being the longer of the two operand strings. If a *pad* character is supplied, the shorter string is padded on the right; otherwise, the operation terminates at the end of the shorter string and the remainder of the longer string is appended to the result.
Example:

```
bitxor('0313'x,'001F'x)        ==> '030C'x
```

C2B()

Usage: C2B(*string*)
Converts the character string into the equivalent string of binary digits.
See Also: C2X()
Example:

```
say c2b('abc')                 ==> 011000010110001001100011
```

C2D()

Usage: C2D(*string,[n]*)
Converts the *string* argument from its character representation to the corresponding decimal number, expressed as ASCII digits (0–9). If *n* is supplied, the character string is considered to be a number expressed in *n* bytes. The string is truncated or padded with nulls on the left as required, and the sign bit is extended for the conversion.
Examples:

```
say c2d('0020'x)               ==> 32
say c2d('FFFF')                ==> -1
say c2d('FF0100'x,2)           ==> 256
```

C2X()

Usage: C2X(*string*)
Converts the *string* argument from its character representation to the corresponding hexadecimal number, expressed as the ASCII characters 0–9 and A–F.
See Also: C2B()
Example:

```
say c2x('abc')                 ==> 616263
```

CENTER() or CENTRE()

Usage: CENTER(*string,length,[pad]*) *or* CENTRE(*string,length,[pad]*)
Centers the *string* argument in a string with the specified *length*. If the length is longer than that of the string, *pad* characters or blanks are added as necessary.
Examples:

```
say center('abc',6)              ==> ' abc  '
say center('abc',6,'+')          ==> '+abc++'
say center('123456',3)           ==> '234'
```

CLOSE()

Usage: CLOSE(*file*)
Closes the file specified by the given logical name. The returned value is a boolean success flag, and will be 1 unless the specified file was not open.
Example:

```
say close('input')               ==> 1
```

COMPRESS()

Usage: COMPRESS(*string,[list]*)
If the *list* argument is omitted, the function removes leading, trailing, or embedded blank characters from the *string* argument. If the optional *list* is supplied, it specifies the characters to be removed from the string.
Examples:

```
say compress('  why not ')       ==> whynot
say compress('++12-34-+','+-')   ==> 1234
```

COMPARE()

Usage: COMPARE(*string1,string2,[pad]*)
Compares two strings and returns the index of the first position in which they differ, or 0 if the strings are identical. The shorter string is padded as required using the supplied character or blanks.
Examples:

```
say compare('abcde','abcce')      ==> 4
say compare('abcde','abcde')      ==> 0
say compare('abc++','abc+-','+') ==> 5
```

COPIES()

Usage: COPIES(*string,number*)
Creates a new string by concatenating the specified number of copies of the original. The *number* argument may be zero, in which case the null string is returned.

Example:

```
say copies('abc',3)              ==> abcabcabc
```

D2C()

Usage: `D2C(`*number*`)`
Creates a string whose value is the binary (packed) representation of the given decimal number.
Example:

```
d2c(31)                          ==> '1F'x
```

DATATYPE()

Usage: `DATATYPE(`*string,[option]*`)`
If the *option* parameter is not specified, `DATATYPE()` tests whether the *string* parameter is a valid number and returns either `NUM` or `CHAR`. If an option keyword is given, the boolean return indicates whether the string satisfied the requested test. The following option keywords are recognized:

Table 6.1 `DATATYPE()` Options

Keyword	Characters Accepted
`Alphanumeric`	Alphabetics (A-Z,a-z) or Numerics (0-9)
`Binary`	Binary Digits String
`Lowercase`	Lowercase Alphabetics (a-z)
`Mixed`	Mixed Upper/Lowercase
`Numeric`	Valid Numbers
`Symbol`	Valid REXX Symbols
`Upper`	Uppercase Alphabetics (A-Z)
`Whole`	Integer Numbers
`X`	Hex Digits String

Examples:

```
say datatype('123')              ==> NUM
say datatype('1a f2','x')        ==> 1
say datatype('aBcde','L')        ==> 0
```

DELSTR()

Usage: `DELSTR(`*string,n,[length]*`)`
Deletes the substring of the *string* argument beginning with the *n*th character for the specified *length* in characters. The default length is the remaining length of the string.

Example:

```
say delstr('123456',2,3)         ==> 156
```

DELWORD()

Usage: DELWORD(*string,n,[length]*)
Deletes the substring of the *string* argument beginning with the *n*th word for the specified *length* in words. The default length is the remaining length of the string. The deleted string includes any trailing blanks following the last word.
Examples:

```
say delword('Tell me a story',2,2) ==> 'Tell story'
say delword('one two three',3)     ==> 'one two '
```

EOF()

Usage: EOF(*file*)
Checks the specified logical file name and returns the boolean value 1 (**True**) if the end-of-file has been reached, and 0 (**False**) otherwise.
Example:

```
say eof(infile)                    ==> 1
```

ERRORTEXT()

Usage: ERRORTEXT(*n*)
Returns the error message associated with the specified ARexx error code. The null string is returned if the number is not a valid error code.
Example:

```
say errortext(41)                  ==> Invalid expression
```

EXISTS()

Usage: EXISTS(*filename*)
Tests whether an external file of the given *filename* exists. The name string may include device and directory specifications.
Example:

```
say exists('df0:c/ed')             ==> 1
```

EXPORT()

Usage: EXPORT(*address,[string],[length],[pad]*)
Copies data from the (optional) string into a previously-allocated memory area, which must be specified as a 4-byte address. The *length* parameter specifies the maximum number of characters to be copied; the default is the length of the string. If the specified length is longer than the string, the remaining area is filled with the *pad* character or nulls ('00'x). The returned value is the number of characters copied.

Caution is advised in using this function. Any area of memory can be overwritten, possibly causing a system crash. Task switching is forbidden while the copy is being done, so system performance may be degraded if long strings are copied.
See Also: IMPORT(), STORAGE()

Example:

```
count = export('0004 0000'x,'The answer')
```

FREESPACE()

Usage: FREESPACE(*address,length*)
Returns a block of memory of the given length to the interpreter's internal pool. The *address* argument must be a 4-byte string obtained by a prior call to GETSPACE(), the internal allocator. It is not always necessary to release internally-allocated memory, since it will be released to the system when the program terminates. However, if a very large block has been allocated, returning it to the pool may avoid memory space problems. The return value is a boolean success flag.
See Also: GETSPACE()
Example:

```
say freespace('00042000'x,32)   ==> 1
```

GETCLIP()

Usage: GETCLIP(*name*)
Searches the Clip List for an entry matching the supplied *name* parameter, and returns the associated value string. The name-matching is case-sensitive, and the null string is returned if the name cannot be found. The usage and maintenance of Clip List entries is described in the Chapter 9.
See Also: SETCLIP()
Example:

```
/* Assume 'numbers' contains 'PI=3.14159'       */
say getclip('numbers')          ==> PI=3.14159
```

GETSPACE()

Usage: GETSPACE(*length*)
Allocates a block of memory of the specified length from the interpreter's internal pool. The returned value is the 4-byte address of the allocated block, which is not cleared or otherwise initialized. Internal memory is automatically returned to the system when the ARexx program terminates, so this function should not be used to allocate memory for use by external programs. The Support Library (described in Appendix D) includes the function ALLOCMEM() which to allocate memory from the system free list.
See Also: FREESPACE()
Example:

```
say c2x(getspace(32))           ==> '0003BF40'x
```

HASH()

Usage: HASH(*string*)
Returns the hash attribute of a string as a decimal number, and updates the internal hash value of the string.

Example:

```
say hash('1')                    ==> 49
```

IMPORT()

Usage: `IMPORT(`*address,[length]*`)`
Creates a string by copying data from the specified 4-byte address. If the length parameter is not supplied, the copy terminates when a null byte is found.
See Also: `EXPORT()`
Example:

```
extval = import('0004 0000'x,8)
```

INDEX()

Usage: `INDEX(`*string,pattern,[start]*`)`
Searches for the first occurrence of the *pattern* argument in the *string* argument, beginning at the specified *start* position. The default start position is 1. The returned value is the index of the matched pattern, or 0 if the pattern was not found.
Examples:

```
say index("123456","23")         ==> 2
say index("123456","77")         ==> 0
say index("123123","23",3)       ==> 5
```

INSERT()

Usage: `INSERT(`*new,old,[start],[length],[pad]*`)`
Inserts the *new* string into the *old* string after the specified *start* position. The default starting position is 0. The new string is truncated or padded to the specified *length* as required, using the supplied pad character or blanks. If the start position is beyond the end of the old string, the old string is padded on the right.
Examples:

```
say insert('ab','12345')         ==> ab12345
say insert('123','++',3,5,'-')   ==> ++-123--
```

LASTPOS()

Usage: `LASTPOS(`*pattern,string,[start]*`)`
Searches backwards for the first occurrence of the *pattern* argument in the *string* argument, beginning at the specified *start* position. The default starting position is the end of the string. The returned value is the index of the matched pattern, or 0 if the pattern was not found.

Examples:

```
say lastpos("123234","2")       ==> 4
say lastpos("123234","5")       ==> 0
say lastpos("123234","2",3)     ==> 2
```

LEFT()

Usage: LEFT(*string,length,[pad]*)
Returns the leftmost substring in the given *string* argument with the specified *length*. If the substring is shorter than the requested length, it is padded on the left with the supplied *pad* character or blanks.
Examples:

```
say left('123456',3)            ==> 123
say left('123456',8,'+')        ==> 123456++
```

LENGTH()

Usage: LENGTH(*string*)
Returns the length of the string.
Example:

```
say length('three')             ==> 5
```

MAX()

Usage: MAX(*number,number[,number,...]*)
Returns the maximum of the supplied arguments, all of which must be numeric. At least two parameters must be supplied.
Example:

```
say max(2.1,3,-1)               ==> 3
```

MIN()

Usage: MIN(*number,number[,number,...]*)
Returns the minimum of the supplied arguments, all of which must be numeric. At least two parameters must be supplied.
Example:

```
say min(2.1,3,-1)               ==> -1
```

OPEN()

Usage: OPEN(*file,filename,[*'Append' | 'Read' | 'Write'*]*)
Opens an external file for the specified operation. The *file* argument defines the logical name by which the file will be referenced. The *filename* is the external name of the file, and may include device and directory specifications. The function returns a boolean value that indicates whether the operation was successful. There is no limit to the number of files that

can be open simultaneously, and all open files are closed automatically when the program exits.
See Also: CLOSE(), READ(), WRITE()
Examples:

```
say open('MyCon','CON:160/50/320/100/MyCon/cds')  ==> 1
say open('outfile','ram:temp','W')                ==> 1
```

OVERLAY()

Usage: OVERLAY(*new,old,[start],[length],[pad]*)
Overlays the *new* string onto the *old* string beginning at the specified *start* position, which must be positive. The default starting position is 1. The new string is truncated or padded to the specified *length* as required, using the supplied *pad* character or blanks. If the start position is beyond the end of the old string, the old string is padded on the right.
Examples:

```
say overlay('bb','abcd')         ==> bbcd
say overlay('4','123',5,5,'-')   ==> 123--4----
```

POS()

Usage: POS(*pattern,string,[start]*)
Searches for the first occurrence of the *pattern* argument in the *string* argument, beginning at the position specified by the *start* argument. The default starting position is 1. The returned value is the index of the matched string, or 0 if the pattern wasn't found.
Examples:

```
say pos('23','123234')           ==> 2
say pos('77','123234')           ==> 0
say pos('23','123234',3)         ==> 4
```

PRAGMA()

Usage: PRAGMA(*option,[value]*)
This function allows a program to change various attributes relating to the system environment within which the program executes. The *option* argument is a keyword that specifies an environmental attribute; the currently implemented options are `Directory` and `Priority`. The *value* argument supplies the new attribute value to be installed. The value returned by the function depends on the attribute selected. Some attributes return the previous value installed, while others may simply set a boolean success flag. The currently defined option keywords are listed below.

- `Directory`. Specifies a new "current" directory. The current directory is used as the "root" for filenames that do not explicitly include a device specification. The return value is a boolean success flag.
- `Priority`. Specifies a new task priority. The priority value must be an integer in the range -128 to 127, but the practical range is much more limited. ARexx programs should never be run at a priority higher than that of the resident process, which currently runs at priority 4. The returned value is the previous priority level.

Examples:

```
say pragma('priority',-5)        ==> 0
call pragma 'Directory','df0:system'
```

RANDOM()

Usage: RANDOM(*[min],[max],[seed]*)
Returns a pseudorandom integer in the interval specified by the *min* and *max* arguments. The default minimum value is 0 and the default maximum value is 999. The interval **max-min** must be less than or equal to 1000. If a greater range of random integers is required, the values from the RANDU() function can be suitable scaled and translated.

The *seed* argument can be supplied to initialize the internal state of the random number generator.
See Also: RANDU()
Example:

```
thisroll = random(1,6)  /* might be 1   */
nextroll = random(1,6)  /* snake eyes?  */
```

RANDU()

Usage: RANDU(*[seed]*)
Returns a uniformly-distributed pseudorandom number between 0 and 1. The number of digits of precision in the result is always equal to the current Numeric Digits setting. With the choice of suitable scaling and translation values, RANDU() can be used to generate pseudorandom numbers on an arbitrary interval.

The optional *seed* argument is used to initialize the internal state of the random number generator.
See Also: RANDOM()
Example:

```
firsttry = randu()      /* 0.371902021? */
numeric digits 3
tryagain = randu()      /* 0.873?       */
```

READCH()

Usage: READCH(*file,length*)
Reads the specified number of characters from the given logical file into a string. The length of the returned string is the actual number of characters read, and may be less than the requested length if, for example, the end-of-file was reached.
See Also: READLN()
Example:

```
instring = readch('input',10)
```

READLN()

Usage: READLN(*file*)
Reads characters from the given logical file into a string until a "newline" character is found. The returned string does not include the "newline."
See Also: READCH()
Examples:

```
instring = readln('MyFile')
```

REMLIB()

Usage: REMLIB(*name*)
Removes an entry with the given *name* from the Library List maintained by the resident process. The boolean return is 1 if the entry was found and successfully removed. Note that this function does not make a distinction between function libraries and function hosts, but simply removes a named entry.
See Also: ADDLIB()
Example:

```
say remlib('MyLibrary.library') ==> 1
```

REVERSE()

Usage: REVERSE(*string*)
Reverses the sequence of characters in the string.
Example:

```
say reverse('?ton yhw')          ==> why not?
```

RIGHT()

Usage: RIGHT(*string,length,[pad]*)
Returns the rightmost substring in the given *string* argument with the specified *length.* If the substring is shorter than the requested length, it is padded on the left with the supplied *pad* character or blanks.
Examples:

```
say right('123456',4)            ==> 3456
say right('123456',8,'+')        ==> ++123456
```

SEEK()

Usage: SEEK(*file,offset,['Begin' | 'Current' | 'End']*)
Moves to a new position in the given logical file, specified as an offset from an anchor position. The default anchor is Current. The returned value is the new position relative to the start of the file.
Examples:

```
say seek('input',10,'B')         ==> 10
say seek('input',0,'E')          ==> 356  /* file length */
```

SETCLIP()

Usage: SETCLIP(*name,[value]*)
Adds a name-value pair to the Clip List maintained by the resident process. If an entry of the same name already exists, its value is updated to the supplied value string. Entries may be removed by specifying a null value. The function returns a boolean value that indicates whether the operation was successful.
Examples:

```
say setclip('path','df0:s')      ==> 1
say setclip('path')              ==> 1
```

SHOW()

Usage: SHOW(*option,[name],[pad]*)
Returns the names in the resource list specified by the *option* argument, or tests to see whether an entry with the specified *name* is available. The currently implemented options keywords are `Clip`, `Files`, `Libraries`, and `Ports`, which are described below.

- `Clip`. Examines the names in the Clip List.
- `Files`. Examines the names of the currently open logical file names.
- `Libraries`. Examines the names in the Library List, which are either function libraries or function hosts.
- `Ports`. Examines the names in the system Ports List.

If the *name* argument is omitted, the function returns a string with the resource names separated by a blank space or the *pad* character, if one was supplied. If the *name* argument is given, the returned boolean value indicates whether the name was found in the resource list. The name entries are case-sensitive.

SIGN()

Usage: SIGN(*number*)
Returns 1 if the *number* argument is positive or zero, and -1 if *number* is negative. The argument must be numeric.
Examples:

```
say sign(12)                     ==> 1
say sign(-33)                    ==> -1
```

SPACE()

Usage: SPACE(*string,n,[pad]*)
Reformats the *string* argument so that there are *n* spaces (blank characters) between each pair of words. If the *pad* character is specified, it is used instead of blanks as the separator character. Specifying *n* as 0 will remove all blanks from the string.
Examples:

```
say space('Now is the time',3)  ==> 'Now   is   the   time'
say space('Now is the time,0)   ==> 'Nowisthetime'
say space('1 2 3',1,'+')        ==> '1+2+3'
```

STORAGE()

Usage: STORAGE(*[address],[string],[length],[pad]*)
Calling STORAGE() with no arguments returns the available system memory. If the *address* argument is given, it must be a 4-byte string, and the function copies data from the (optional) *string* into the indicated memory area. The *length* parameter specifies the maximum number of bytes to be copied, and defaults to the length of the string. If the specified length is longer than the string, the remaining area is filled with the *pad* character or nulls ('00'x.)

The returned value is the previous contents of the memory area. This can be used in a subsequent call to restore the original contents.

Caution is advised in using this function. Any area of memory can be overwritten, possibly causing a system crash. Task switching is forbidden while the copy is being done, so system performance may be degraded if long strings are copied.
See Also: EXPORT()
Examples:

```
say storage()                  ==> 248400
oldval = storage('0004 0000'x,'The answer')
call storage '0004 0000'x,,32,'+'
```

STRIP()

Usage: STRIP(*string,[{'B' | 'L' | 'T'}],[pad]*)
If neither of the optional parameters is supplied, the function removes both leading and trailing blanks from the *string* argument. The second argument specifies whether Leading, Trailing, or Both (leading and trailing) characters are to be removed. The optional *pad* (or unpad, perhaps) argument selects the character to be removed.
Examples:

```
say strip('  say what?  ')      ==> 'say what?'
say strip('  say what?  ','L')  ==> 'say what?  '
say strip('++123+++','B','+')   ==> '123'
```

SUBSTR()

Usage: SUBSTR(*string,start,[length],[pad]*)
Returns the substring of the *string* argument beginning at the specified *start* position for the specified *length*. The starting position must be positive, and the default length is the remaining length of the string. If the substring is shorter than the requested length, it is padded on the left with the blanks or the specified *pad* character.
Examples:

```
say substr('123456',4,2)        ==> 45
say substr('myname',3,6,'=')    ==> name==
```

SUBWORD()

Usage: SUBWORD(*string,n,[length]*)
Returns the substring of the *string* argument beginning with the *n*th word for the specified *length* in words. The default length is the remaining length of the string. The returned string will never have leading or trailing blanks.
Example:

```
say subword('Now is the time  ',2,2)  ==> is the
```

SYMBOL()

Usage: SYMBOL(*name*)
Tests whether the *name* argument is a valid REXX symbol. If the name is not a valid symbol, the function returns the string BAD. Otherwise, the returned string is LIT if the symbol is uninitialized and VAR if it has been assigned a value.
Examples:

```
say symbol('J')              ==> VAR
say symbol('x')              ==> LIT
say symbol('++')             ==> BAD
```

TIME()

Usage: TIME(*option*)
Returns the current system time or controls the internal elapsed time counter. The valid option keywords are listed below.

Table 6.2 TIME() Options

Option Keyword	Description
Elapsed	Elapsed time in seconds
Hours	Current time in hours since midnight
Minutes	Current time in minutes since midnight
Reset	Reset the elapsed time clock
Seconds	Current time in seconds since midnight

If no option is specified, the function returns the current system time in the form HH:MM:SS.

Examples:

```
/* Suppose that the time is 1:02 AM ...        */
say time('Hours')            ==> 1
say time('m')                ==> 62
say time('S')                ==> 3720
call time 'R'                /* reset timer  */
say time('E')                ==> .020
```

TRACE()

Usage: TRACE(*option*)
Sets the tracing mode to that specified by the *option* keyword, which must be one of the valid alphabetic or prefix options. The tracing options are described in Chapter 7. The TRACE() function will alter the tracing mode even during interactive tracing, when TRACE instructions in the source program are ignored. The returned value is the mode in effect before the function call; this allows the previous trace mode to be restored later.
Example:

```
/* Assume tracing mode is ?ALL                    */
say trace('Results')            ==> ?A
```

TRANSLATE()

Usage: TRANSLATE(*string,[output],[input],[pad]*)
This function constructs a translation table and uses it to replace selected characters in the argument string. If only the *string* argument is given, it is translated to uppercase. If an *input* table is supplied, it modifies the translation table so that characters in the argument string that occur in the input table are replaced with the corresponding character in the *output* table. Characters beyond the end of the output table are replaced with the specified *pad* character or a blank.

Note that the result string is always of the same length as the original string. The input and output tables may be of any length.
Examples:

```
say translate("abcde","123","cbade","+")   ==> 321++
say translate("low")                       ==> LOW
say translate("0110","10","01")            ==> 1001
```

TRIM()

Usage: TRIM(*string*)
Removes trailing blanks from the *string* argument.
Example:

```
say length(trim(' abc  '))      ==> 4
```

UPPER()

Usage: UPPER(*string*)
Translates the *string* to uppercase. The action of this function is equivalent to that of TRANSLATE(*string*), but it is slightly faster for short strings.
Example:

```
say upper('One Fine Day')       ==> ONE FINE DAY
```

VALUE()

Usage: `VALUE(`*name*`)`
Returns the value of the symbol represented by the *name* argument.
Example:

```
/* Assume that J has the value 12              */
say value('j')                  ==> 12
```

VERIFY()

Usage: `VERIFY(`*string,list,*`['Match']`*`)`*
If the `Match` argument is omitted, the function returns the index of the first character in the *string* argument which is **not** contained in the *list* argument, or 0 if all of the characters are in the list. If the `Match` keyword is supplied, the function returns the index of the first character which **is** in the list, or 0 if none of the characters are.
Examples:

```
say verify('123456','0123456789')       ==> 0
say verify('123a56','0123456789')       ==> 4
say verify('123a45','abcdefghij','m')   ==> 4
```

WORD()

Usage: `WORD(`*string,n*`)`
Returns the *n*th word in the *string* argument, or the null string if there are fewer than *n* words.
Example:

```
say word('Now is the time  ',2)         ==> is
```

WORDINDEX()

Usage: `WORD(`*string,n*`)`
Returns the starting position of the *n*th word in the argument string, or 0 if there are fewer than *n* words.
Example:

```
say wordindex('Now is the time  ',3)    ==> 8
```

WORDLENGTH()

Usage: `WORDLENGTH(`*string,n*`)`
Returns the length of the *n*th word in the *string* argument.
Example:

```
say wordlength('one two three',3)       ==> 5
```

WORDS()

Usage: `WORDS(`*string*`)`
Returns the number of words in the *string* argument.
Example:

```
say words("You don't say!")              ==> 3
```

WRITECH()

Usage: `WRITECH(`*file,string*`)`
Writes the *string* argument to the given logical file. The returned value is the actual number of characters written.
Example:

```
say writech('output','Testing')          ==> 7
```

WRITELN()

Usage: `WRITELN(`*file,string*`)`
Writes the *string* argument to the given logical file with a "newline" appended. The returned value is the actual number of characters written.
Example:

```
say writeln('output','Testing')          ==> 8
```

X2C()

Usage: `X2C(`*string*`)`
Converts a string of hex digits into the (packed) character representation. Blank characters are permitted in the argument string at byte boundaries.
Examples:

```
say x2c('12ab')                  ==> '12ab'x
say x2c('12 ab')                 ==> '12ab'x
```

XRANGE()

Usage: `XRANGE(`*[start],[end]*`)`
Generates a string consisting of all characters numerically between the specified *start* and *end* values. The default start character is `'00'x`, and the default end character is `'FF'x`. Only the first character of the *start* and *end* arguments is significant.
Examples:

```
say xrange()                     ==> '00010203 ... FDFEFF'x
say xrange('a','f')              ==> 'abcdef'
say xrange(,'10'x)               ==> '000102030405060708090'x
```

Chapter 7

Tracing and Interrupts

ARexx provides tracing and source-level debugging facilities that are unusual in a high-level language. *Tracing* refers to the ability to display selected statements in a program as the program executes. When a clause is traced, its line number, source text, and related information are displayed on the console. The tracing action of the interpreter is determined by a *trace option* that selects which source clauses will be traced, and two modifier flags that control *command inhibition* and *interactive tracing.*

The internal *interrupt system* enables an ARexx program to detect certain synchronous or asynchronous events and to take special actions when they occur. Events such as a syntax error or an external halt request that would normally cause the program to exit can instead be trapped so that corrective actions can be taken.

7-1 Tracing Options

Trace options are sometimes called an alphabetic options, since the keywords that select an option can be shortened to one letter for convenience. The alphabetic options are:

- `ALL`. All clauses are traced.
- `COMMANDS`. All command clauses are traced before being sent to the external host. Non-zero return codes are displayed on the console.
- `ERRORS`. Commands that generate a non-zero return code are traced after the clause is executed.
- `INTERMEDIATES`. All clauses are traced, and intermediate results are displayed during expression evaluation. These include the values retrieved for variables, expanded compound names, and the results of function calls.
- `LABELS`. All label clauses are traced as they are executed. A label will be displayed each time a transfer of control takes place.
- `NORMAL`. Command clauses with return codes that exceed the current error failure level are traced after execution, and an error message is displayed. This is the default trace option.
- `RESULTS`. All clauses are traced before execution, and the final result of each expression is displayed. Values assigned to variables by `ARG`, `PARSE`, or `PULL` instructions are also displayed.
- `SCAN`. This is a special option that traces all clauses and checks for errors, but suppresses the actual execution of the statements. It is helpful as a preliminary screening step for a newly-created program.

The tracing mode can be set using either the `TRACE` instruction or the `TRACE()` Built-In function. The `RESULTS` trace option is recommended for general-purpose testing. Tracing can be selectively disabled from within a program so that previously-tested parts of a program can be skipped.

7-2 Display Formatting

Each trace line displayed on the console is indented to show the effective control (nesting) level at that clause, and is identified by a special three-character code, as shown in Table 7.1 below. The source for each clause is preceded by its line number in the program. Expression results or intermediates are enclosed in double quotes so that leading and trailing blanks will be apparent.

Table 7.1 Tracing Prefix Codes

Code	Displayed Values
+++	Command or syntax error
>C>	Expanded compound name
>F>	Result of a function call
>L>	Label clause
>O>	Result of a dyadic operation
>P>	Result of a prefix operation
>U>	Uninitialized variable
>V>	Value of a variable
>>>	Expression or template result
>.>	"Placeholder" token value

Tracing Output

The tracing output from a program is always directed to one of two logical streams. The interpreter first checks for a stream named STDERR, and directs the output there if the stream exists. Otherwise the trace output goes to the standard output stream STDOUT and will be interleaved with the normal console output of the program. The STDERR and STDOUT streams can be opened and closed under program control, so the programmer has complete control over the destination of tracing output.

In some cases a program may not have a predefined output stream. For example, a program invoked from a host application that did not provide input and output streams would not have an output console. To provide a tracing facility for such programs, the resident process can open a special *global tracing console* for use by any active program. When this console opens, the interpreter automatically opens a stream named STDERR for each ARexx program in which STDERR is not currently defined, and the program then diverts its tracing output to the new stream.

The global console can be opened and closed using the command utilities tco and tcc, respectively. The console may not close immediately upon request, however. The resident process waits until all active programs have diverted their tracing streams back to the default state before actually closing the console. Applications programs may provide direct control over the tracing console by sending request packets to the resident process, which is discussed in Chapter 10.

The trace stream (STDERR or STDOUT) is also used for trace input, so a program in interactive tracing mode will wait for user input from this console. The global tracing console is always shared among all currently active programs. Since it may be confusing to

have several programs simultaneously writing writing to the same console, it is recommended that only one program at a time be traced using the global console.

Command Inhibition

ARexx provides a tracing mode called *command inhibition* that suppresses host commands. In this mode command clauses are evaluated in the normal manner, but the command is not actually sent to the external host, and the return code is set to zero. This provides a way to test programs that issue potentially destructive commands, such as erasing files or formatting disks. Command inhibition does not apply to command clauses that are entered interactively. These commands are always performed, but the value of the special variable RC is left unchanged.

Command inhibition may be used in conjunction with any trace option. It is controlled by the "!" character, which may appear by itself or may precede any of the alphabetic options in a TRACE instruction. Each occurrence of the "!" character "toggles" the inhibition mode currently in effect. Command inhibition is cleared when tracing is set to OFF.

7-3 Interactive Tracing

Interactive tracing is a debugging facility that allows the user to enter source statements while a program is executing. These statements may be used to examine or modify variable values, issue commands, or otherwise interact with the program. Any valid language statements can be entered interactively, with the same rules and restrictions that apply to the INTERPRET instruction. In particular, compound statements such as DO and SELECT must be complete within the entered line.

Interactive tracing can be used with any of the trace options. While in interactive tracing mode, the interpreter pauses after each traced clause and prompts for input with the code "+++." At each pause, three types of user responses are possible:

- If a null line is entered, the program continues to the next pause point.
- If an "=" character is entered, the preceding clause is executed again.
- Any other input is treated as a debugging statement, and is scanned and executed.

The pause points during interactive tracing are determined by the tracing option currently in effect, as the interpreter pauses only after a traced clause. However, certain instructions cannot be safely (or sensibly) re-executed, so the interpreter will not pause after executing one of these. The "no-pause" instructions are CALL, DO, ELSE, IF, THEN, and OTHERWISE. The interpreter will also not pause after any clause that generated an execution error.

Interactive tracing mode is controlled by the "?" character, either by itself or in combination with an alphabetic trace option. Any number of "?" characters may precede an option, and each occurrence toggles the mode currently in effect. For example, if the current trace option was NORMAL, then "TRACE ?R" would set the option to RESULTS and select interactive tracing mode. A subsequent "TRACE ?" would turn off interactive tracing.

Error Processing

The ARexx interpreter provides special error processing while it executes debugging statements. Errors that occur during interactive debugging are reported, but do not cause the program to terminate. This special processing applies only to the statements that were entered interactively. Errors occurring in the program source statements are treated in the usual way whether or not the interpreter is in interactive tracing mode.

In addition to the special error processing, the interpreter also disables the internal interrupt flags during interactive debugging. This is necessary to prevent an accidental transfer of control due to an error or uninitialized variable. However, if a "`SIGNAL label`" instruction is entered, the transfer will take place, and any remaining interactive input will be abandoned. The `SIGNAL` instruction can still be used to alter the interrupt flags, and the new settings will take effect when the interpreter returns to normal processing.

The External Tracing Flag

The ARexx resident process maintains an *external tracing flag* that can be used to force programs into interactive tracing mode. The tracing flag can be set using the `ts` command utility. When the flag is set, any program not already in interactive tracing mode will enter it immediately. The internal trace option is set to `RESULTS` unless it is currently set to `INTERMEDIATES` or `SCAN`, in which case it remains unchanged. Programs invoked while the external tracing flag is set will begin executing in interactive tracing mode.

The external tracing flag provides a way to regain control over programs that are caught in loops or are otherwise unresponsive. Once a program enters interactive tracing mode, the user can step through the program statements and diagnose the problem. There is one caveat, though: external tracing is a global flag, so all currently-active programs are affected by it. The tracing flag remains set until it is cleared using the "`te`" command utility. Each program maintains an internal copy of the last state of the tracing flag, and sets its tracing option to `OFF` when it observes that the tracing flag has been cleared.

7-4 Interrupts

ARexx maintains an internal interrupt system that can be used to detect and trap certain error conditions. When an interrupt is enabled and its corresponding condition arises, a transfer of control to the label specific to that interrupt occurs. This allows a program to retain control in circumstances that might otherwise cause the program to terminate. The interrupt conditions can caused by either *synchronous* events like a syntax error, or *asynchronous* events like a "control-C" break request. Note that these internal interrupts are completely separate from the hardware interrupt system managed by the EXEC operating system.

The interrupts supported by ARexx are described below. The name assigned to each is actually the label to which control will be tranferred. Thus, a `SYNTAX` interrupt will transfer control to the label "`SYNTAX:`." Interrupts can be enabled or disabled using the `SIGNAL` instruction. For example, the instruction "`SIGNAL ON SYNTAX`" would enable the `SYNTAX` interrupt.

- BREAK_C. This interrupt will trap a control-C break request generated by DOS. If the interrupt is not enabled, the program terminates immediately with the error message "Execution halted" and returns with the error code set to 2.
- BREAK_D. The interrupt will detect and trap a control-D break request issued by DOS. The break request is ignored if the interrupt is not enabled.
- BREAK_E. The interrupt will detect and trap a control-E break request issued by DOS. The break request is ignored if the interrupt is not enabled.
- BREAK_F. The interrupt will detect and trap a control-F break request issued by DOS. The break request is ignored if the interrupt is not enabled.
- ERROR. This interrupt is generated by any host command that returns a non-zero code.
- HALT. An external halt request will be trapped if this interrupt is enabled. Otherwise, the program terminates immediately with the error message "Execution halted" and returns with the error code set to 2.
- IOERR. Errors detected by the I/O system will be trapped if this interrupt is enabled.
- NOVALUE. An interrupt will occur if an uninitialized variable is used while this condition is enabled. The usage could be within an expression, in the UPPER instruction, or with the VALUE() built-in function.
- SYNTAX. A syntax or execution error will generate this interrupt. Not all errors such errors can be trapped, however. In particular, certain errors occurring before a program is actually executing, and those detected by the ARexx external interface, cannot be trapped by the SYNTAX interrupt.

When an interrupt forces a transfer of control, all of the currently active control ranges are dismantled, and the interrupt that caused the transfer is disabled. This latter action is necessary to prevent a possible recursive interrupt loop. Only the control structures in the current environment are affected, so an interrupt generated within a function will not affect the caller's environment.

Special Variables. Two special variables are affected when an interrupt occurs. The variable SIGL is always set to the current line number before the transfer of control takes place, so that the program can determine which source line was being executed. When an ERROR or SYNTAX interrupt occurs, the variable RC is set to the error code that caused the condition. For ERROR interrupts this value will be a command return code, and can usually be interpreted as an error severity level. The value for SYNTAX interrupts is always an ARexx error code.

Interrupts are useful primarily to allow a program to take special error-recovery actions. Such actions might involve informing external programs that an error occurred, or simply reporting further diagnostics to help in isolating the problem. In the following example, the program issues a "message" command to an external host called "MyEdit" whenever a syntax error is detected:

```
/* A macro program for 'MyEdit'                */
signal on syntax        /* enable interrupt    */
  .
  . (normal processing)
  .
exit
syntax:                 /* syntax error detected*/
   address 'MyEdit'
   'message' 'error' rc errortext(rc)
   exit 10
```

Chapter 8

Parsing and Templates

Parsing is a operation that extracts substrings from a string and assigns them to variables. It corresponds roughly to the notion of a "formatted read" used in other languages, but has been generalized in the several ways. Parsing is performed using the PARSE instruction or its variants ARG and PULL. The input for the operation is called the *parse string* and can come from several sources; these source options are described with the PARSE instruction in Chapter 4.

Parsing is controlled by a *template*, a group of tokens that specifies both the variables to be given values and the way to determine the value strings. Templates were described briefly with the PARSE instruction; the present chapter presents a more formal description of their structure and operation.

String-manipulation functions like SUBSTR() and INDEX() could also be used for parsing, but it is more efficient to use the instruction statements. This is especially true if many fields are to be extracted from a string.

8-1 Template Structure

The tokens that are valid in a template are symbols, strings, operators, parentheses, and commas. Any blanks that may be present as separators are removed before the template is processed. The tokens in a template ultimately serve to specify one of the two basic template objects:

- *Markers* determine a scan position within the parse string, and
- *Targets* are symbols to be assigned a value.

With these objects in mind, the parsing process can be described as one of associating with each target a starting and ending position in the parse string. The substring between these positions then becomes the value for the target.

Markers. There are three types of marker objects:

- *Absolute* markers specify an actual index position in the parse string,
- *Relative* markers specify a positive or negative offset from the current position, and
- *Pattern* markers specify a position implicitly, by matching the pattern against the parse string beginning at the current scan position.

Targets. Targets are usually specified by variable symbols. The *placeholder* is a special type of target, and is denoted by a period (.) symbol. A placeholder behaves like a normal target except that a value is not actually assigned to it.

Targets, like markers, can affect the scan position if value strings are being extracted by *tokenization*. Parsing by tokenization extracts words (tokens) from the parse string, and

is used whenever a target is followed immediately by another target. During tokenization the current scan position is advanced past any blanks to the start of the next word. The ending index is the position just past the end of the word, so that the value string has neither leading nor trailing blanks.

Template Objects

Each template object is specified by one or more tokens, which have the following interpretations.

Symbols. A symbol token may specify either a target or a marker object. If it follows an operator token (+, -, or =), it represents a marker, and the symbol value is used as an absolute or relative position. Symbols enclosed in parentheses specify pattern markers, and the the symbol value is used as the pattern string.

If neither of the preceding cases applies and the symbol is a variable, then it specifies a target. Fixed symbols always specify absolute markers and must be whole numbers, except for the period (.) symbol which defines a placeholder target.

Strings. A string token always represents a pattern marker.

Parentheses. A symbol enclosed in parentheses is a pattern marker, and the value of the symbol is used as the pattern string. While the symbol may be either fixed or variable, it will usually be a variable, since a fixed pattern could be given more simply as a string.

Operators. The three operators "+," "-," and "=" are valid within a template, and must be followed by a fixed or variable symbol. The value of the symbol is used as a marker and must therefore represent a whole number. The "+" and "-" operators signify a relative marker, whose value is negated by the "-" operator. The "=" operator indicates an absolute marker, and is optional if the marker is defined by a fixed symbol.

Commas. The comma (,) marks the end of a template, and is used as a separator when multiple templates are provided with an instruction. The interpreter obtains a new parse string before processing each succeeding template. For some source options, the new string will be identical to the previous one. The `ARG`, `EXTERNAL`, and `PULL` options will generally supply a different string, as will the `VAR` option if the variable has been modified.

The Scanning Process

Scan positions are expressed as an index in the parse string, and can range from 1 (the start of the string) to the length of the string plus 1 (the end). An attempt to set the scan position before the start or after the end of the string instead sets it to the beginning or end, respectively.

The substring specified by two scan indices includes the characters from the starting position up to, but not including, the ending position. For example, the indices 1 and 10 specify characters 1-9 in the parse string. One additional rule is applied if the second scan index is less than or equal to the first: in this case the *remainder* of the parse string is used as the substring. This means that a template specification like

```
parse arg 1 all 1 first second
```

will assign the entire parse string to the variable ALL. Of course, if the current scan index is already at the end of the parse string, then the remainder is just the null string.

When a pattern marker is matched against the parse string, the marker position is the index of the first character of the matched pattern, or the end of the string if no match was found. The pattern is removed from the string whenever a match is found. This is the only operation that modifies the parse string during the parsing process.

Templates are scanned from left to right with the initial scan index set to 1, the start of the parse string. The scan position is updated each time a marker object is encountered, according to the type and value of the marker. Whenever a target object is found, the value to be assigned is determined by examining the next template object. If the next object is another target, the value string is determined by tokenizing the parse string. Otherwise, the current scan position is used as the start of the value string, and the position specified by the following marker is used as the end point.

The scan continues until all of the objects in the template have been used. Note that every target will be assigned a value; once the parse string has been exhausted, the null string is assigned to any remaining targets.

8-2 Templates in Action

The preceding section is rather abstract, so let's look now at some examples of parsing with templates.

Parsing by Tokenization

Computer programs frequently require splitting a string into its component words or tokens. This is easily accomplished with a template consisting entirely of variables (targets).

```
/* Assume "hammer 1 each $600.00" was entered   */
pull item qty units cost .
```

In this example the input line from the PULL instruction is split into words and assigned to the variables in the template. The variable item receives the value "hammer," qty is set to "1," units is set to "each," and cost gets the value "$600.00." The final placeholder (.) is given a null value, since there are only four words in the input. However, it forces the preceding variable cost to be given a tokenized value. If the placeholder were omitted, the remainder of the parse string would be assigned to cost, which would then have a leading blank.

In the next example, the first word of a string is removed and the remainder is placed back in the string. The process continues until no more words are extracted.

```
/* Assume "result" contains a string of words   */
do forever
   /* Get first word of string                  */
   parse var result first result
   if first == '' then leave
   /* ... process words ...                     */
   end
```

Pattern Parsing

The next example uses pattern markers to extract the desired fields. The "pattern" in this case is very simple — just a single character — but in general can be an arbitrary string of any length. This form of parsing is useful whenever delimiter characters are present in the parse string.

```
/* Assume an argument string "12,35.5,1"        */
arg hours ',' rate ',' withhold
```

Keep in mind that the pattern is actually *removed* from the parse string when a match is found. If the parse string is scanned again from the beginning, the length and structure of the string may be different than at the start of the parsing process. However, the original source of the string is never modified.

Positional Markers

Parsing with positional markers is used whenever the fields of interest are known to be in certain positions in a string. In the next example, the records being processed contain a variable length field. The starting position and length of the field are given in the first part of the record, and a variable positional marker is used to extract the desired field.

```
/* records look like:                           */
/* start:   1-5                                 */
/* length:  6-10                                */
/* name:    @(start,length)                     */
parse value record with 1 start +5 length +5 =start name +length
```

The "`=start`" sequence in the above example is an absolute marker whose value is the position placed in the variable `start` earlier in the scan. The "`+length`" sequence supplies the effective length of the field.

Multiple Templates

It is sometimes useful to specify more than one template with an instruction, which can be done by separating the templates with a comma. In this next example, the `ARG` instruction (or `PARSE UPPER ARG`) is used to access the argument strings provided when the program was called. Each template accesses the succeeding argument string.

```
/* Assume arguments were ('one two',12,sort)    */
arg first second,amount,action,option
```

The first template consists of the variables **first** and **second**, which are set to the values "one" and "two," respectively. In the next two templates **amount** gets the value "12" and **action** is set to "SORT." The last template consists of the variable "option," which is set to the null string, since only three arguments were available.

When multiple templates are used with the EXTERNAL or PULL source options, each additional template requests an additional line of input from the user. In the next example two lines of input are read:

```
/* read last, first, and middle names and ssn   */
pull last ',' first middle,ssn
```

The first input line is expected to have three words, the first of which is followed by a comma, which are assigned to the variables **last**, **first**, and **middle**. The entire second input line is assigned to the variable **ssn**.

Multiple templates can be useful even with a source option that returns the identical parse string. If the first template included pattern markers that altered the parse string, the subsequent templates could still access the original string. Note that subsequent parse strings obtained from the VALUE source do not cause the expression to be reevaluated, but only retrieve the prior result.

Chapter 9

The Resident Process

This chapter describes some of the capabilities of the ARexx *resident process*, a global communications and resources manager. The material presented here is directed to the general user; Chapter 10 covers these topics in greater depth for software developers who wish to integrate ARexx with other applications programs.

The resident process must be active before any ARexx programs can be run. It announces its presence to the system by opening a public message port named "REXX," so applications programs that use ARexx should check for the presence of this port. If the port is not open, the user can either be informed that the macro processor is not available, or else the applications program can start up the resident process. The latter option can be done using the **rexxmast** command.

The primary function of the resident process is to launch ARexx programs. When an applications program sends a "command" or "function" message to the "REXX" port, the resident process creates a new DOS process to execute the program, and forwards the invocation message to newly created process. It also creates a new instance of the ARexx global data structure, which links together all of the structures manipulated by the program.

In addition to launching programs, the resident process manages various resources used by ARexx. These resources include a list of available function libraries called the Library List, a list of (name,value) pairs called the Clip List, and a list of the currently active ARexx programs. Built-In functions are available to manipulate the Library List and Clip List from within an ARexx program. Applications programs can modify a resource list either by sending a request packet to the resident process or by direct manipulation of the list.

9-1 Command Utilities

ARexx is supplied with a number of command utilities to provide various control functions. These are executable modules that can be run from the CLI, and should reside in the system command (`C:`) directory for convenience. These commands are relevant only when the ARexx resident process is active.

The functions performed by these utilities may also be available from within an applications program. All of the utilities are implemented by sending message packets to the resident process, so an application designed to work closely with ARexx could easily provide these functions as part of its control menu.

HI

Usage: `HI`
Sets the global halt flag, which causes all active programs to receive an external halt request. Each program will exit immediately unless its `HALT` interrupt has been enabled. The halt flag does not remain set, but is cleared automatically after all current programs have received the request.

RX

Usage: `RX` *name [arguments]*
This command launches an ARexx program. If the specified *name* includes an explicit path, only that directory is searched for the program; otherwise, the current directory and the system `REXX:` device are checked for a program with the given name. The optional argument string is passed to the program.

RXSET

Usage: `RXSET` *name value*
Adds a (name,value) pair to the Clip List. Name strings are assumed to be in mixed case. If a pair with the same name already exists, its value is replaced with the current string. If a name without a value string is given, the entry is removed from the Clip List.

RXC

Usage: `RXC`
Closes the resident process. The "REXX" public port is withdrawn immediately, and the resident process exits as soon as the last ARexx program finishes. No new programs can be launched after a "close" request.

TCC

Usage: `TCC`
Closes the global tracing console as soon as all active programs are no longer using it. All read requests queued to the console must be satisfied before it can be closed.

TCO

Usage: `TCO`
Opens the global tracing console. The tracing output from all active programs is diverted automatically to the new console. The console window can be moved and resized by the user, and can be closed with the "`TCC`" command.

TE

Usage: `TE`
Clears the global tracing flag, which forces the tracing mode to `OFF` for all active ARexx programs.

TS

Usage: `TS`
Starts interactive tracing by setting the external trace flag, which forces all active ARexx programs into interactive tracing mode. Programs will start producing trace output and will pause after the next statement. This command is useful for regaining control over programs caught in infinite loops or otherwise misbehaving. The trace flag remains set until cleared by the `TE` command, so subsequent program invocations will begin executing in interactive tracing mode.

9-2 Resource Management

Individual ARexx programs manage their internal memory allocation and I/O file resources, but some resources need to be available to all programs. The management of these global resources is one of the major functions of the resident process. Global resources are maintained as doubly-linked lists, in keeping with the general design principles of the EXEC operating system. Linked lists provide a flexible and open mechanism for resource management, and help avoid the built-in limits common with other approaches.

The Global Tracing Console

The tracing output from an ARexx program usually goes to the standard output stream `STDOUT`, and is therefore interleaved with the normal output of the program. Since this may be confusing at times, a global trace console can be opened to display only tracing output. The console can be opened using the `tco` command utility or by sending an `RXTCOPN` request packet to the resident process. ARexx programs will automatically divert their tracing output to the new window, which is opened as a standard AmigaDOS console. The user can move it and resize it as required.

The tracing console also serves as the input stream for programs during interactive tracing. When a program pauses for tracing input, the input line must be entered at the trace console. Any number of programs may use the tracing console simultaneously, although it is generally recommended that only one program at a time be traced.

The tracing console can be closed using the `tcc` command or by sending an `RXTCCLS` request packet to the resident process. The closing is delayed until all read requests to the console have been satisfied. Only when all of the active programs indicate that they are no longer using the console will it actually be closed.

The Library List

The resident process maintains a Library List of the *function libraries* and *function hosts* currently available to ARexx programs. This list is used to resolve all references to external functions. Each entry has an associated search priority in the range 100 to -100, with the higher-valued entries being searched first until the requested function is found. The list is searched by calling each entry, using the appropriate protocol, until the return code indicates that the function was found.

The two types of entities maintained by the list are quite different in some respects, but the ultimate way in which a function call is resolved is transparent to the calling program. A function library is a collection of functions organized as an Amiga shared library, while a function host is a separate task that manages a message port. Function libraries are called as part of the ARexx interpreter's task context, but calls to function hosts are mediated by passing a message packet. The ARexx resident process is itself a function host, and is installed in the Library List at a priority of -60.

The resident process provides addition and deletion operations for maintaining the Library List; these operations are performed by sending an appropriate message packet. The Library List is always maintained in priority order. Within a given priority level any new entries are added to the end of the chain, so that entries added first will be searched

first. The priority levels are significant if any of the libraries have duplicate function name definitions, since the function located further down the search chain could never be called.

Function Libraries. Each function library entry in the Library List contains a library name, a search priority, an entry point offset, and a version number. The library name must refer to a standard Amiga shared library residing in the system `LIBS:` directory so that it can be loaded when needed. Function libraries can be created and maintained by users or applications developers; Chapter 10 has information on their design and implementation.

The "query" function is the library entry point that is actually called by the interpreter. It must be specified as an integer offset (e.g. "-30") from the library base. The return code from the query call then indicates whether the desired function was found; if it was, the function is called with the parameters passed by the interpreter and the function result is returned to the caller. Otherwise, the search continues with the next entry in the list. In either event the library is closed to await the next call.

A note of caution: not every Amiga shared library can be used as a function library. Function libraries must have a special entry point to perform the dynamic linking required to access the functions from within ARexx. Each library should include documentation providing its version number and the integer offset to its "query" entry point.

Function Hosts. The name associated with a function host is the name of its public message port. Function calls are passed to the host as a message packet; it is then up to the individual host to determine whether the specified function name is one that it recognizes. The name resolution is completely internal to the host, so function hosts provide a natural gateway mechanism for implementing remote procedure calls to other machines in a network.

The Clip List

The Clip List maintains a set of (name,value) pairs that may be used for a variety of purposes. Each entry in the list consists of a name and a value string, and may be located by name. Since the Clip List is publicly accessible, it may be used as a general clipboard-like mechanism for intertask communication. In general, the names used should be chosen to be unique to an application to prevent collisions with other programs. Any number of entries may be posted to the list.

One potential application for the Clip List is as a mechanism for loading predefined constants into an ARexx program. The language definition does not include a facility comparable to the "header file" preprocessor in the "C" language. However, consider a string in the Clip List of the form

```
pi=3.14159; e=2.718; sqrt2=1.414 ...
```

i.e., a series of assignments separated by semicolons. In use, such a string could be retrieved by name using the Built-In function `GETCLIP()` and then `INTERPRET`ed within the program. The assignment statements within the string would then create the required constant definitions. The following program fragment illustrates the process:

```
/* assume a string called "numbers" is available*/
numbers = getclip('numbers')  /* case-sensitive */
interpret numbers             /* ... assignments*/
...
```

More generally, the strings would not be restricted to contain only assignment statements, but could include any valid ARexx statements. The Clip List could thus provide a series of programs for initializations or other processing tasks.

The resident process supports addition and deletion operations for maintaining the Clip List. The names in the (name,value) pairs are assumed to be in mixed case, and are maintained to be unique in the list. An attempt to add a string with an existing name will simply update the value string. The name and value strings are copied when an entry is posted to the list, so the program that adds an entry is not required to maintain the strings.

Entries posted to the Clip List remain available until explicitly removed. The Clip List is automatically released when the resident process exits.

Chapter 10

Interfacing to ARexx

This chapter discusses the issues involved in designing and implementing an interface between ARexx and an external applications program. The material presented here is directed to software developers, so a high degree of familiarity with programming the Amiga in either "C" or assembly-language is assumed.

ARexx can interact with external programs in several ways. The *command interface* is used to communicate with an external program running as a separate task in the Amiga's multitasking environment. The interaction takes place by passing messages between public message ports, and is in many ways similar to the interaction of a program with Intuition, the Amiga's window and menu manager. The command interface provides both a means of sharing data and a method of controlling an applications program.

Function libraries provide a mechanism for calling external code as part of an ARexx program's task context. The linkages for such calls are established dynamically at run time rather than when the program is linked, so each function library must include an entry point to match function names with the address of the function to be called.

Function hosts are external tasks that manage a public message port for communicating with ARexx or other programs. Both function hosts and function libraries are managed by the Library List, which provides a prioritized search mechanism for resolving function names. Function hosts may be used as a gateway into a network to provide a remote procedure call facility. ARexx imposes no constraints on the internal operations of a function host, except to require that message packets be returned with an appropriate code.

The *resident process* acts as the hub for communications between ARexx and external entities. It opens and manages a public message port named "REXX," and provides a number of support services. Note that the resident process is itself a "host application" whose function it is to launch ARexx programs and maintain global resources. The activation structures for all ARexx programs are linked into a list maintained by the resident process, and in principle their complete internal states are accessible to external programs.

The ARexx interpreter is structured as an Amiga shared library and includes entry points specifically designed to help implement an interface to ARexx. Functions are available to create and delete message packets, argument strings, and other resources. Software developers are urged to use these library routines whenever possible, as they provide "safe" access to the internal structures. The ARexx Systems Library functions are documented in Appendix C. The distribution disk contains the INCLUDE files required to work with the library and data structures.

10-1 Basic Structures

Most developers will need to work with only two of the data structures used by ARexx. The **RexxArg** structure is used for all of the strings manipulated by the interpreter. It is usually passed as an *argstring*, a pointer offset from the structure base that may be treated like an ordinary string pointer. The **RexxMsg** structure is an extension of an EXEC **Message**, and is the message packet used for all communications with external programs.

Argstrings. All ARexx strings are maintained as **RexxArg** structures, which are diagrammed in Table 10.1 below. Note that this actually a variable-length structure allocated for each specific string length. String parameters are sent in the form of *argstrings*, a pointer to the string buffer area of the **RexxArg** structure. The string in the structure is always given a trailing null byte, so that external programs can treat argstrings like a pointer to a null-terminated string. Additional data about the string (its length, hash code, and attributes) are available at negative offsets from the argstring pointer.

Table 10.1 The **RexxArg** Structure

```
STRUCTURE RexxArg,0
LONG     ra_Size        ; allocated length
UWORD    ra_Length      ; length of string
UBYTE    ra_Flags       ; attribute flags
UBYTE    ra_Hash        ; hash code
STRUCT   ra_Buff,8      ; buffer (argstring points here)
```

Library functions are available to create and delete argstrings, and for converting integers into argstring format. The function `CreateArgstring()` allocates a structure and copies a string into it, and returns an argstring pointer to the structure. The function `DeleteArgstring()` can be used to release an argstring when it is no longer needed.

Message Packets. All communications between ARexx and external programs are mediated with message packets, whose structure is diagrammed in Table 10.2 below. Functions are provided in the ARexx Systems Library to create, initialize, and delete these message packets. Each packet sent from ARexx to an external program is marked with a special pointer in its name field. This can be used to distinguish the message packets from those belonging to other programs, in case a message port is being shared.

Message packets are created using the `CreateRexxMsg()` function, and can be released using the `DeleteRexxMsg()` function. Note that the message packets passed by ARexx to a host application (as a command, for instance) are identical to the packets the host would use to invoke an ARexx program. This commonality of design means that only one set of functions is needed to create and delete message packets, and that external programs can use the same routines that the interpreter uses to handle the packets.

Resource Nodes. A somewhat higher-level data structure called a "resource node" (a **RexxRsrc** structure) is used extensively within ARexx to maintain resource lists. These nodes are variable-length structures that include the total allocated length as a field within the node, and that also provide for an "auto-delete" function. This latter capability allows

the address of a clean-up function to be associated with the node so that an entire (possibly inhomogeneous) list of resource nodes can be deallocated with a single function call.

Table 10.2 The **RexxMsg** Structures

```
STRUCTURE RexxMsg,MN_SIZE
APTR     rm_TaskBlock    ; global pointer
APTR     rm_LibBase      ; library pointer
LONG     rm_Action       ; command code
LONG     rm_Result1      ; primary result
LONG     rm_Result2      ; secondary result
STRUCT   rm_Args,16*4    ; arguments (ARG0-ARG15)
                       ; the extension area
APTR     rm_PassPort     ; forwarding port
APTR     rm_CommAddr     ; host address
APTR     rm_FileExt      ; file extension
LONG     rm_Stdin        ; input stream
LONG     rm_Stdout       ; output stream
LONG     rm_avail        ; reserved
LABEL    rm_SIZEOF       ; 128 bytes
```

10-2 Designing a Command Interface

The minimal command interface between ARexx and an applications program requires only a public message port and a routine to process the commands received. For most host applications this will require little extra machinery, as the program will probably already have several message ports for key and menu events, timer messages, and so on. Processing the command strings should be relatively straightforward for command-oriented applications. Hosts that are entirely menu-driven will require somewhat more additional programming, unless commands are supported only as simulated menu events. The specific choice of which commands to support is always left up to the applications designer, as ARexx imposes no restrictions on the structure of the commands that can be issued.

The basic sequence of events in the command interface begins when the host sends a command invocation message to the ARexx resident process. This is usually in response to a direct input from the user, such as a command that was not recognized as one of the primitives supported by the host. When the resident process receives the message packet, it spawns a new DOS process the run the macro program. The command line is parsed to extract the command token (the first word), and the interpreter searches for a file that matches the command name.

Once a macro program file has been found, it is executed by the interpreter and (usually) results in one or more commands being issued back to the host application's public port. The macro program waits while each command is processed by the host, and takes appropriate actions if the return code indicates that an error occurred. Eventually the macro program finishes and returns the invocation message packet back to the host.

Error handling is an important consideration in the interface design. Macro programs must receive meaningful return codes so that processing actions can be altered when errors

occur. Normally, the host application should not return a message packet until the command has been processed and its error status is known. Hosts that support two streams of commands (from the user and from the command interface) will need a flag to indicate the source of each command. Errors in user commands might normally be reported on the screen, but errors in ARexx commands must be reported by setting the result field in the message packet.

Return codes should generally be chosen to follow the model of an *error severity level*, with small integers representing relatively harmless conditions and larger values indicating progressively more severe errors. This will allow a characteristic failure level to be established within a macro program, so that insignificant errors can be ignored. The choice of the specific return code values is left to the applications designer.

Receiving Command Messages

Each host application must open a public message port to support the command interface. When a macro program issues a command to the host, a message packet containing the command is sent to this public port. The structure of these message packets is shown in Table 10.2. The `rm_Action` field will be set to `RXCOMM`, and the `ARG0` parameter slot will contain the command as an argstring pointer. Parameter slots `ARG1`–`ARG15` are not used for command messages. Two other fields are potentially of interest: the `rm_RexxTask` field contains a pointer to the global data structure for the program that issued the command, and the `rm_LibBase` field has the ARexx library base address. The fields in the extension area may also be of interest to the host program; these are described later on. Except for setting the result fields `rm_Result1` and `rm_Result2`, the host application should not alter the message packet.

Result Fields

When the host program finishes processing the command, it must set the primary result field `rm_Result1` to an *error severity level* or zero if no errors occurred. This is the field which will be assigned to the special variable `RC` in the macro program. The secondary result field `rm_Result2` should be set to zero unless a result string (as described below) is being returned. The packet can then be returned to the sender using the EXEC function `ReplyMsg()`.

In some cases a macro program may request a result string by setting the `RXFB_RESULT` modifier bit in the command code. If possible, the host application should then return the result as an argstring pointer in the secondary result field `rm_Result2`. A result string should only be returned if *explicitly requested* and if *no errors occurred* during the call (`rm_Result1` set to zero.) Failure to observe these rules will result either in memory loss or in corruption of the system free-memory list.

Multiple Host Processes

Many applications programs support concurrent activities on several sets of data. For example, most text editors allow several files to be edited at once. A command issued from a particular instance of the editor might invoke an ARexx macro program, so clearly any commands issued from that macro would have to be directed to the correct instance of the editor. ARexx provides for this by allowing the applications program to declare an initial

host address when a program is invoked. A separate message port would be opened for each instance of the host application, and this port would be named as the initial host address for all invocations from that instance. In the example above, if the editor opened two ports named "MyEdit1" and "MyEdit2," then programs invoked by the "MyEdit1" instance would send commands back to the "MyEdit1" port.

Multiple Message Ports. Host applications are not limited to having a single message port for commands. If several different kinds of commands are to be received, it might be appropriate to set up more than one port. Macro programs would then use the `ADDRESS` instruction to direct commands to the appropriate port. The different ports could be used simultaneously, since ARexx programs execute as separate tasks.

10-3 Invoking ARexx Programs

ARexx programs are invoked by sending a message packet to the resident process. Programs may be invoked as either a command or as a function. The command mode of invocation is generally simpler, as it requires setting only a few fields in the message packet.

Message Packets

The structure of the message packet supported by ARexx is shown in Table 10.2. This structure provides fields for passing arguments and for specifying overrides to various internal defaults. The packets are cleared (set to 0) when allocated, and the client-supplied fields are never altered by ARexx. Message packets can be reused after being returned, and generally only one is required.

Command (Action) Code. The `rm_Action` field of the message packet determines the mode of invocation. It can be set to either `RXCOMM` or `RXFUNC` for command or function mode, respectively. Several modifier flags can be used with the command code; these are described later in this chapter.

Argument Strings. Command strings, function names, and argument strings must be supplied as argstrings. Strings can be conveniently packaged into argstrings using the `CreateArgstring()` library function, which takes a string pointer and a length as its arguments. Argstrings point to a null-terminated string and may be treated like an ordinary string pointer in most cases. In principle, a host application could build the argstrings directly, but since the strings must remain unchanged for the duration of an ARexx program, the host might need to maintain many such structures.

The argstring pointer returned by `CreateArgstring()` is installed in the appropriate parameter slot of the message packet: `ARG0` for the command string or function name, and `ARG1`–`ARG15` for argument strings. Argstrings can be recycled after a packet has returned by calling the `DeleteArgstring()` function.

Sending the Packet. Once the required fields have been filled in, the host application can send the packet to the "REXX" public port using the EXEC function `PutMsg()`. The address of the "REXX" port can be obtained by a call to the `FindPort()` function, but this address should not be cached internally, since the port could close at any time. To be absolutely safe, the calls to `FindPort()` and `PutMsg()` should be bracketed by calls to the

EXEC routines `Forbid()` and `Permit()`. This will exclude the slight possibility that the message port could close in the few microseconds before the message packet is actually sent to the port address.

After sending the packet the host can return to its normal processing, since the macro program will execute as a separate task. In most cases it will be advisable to "lock-out" further user commands while the ARexx program is running, to preserve the integrity of any shared data structures that may be accessed externally.

Command Invocations

In the command mode of invocation the host supplies a command string consisting of a name token followed by an argument string. ARexx parses the string to extract the command name, which is usually the name of a program file. The default action is to use the remainder of the command string as the (single) argument string for the program. This may be overridden by requesting *command tokenization*, which is done by setting the `RXFB_TOKEN` modifier flag in the action code of the message packet. In this case the entire command string will be parsed, and the program may have many argument strings. (There is no limit to the number of arguments that may be derived from the command string, since they don't have to fit into the parameter slots of the message packet.)

The parsing process uses "white space" (blanks, tabs, etc.) as the token separators, and has a several special features.

Quoting Convention. Either single (') or double (") quotes may be used to surround items that include "white space" and would otherwise be separated during parsing. Single quotes may appear within a double-quote-delimited token, and vice versa; however, double-delimiter sequences are not accepted. The quotes are removed from the parsed token. An "implicit" quote at the end of the string is also recognized. If the command string ends before the closing delimiter has been found, the null byte is accepted as the final delimiter. For example,

```
look.rexx "Now is the time" "can't you see
```

is a command with the two argument strings "`Now is the time`" and "`can't you see`" (but without the quotes.)

String Files. If the command name (the first token of the string) is quoted, it is assumed to be a "string file" — an ARexx program in a string, rather than the name of a disk file. This is a convenient way to run very brief programs, although programs of any length may be stored this way. If command tokenization has not been specified, the remainder of the string is not scanned and no quote characters are removed. In this case the quoting convention is useful only for indicating "string file" programs. The entire command string can be declared as a "string file" by setting the `RXFB_STRING` modifier flag of the action code. When this flag is set, no parsing at all is applied to the command.

Result Strings. Command invocations do not usually request a result string, but can do so by setting the `RXFB_RESULT` modifier flag. The host application must be prepared to recycle the returned result string once it is no longer needed.

Function Invocations

In a function invocation the host application supplies a function name string and from 0 to 15 argument strings. The name string is used to locate an external program file and may include directory specifiers and a file extension. The actual argument count (not including the name string) must be placed in the low-order byte of the command code.

This mode of invocation is normally used when a result string is expected and the argument strings are conveniently available. Note that a result does not have to be requested, however.

Result Strings. Function invocations request a result string by setting the `RXFB_RESULT` modifier flag bit. If no errors occurred and a result string was requested, the secondary result field in the returned packet will be a pointer to the result string. However, if the program exited without supplying a result, the secondary field will be zero.

String Files. The function name argument may specify a "string file" rather than the name of a filing system object. This is indicated by setting the `RXFB_STRING` modifier flag.

Search Order

The search for a program file matching a command or function name is normally a two-step process. For each directory to be checked, a search is made first with the current file extension appended to the name string. If this search fails, the second search uses the unmodified name string. The first step is skipped if the command or function name includes an explicit file extension.

The default file extension is "`.rexx`," but this can be changed by supplying a file extension string in an extended message packet. Host applications will usually specify a file extension, since it provides a convenient way to distinguish the macro programs that are specific to that application. Refer to the section on Extension Fields for further details.

The search path for a program depends on the way the program name was specified. If an explicit device or directory specification precedes the program name, only that directory will be searched. For example, the command-level invocation of "`rx df0:s/test`" will search only the `df0:s` directory for a file named `test.rexx` or `test`. If the program name does not include a path, the search path begins with the current directory and proceeds to the system `REXX:` directory. To further the above example, invoking the program as "`rx test 1 2 3`" would search for the files `test.rexx`, `test`, `REXX:test.rexx`, and `REXX:test`, in that order.

If an ARexx program cannot be found, one alternative action may be taken. If the `rm_PassPort` field of an extended packet was supplied, the message packet is passed along to that port, which might be the next process in a search chain. Otherwise the message is returned with a "Program not found" error indication (error code 1.)

Extension Fields

The **RexxMsg** structure includes several "extension fields" that can be used to override various defaults when a program is invoked. These extension fields can be filled in selectively, and only the non-zero values will override the corresponding default. ARexx never modifies the extension area.

Host applications should supply values for the file extension and host address fields of the message packet. The file extension affects which program files will match a given command name, and allows macro programs specific to the host to be given distinctive names. The host address must refer to a public message port, and will usually indicate the host's own port. Any appropriate (but usually short) strings can be chosen for these values. Often, the name of the applications program itself can be used as its host address and file extension.

PassPort. Th `rm_PassPort` field allows the search for a program to be "passed along" to another message port after checking for an ARexx program. If the command or function name doesn't resolve to an ARexx program, the message packet is forwarded to the message port specified as the PassPort. This allows applications to maintain control over the search order for external program files.

Note that the `rm_PassPort` field must be the actual *address* of a message port, rather than a name string. The PassPort therefore does not have to be a public port, but the port should be a secured resource, since the message is sent directly to this address without checking to see whether it is a valid message port.

Host Address. The `rm_CommAddr` field overrides the default initial host address, which is "REXX." The host address is the name of the message port to which commands will be directed, and is supplied as a pointer to a null-terminated string. Applications that support multiple instances of user data will usually create a separate message port for each instance. The name of this port would then be supplied as the host address for any commands issued from that instance.

File Extension. The `rm_FileExt` field is used to override the default file extension for ARexx programs, which is "REXX." Host applications can use the file extension to distinguish the names of the macro programs specific to that application. It is supplied as a pointer to a null-terminated string.

Input and Output Streams. The default input and output streams for an ARexx program are inherited from the host application's process structure, if the host is a process rather than just a task. One or both of these streams may be overridden by supplying an appropriate value in the `rm_Stdin` or `rm_Stdout` fields. The values supplied must be valid DOS filehandles, and must not be closed while the program is executing. The streams are installed directly into the program's process structure, replacing the prior values.

The output stream is also used as the default tracing stream for the program. If interactive tracing is to be used in a program, the output stream should refer to a console device, since it will be used for input as well.

In the event that an ARexx program is invoked by an EXEC task, rather than by an DOS process, the extension field streams are the only way that the launched program can be given default I/O streams.

Interpreting the Result Fields

The message packet that invoked an ARexx program is returned to the client when the program finishes. The two result fields will contain error codes or possibly a result string. The interpretation of the result fields depends partly on the mode of invocation. If the primary result field `rm_Result1` is zero, the program executed normally and the secondary field `rm_Result2` will contain a pointer to a result string, assuming that one was requested (and available.)

If the primary result is non-zero, it represents either an error severity level or else the return code from a command invocation. The two cases can be distinguished by examining the secondary result. If the secondary field is also non-zero, an error occurred and the secondary field is an ARexx error code. If the secondary result is zero, then the primary result is the return code passed by an "`EXIT rc`" or "`RETURN rc`" instruction in the program. The application program can use this return code either as an error indication or to initiate some particular processing action.

Result strings are always returned as an argstring and become the property (that is, responsibility) of the host. When the string is no longer needed, it can be released using the `DeleteArgstring()` function.

Errors occurring in macro programs should usually be reported to the user. Explanatory messages are available for all ARexx error codes, and can be obtained by calling the ARexx Systems Library function `ErrorMsg()`.

10-4 Communicating with the Resident Process

All communications with the resident process are handled by passing message packets, which were previously diagrammed in Table 10.2. The packet has a command field that describes the action to be performed and parameter fields that are specific to the command. Message packets are processed as they are received, and are then either returned to the sender or passed along to another process (in the case of a program invocation.) The packet includes two result fields that are used to return error codes or result strings. The parameter fields of the message packet may contain either (long) integer values or pointers to argument strings. String arguments are assumed to be argstring pointers unless otherwise specified.

Command (Action) Codes

The command codes that are currently implemented in the resident process are described below. Commands are listed by their mnemonic codes, followed by the valid modifier flags. The final code value is always the logical OR of the code value and all of the modifier flags selected. The command code is installed in the `rm_Action` field of the message packet.

Usage: `RXADDCON [RXFB_NONRET]`
This code specifies an entry to be added to the Clip List. Parameter slot `ARG0` points to the name string, slot `ARG1` points to the value string, and slot `ARG2` contains the length of

the value string. The name and value arguments do not need to be argstrings, but can be just pointers to storage areas. The name should be a null-terminated string, but the value can contain arbitrary data including nulls.

Usage: RXADDFH [RXFB_NONRET]
This action code specifies a function host to be added to the Library List. Parameter slot ARG0 points to the (null-terminated) host name string, and slot ARG1 holds the search priority for the node. The search priority should be an integer between 100 and -100 inclusive; the remaining priority ranges are reserved for future extensions. If a node already exists with the same name, the packet is returned with a warning level error code. Note that no test is made at this time as to whether the host port exists.

Usage: RXADDLIB [RXFB_NONRET]
This code specifies an entry to be added to the Library List. Parameter slot ARG0 points to a null-terminated name string referring either to a function library or a function host. Slot ARG1 is the priority for the node and should be an integer between 100 and -100 inclusive; the remaining priority ranges are reserved for future extensions. Slot ARG2 contains the entry point offset and slot ARG3 is the library version number. If a node already exists with the same name, the packet is returned with a warning level error code. Otherwise, a new entry is added and the library or host becomes available to ARexx programs. Note that no test is made at this time as to whether the library exists and can be opened.

Usage: RXCOMM [RXFB_TOKEN] [RXFB_STRING] [RXFB_RESULT] [RXFB_NOIO]
Specifies a command-mode invocation of an ARexx program. Parameter slot ARG0 must contain an argstring pointer to the command string. The RXFB_TOKEN flag specifies that the command line is to be tokenized before being passed to the invoked program. The RXFB_STRING flag bit indicates that the command string is a "string file." Command invocations do not normally return result strings, but the RXFB_RESULT flag can be set if the caller is prepared to handle the cleanup associated with a returned string. The RXFB_NOIO modifier suppresses the inheritance of the host's input and output streams.

Usage: RXFUNC [RXFB_RESULT] [RXFB_STRING] [RXFB_NOIO] *argcount*
This command code specifies a function invocation. Parameter slot ARG0 contains a pointer to the function name string, and slots ARG1 through ARG15 point to the argument strings, all of which must be passed as argstrings. The lower byte of the command code is the argument count; this count excludes the function name string itself. Function calls normally set the RXFB_RESULT flag, but this is not mandatory. The RXFB_STRING modifier indicates that the function name string is actually a "string file". The RXFB_NOIO modifier suppresses the inheritance of the host's input and output streams.

Usage: RXREMCON [RXFB_NONRET]
This code requests that an entry be removed from the Clip List. Parameter slot ARG0 points to the null-terminated name to be removed. The Clip List is searched for a node matching the supplied name, and if a match is found the list node is removed and recycled. If no match is found the packet is returned with a warning error code.

Usage: RXREMLIB [RXFB_NONRET]
This command removes a Library List entry. Parameter slot ARG0 points to the null-terminated string specifying the library to be removed. The Library List is searched for a

node matching the library name, and if a match is found the node is removed and released. If no match is found the packet is returned with a warning error code. The library node will not be removed if the library is currently being used by an ARexx program.

Usage: `RXTCCLS [RXFB_NONRET]`
This code requests that the global tracing console be closed. The console window will be closed immediately unless one or more ARexx programs are waiting for input from the console. In this event, the window will be closed as soon as the active programs are no longer using it.

Usage: `RXTCOPN [RXFB_NONRET]`
This command requests that the global tracing console be opened. Once the console is open, all active ARexx programs will divert their tracing output to this console. Tracing input (for interactive debugging) will also be diverted to the new console. Only one console can be opened; subsequent `RXTCOPN` requests will be returned with a warning error message.

Modifier Flags

Command codes may include modifier flags to select various processing options. Modifier flags are specific to certain commands, and are ignored otherwise.

`RXFB_NOIO`. This modifier is used with the `RXCOMM` and `RXFUNC` command codes to suppress the automatic inheritance of the host's input and output streams.

`RXFB_NONRET`. Specifies that the message packet is to be recycled by the resident process rather than being returned to the sender. This implies that the sender doesn't care about whether the requested action succeeded, since the returned packet provides the only means of acknowledgement. Message packets are released using the library function `DeleteRexxMsg()`.

`RXFB_RESULT`. This modifer is valid with the `RXCOMM` and `RXFUNC` commands, and requests that the called program return a result string. If the program `EXIT`s (or `RETURN`s) with an expression, the expression result is returned to the caller as an argstring. It is then the caller's responsibility to release the argstring when it is no longer needed; this can be done using the library function `DeleteArgstring()`.

`RXFB_STRING`. This modifer is valid with the `RXCOMM` and `RXFUNC` command codes. It indicates that the command or function argument (in slot ARG0) is a "string file" rather than a file name.

`RXFB_TOKEN`. This flag is used with the `RXCOMM` code to request that the command string be completely tokenized before being passed to the invoked program. Programs invoked as commands normally have only a single argument string. The tokenization process uses "white space" to separate the tokens, except within quoted strings. Quoted strings can use either single or double quotes, and the end of the command string (a null character) is considered as an implicit closing quote.

Result Fields

The resident process uses the standard command-level conventions for the primary return code installed in `rm_Result1`. Minor or warning errors are indicated by a value of 5, and more serious errors are returned as values of 10 or 20. The secondary result field `rm_Result2` will either be zero or an ARexx error code if applicable.

Note that `RXCOMM` and `RXFUNC` messages are returned directly by the invoked macro program, rather than by the resident process.

10-5 External Function Libraries

ARexx supports external function libraries as a mechanism for user-defined extensions to the language. Function libraries may be written and maintained by users or applications developers.

Design Considerations

There are several different purposes for which a function library might be designed. In the simplest case, a library could be used to extend the string manipulation or mathematical capabilities of the language by defining new functions. Such a library could be entirely self-contained or might call other system libraries to perform specific operations.

Another alternative would be to build a library that interacts closely with an external applications program. This could allow specific operations in the host application to be be performed as function calls rather than as commands. There are several advantages to this approach, as it avoids the need to parse command strings and does not require the multiple task context changes associated with message-passing. The library might include entry points for specific operations as well as functions to support special processing required by the applications program.

Function libraries can also serve as bridges to other system or applications libraries. For example, if a program needed to call the functions in a graphics library, a bridge library could be built to match the function names in the program with the appropriate entry point in the graphics library. A related possibility would be to use ARexx as a test driver for a program under development. Once the query table and parameter passing mechanisms for the function library have been built, new routines under development could be tested by just adding a table entry. Since building test programs is often very time-consuming, the flexibility and interactive debugging capabilities of ARexx make it an attractive alternative to compiled languages like "C."

Regardless of the intended application, all function libraries share a common structure. The initial design follows that of the standard EXEC shared library, with the three required entry points `Open`, `Close`, and `Expunge`, plus a reserved slot. The library must also have a "query" entry point, which serves to match the name supplied by ARexx with the intended function. Typically, this will consist of a table of function names and a routine to search for the specified one.

Reentrancy. Functions libraries should be designed to be fully reentrant, since any number of ARexx programs may be running at any time. If this is not feasible due to other design

constraints, the query function should include a lockout mechanism to prevent multiple calls to the library routines.

Calling Convention

The library's query function will be called from the interpreter's context with the address of a message packet in register A0 and the library base in A6. The structure of the message packet is the same as that in Table 10.2, but note that although a message packet is used to carry the arguments, it is not queued at a message port and does not need to be unlinked. The name of the function to be called is carried in the `ARG0` parameter slot. The query function must search for this function name and, if the name cannot be found, must return an error code of 1 ("Program not found") in register D0. The library will then be closed and the search continued in the next function library. The query function should not modify any fields within the message packet, as it must be passed along to the next library until the function is located.

Parameter Conversion

Once the requested function has been found, the query function may need to transform the parameters passed by ARexx into the form expected by the function. Whether the parameter strings need to be converted depends on how they are to be used. In some cases it may be sufficient just to forward a pointer to the message packet to the called function, while in other cases the query function may need to load parameters into registers or to perform conversion operations. The parameters in `ARG1`–`ARG15` are always passed as argstrings, and may be treated like a pointer to a null-terminated string. Further attributes are stored at negative offsets from the argstring pointer, and may be helpful in working with the string.

Numeric quantities are passed as strings of ASCII characters and will need to be converted to integer or floating-point format if arithmetic calculations are to be performed. The ARexx Systems Library includes a limited set of functions to do parameter conversions.

The actual parameter count can be obtained from the low-order byte of the `rm_Action` field in the message packet. The count never includes the function name itself (in `ARG0`), but does include arguments specified as "defaults." Such arguments will have a zero value in the corresponding parameter slot.

Note that the parameter block of the message packet, containing the fields `ARG0`–`ARG15`, is structured like the argument array expected by the `main(argc,argv)` function of a "C" program. This suggests a simple way that a function library could provide a bridge to a series of "C" programs. The query function would need only to determine the address of the called function, and then push the parameter block address and argument count onto the program stack.

Returned Values

Each library function must return an error code and a value string. The error code is returned in register DO, and should be 0 if no errors occurred. The value string must be returned as an argstring pointer in register A1, unless D0 indicates that an error occurred during the call. The mechanisms for creating the proper return values can be made part of the query function, so that all functions in the library share a common return path.

10-6 Direct Manipulation of Data Structures

All of the data structures maintained by the resident process are built into the ARexx Systems Library base and are therefore accessible to external programs. The Task List in the **RexxBase** structure links the global data structures for all currently active ARexx programs. This linkage uses the node contained in the message port of the **RexxTask** structure, rather than at the head of the structure. The **RexxTask** structure is the global data structure and initial storage environment for the ARexx program, and all descendant storage environments are linked into the Environment List. The linkage of internal data structures is such that the complete internal state of all ARexx programs can be reached starting from the library base pointer.

Two library functions, `LockRexxBase()` and `UnlockRexxBase()`, are provided to mediate access to the global structures. The structure base should be locked before reading any of the data items or traversing any of the lists. The present version of these functions provides only a global lock, but future extensions will allow individual resouces to be locked.

In general it should not be necessary to manipulate directly any of these data structures. Functions have been provided in the ARexx Systems Library to perform all of the operations required to interface external program to the ARexx system. It is therefore recommended that applications developers avoid using any of the internal structures except as provided through the library functions.

Appendix A

Error Messages

When the ARexx interpreter detects an error in a program, it returns an error code to indicate the nature of the problem. Errors are normally handled by displaying the error code, the source line number where the error occurred, and a brief message explaining the error condition. Unless the `SYNTAX` interrupt has been previously enabled (using the `SIGNAL` instruction), the program then terminates and control returns to the caller. Most syntax and execution errors can be trapped by the `SYNTAX` interrupt, allowing the user to retain control and perform whatever special error processing is required. Certain errors are generated outside of the context of an ARexx program, and therefore cannot be trapped by this mechanism. Refer to chapter 7 for further information on error trapping and processing.

Associated with each error code is a *severity level* that is reported to the calling program as the primary result code. The error code itself is returned as the secondary result. The subsequent propagation or reporting of these codes is of course dependent on the external (calling) program.

The following pages list all of the currently-defined error codes, along with the associated severity level and message string.

Error: 1 Severity: 5 Message: `Program not found`
The named program could not be found, or was not an ARexx program. ARexx programs are expected to start with a "`/*`" sequence. This error is detected by the external interface and cannot be trapped by the `SYNTAX` interrupt.

Error: 2 Severity: 10 Message: `Execution halted`
A control-C break or an external halt request was received and the program terminated. This error will be trapped if the `HALT` interrupt has been enabled.

Error: 3 Severity: 20 Message: `Insufficient memory`
The interpreter was unable to allocate enough memory for an operation. Since memory space is required for all parsing and execution operations, this error cannot usually be trapped by the `SYNTAX` interrupt.

Error: 4 Severity: 10 Message: `Invalid character`
A non-ASCII character was found in the program. Control codes and other non-ASCII characters may be used in a program by defining them as hex or binary strings. This is a scan phase error and cannot be trapped by the `SYNTAX` interrupt.

Error: 5 Severity: 10 Message: `Unmatched quote`
A closing single or double quote was missing. Check that each string is properly delimited. This is a scan phase error and cannot be trapped by the SYNTAX interrupt.

Error: 6 Severity: 10 Message: Unterminated comment
The closing "*/" for a comment field was not found. Remember that comments may be nested, so each "/*" must be matched by a "*/." This is a scan phase error and cannot be trapped by the SYNTAX interrupt.

Error: 7 Severity: 10 Message: Clause too long
A clause was too long for the internal buffer used as temporary storage. The source line in question should be broken into smaller parts. This is a scan phase error and cannot be trapped by the SYNTAX interrupt.

Error: 8 Severity: 10 Message: Invalid token
An unrecognized lexical token was found, or a clause could not be properly classified. This is a scan phase error and cannot be trapped by the SYNTAX interrupt.

Error: 9 Severity: 10 Message: Symbol or string too long
An attempt was made to create a string longer than the maximum supported by the interpreter. The implementation limits for internal structures are given in Appendix B.

Error: 10 Severity: 10 Message: Invalid message packet
An invalid action code was found in a message packet sent to the ARexx resident process. The packet was returned without being processed. This error is detected by the external interface and cannot be trapped by the SYNTAX interrupt.

Error: 11 Severity: 10 Message: Command string error
A command string could not be processed. This error is detected by the external interface and cannot be trapped by the SYNTAX interrupt.

Error: 12 Severity: 10 Message: Error return from function
An external function returned a non-zero error code. Check that the correct parameters were supplied to the function.

Error: 13 Severity: 10 Message: Host environment not found
The message port corresponding to a host address string could not be found. Check that the required external host is active.

Error: 14 Severity: 10 Message: Requested library not found
An attempt was made to open a function library included in the Library List, but the library could not be opened. Check that the correct name and version of the library were specified when the library was added to the resource list.

Error: 15 Severity: 10 Message: Function not found
A function was called that could not be found in any of the currently accessible libraries, and could not be located as an external program. Check that the appropriate function libraries have been added to the Libraries List.

Error: 16 Severity: 10 Message: Function did not return value
A function was called which failed to return a result string, but did not otherwise report an error. Check that the function was programmed correctly, or invoke it using the CALL instruction.

Error: 17 Severity: 10 Message: Wrong number of arguments
A call was made to a function which expected more (or fewer) arguments. This error will be generated if a Built-In or external function is called with more arguments than can be accomodated in the message packet used for external communications.

Error: 18 Severity: 10 Message: Invalid argument to function
An inappropriate argument was supplied to a function, or a required argument was missing. Check the parameter requirements specified for the function.

Error: 19 Severity: 10 Message: Invalid PROCEDURE
A PROCEDURE instruction was issued in an invalid context. Either no internal functions were active, or a PROCEDURE had already been issued in the current storage environment.

Error: 20 Severity: 10 Message: Unexpected THEN or WHEN
A WHEN or THEN instruction was executed outside of a valid context. The WHEN instruction is valid only within a SELECT range, and THEN must be the next instruction following an IF or WHEN.

Error: 21 Severity: 10 Message: Unexpected ELSE or OTHERWISE
An ELSE or OTHERWISE was found outside of a valid context. The OTHERWISE instruction is valid only within a SELECT range. ELSE is valid only following the THEN branch of an IF range.

Error: 22 Severity: 10 Message: Unexpected BREAK, LEAVE, or ITERATE
The BREAK instruction is valid only within a DO range or inside an INTERPRETed string. The LEAVE and ITERATE instructions are valid only within an *iterative* DO range.

Error: 23 Severity: 10 Message: Invalid statement in SELECT
A invalid statement was encountered within a SELECT range. Only WHEN, THEN, and OTHERWISE statements are valid within a SELECT range, except for the conditional statements following THEN or OTHERWISE clauses.

Error: 24 Severity: 10 Message: Missing or multiple THEN
An expected THEN clause was not found, or another THEN was found after one had already been executed.

Error: 25 Severity: 10 Message: Missing OTHERWISE
None of the WHEN clauses in a SELECT succeeded, but no OTHERWISE clause was supplied.

Error: 26 Severity: 10 Message: Missing or unexpected END
The program source ended before an END was found for a DO or SELECT instruction, or an END was encountered outside of a DO or SELECT range.

Error: 27 Severity: 10 Message: Symbol mismatch
The symbol specified on an END, ITERATE, or LEAVE instruction did not match the index variable for the associated DO range. Check that the active loops have been nested properly.

Error: 28 Severity: 10 Message: Invalid DO syntax
An invalid DO instruction was executed. An initializer expression must be given if a TO or BY expression is specified, and a FOR expression must yield a non-negative integer result.

Error: 29 Severity: 10 Message: Incomplete IF or SELECT
An IF or SELECT range ended before all of the required statements were found. Check whether the conditional statement following a THEN, ELSE, or OTHERWISE clause was omitted.

Error: 30 Severity: 10 Message: Label not found
A label specified by a SIGNAL instruction, or implicitly referenced by an enabled interrupt, could not be found in the program source. Labels defined dynamically by an INTERPRET instruction or by interactive input are not included in the search.

Error: 31 Severity: 10 Message: Symbol expected
A non-symbol token was found where only a symbol token is valid. The DROP, END, LEAVE, ITERATE, and UPPER instructions may only be followed by a symbol token, and will generate this error if anything else is supplied. This message will also be issued if a required symbol is missing.

Error: 32 Severity: 10 Message: Symbol or string expected
An invalid token was found in a context where only a symbol or string is valid.

Error: 33 Severity: 10 Message: Invalid keyword
A symbol token in an instruction clause was identified as a keyword, but was invalid in the specific context.

Error: 34 Severity: 10 Message: Required keyword missing
An instruction clause required a specific keyword token to be present, but it was not supplied. For example, this message will be issued if a SIGNAL ON instruction is not followed by one of the interrupt keywords (e.g. SYNTAX.)

Error: 35 Severity: 10 Message: Extraneous characters
A seemingly valid statement was executed, but extra characters were found at the end of the clause.

Error: 36 Severity: 10 Message: Keyword conflict
Two mutually exclusive keywords were included in an instruction clause, or a keyword was included twice in the same instruction.

Error: 37 Severity: 10 Message: Invalid template
The template provided with an ARG, PARSE, or PULL instruction was not properly constructed. Refer to Chapter 8 for a description of template structure and processing.

Error: 38 Severity: 10 Message: Invalid TRACE request
The alphabetic keyword supplied with a TRACE instruction or as the argument to the TRACE() Built-In function was not valid. Refer to Chapter 7 for the valid TRACE options.

Error: 39 Severity: 10 Message: Uninitialized variable
An attempt was made to use an uninitialized variable while the NOVALUE interrupt was enabled.

Error: 40 Severity: 10 Message: Invalid variable name
An attempt was made to assign a value to a fixed symbol.

Error: 41 Severity: 10 Message: Invalid expression
An error was detected during the evaluation an expression. Check that each operator has the correct number of operands, and that no extraneous tokens appear in the expression. This error will be detected only in expressions that are actually evaluated. No checking is performed on expressions in clauses that are being skipped.

Error: 42 Severity: 10 Message: Unbalanced parentheses
An expression was found with an unequal number of opening and closing parentheses.

Error: 43 Severity: 43 Message: Nesting limit exceeded
The number of subexpressions in an expression was greater than the maximum allowed. The expression should be simplified by breaking it into two or more intermediate expressions.

Error: 44 Severity: 10 Message: Invalid expression result
The result of an expression was not valid within its context. For example, this message will be issued if an increment or limit expression in a DO instruction yields a non-numeric result.

Error: 45 Severity: 10 Message: Expression required
An expression was omitted in a context where one is required. For example, the SIGNAL instruction, if not followed by the keywords ON or OFF, must be followed by an expression.

Error: 46 Severity: 10 Message: Boolean value not 0 or 1
An expression result was expected to yield a boolean result, but evaluated to something other than 0 or 1.

Error: 47 Severity: 10 Message: Arithmetic conversion error
A non-numeric operand was used in a operation requiring numeric operands. This message will also be generated by an invalid hex or binary string.

Error: 48 Severity: 10 Message: Invalid operand
An operand was not valid for the intended operation. This message will be generated if an attempt is made to divide by 0, or if a fractional exponent is used in an exponentiation operation.

Appendix B

Limits and Compatibility

ARexx was designed to adhere closely to the REXX language standard. This appendix discusses those areas where ARexx departs from the standard.

B-1 Limits

Language definitions seldom include predefined limits to the program structures that can be created. Only a few such restrictions were imposed in implementing ARexx, and most of the internal structures are limited only by the total amount of memory available. The current implementation limits are listed below.

- *Length of Strings.* Strings, symbol names, and value strings are limited to a maximum length of 65,535 bytes.
- *Length of Clauses.* Clauses are limited to a maximum of 800 characters after removing comments and multiple blanks.
- *Nodes in Compound Names.* Compound symbol names may include a maximum of 50 nodes, including the stem.
- *Arguments to Functions.* Built-In and external functions are limited to a maximum of 15 arguments. There is no limit to the number of arguments that may be passed to an internal function.
- *Subexpression Nesting.* The maximum nesting level for subexpressions is 32.

B-2 Compatibility

ARexx departs in a few ways from the language definition. The differences can be classified as omissions or extensions, and are described below.

Omissions. The only significant specification of the language standard omitted from this implementation is the arbitrary-precision arithmetic facility. Arithmetic operations are limited to about 14 digits of precision, and the `FUZZ` option is not implemented at all. Only the `SCIENTIFIC` format is used for exponential notation. The full numeric capabilities will be provided in a later release.

Extensions. The following extensions to the language standard have been included in this implementation:

- `BREAK` *Instruction.* A new instruction called `BREAK` has been implemented. It is used to exit from the scope of any `DO` or `INTERPRET` instruction.
- `ECHO` *Instruction.* The `ECHO` instruction has been included as a synonym for `SAY`.
- `SHELL` *Instruction.* The `SHELL` instruction has been included as a synonym for `ADDRESS`.

- `SIGNAL` *Options.* Several additional `SIGNAL` keywords have been implemented. `BREAK_C`, `BREAK_D`, `BREAK_E`, and `BREAK_F` will detect and trap the control-C through control-F signals passed by AmigaDOS. The `IOERR` keyword traps errors detected by the I/O system.

- *Stem Symbols.* A stem symbol is valid anywhere that a simple symbol could be employed.

- *Template Processing.* Templates have been generalized in several ways. Variable symbols may be used as positional tokens if preceded by an operator; the "=" operator is used to denote an absolute position. Multiple templates can be used with all source forms of the `PARSE` instruction.

Appendix C

The ARexx Systems Library

The ARexx interpreter is supplied as a shared library named `rexxsyslib.library` and should reside in the system `LIBS:` directory. While many of the library routines are highly specific to the interpreter, some of the functions will be useful to applications that use ARexx. The library is opened when the ARexx resident process is first loaded and will always be available while it remains active.

The system library routines were designed to be called from assembly-language programs and, unless otherwise noted, save all registers except for A0/A1 and D0/D1. Many routines return values in more than one register to help reduce code size. In addition, the routines will set the condition-code register (CCR) wherever appropriate. In most cases the CCR reflects the value returned in D0.

The library offsets are defined in the file `rxslib.i`, which should be INCLUDEd in the program source code. Calls may be made from "C" programs if suitable binding routines are provided when the program is linked. The definitions for the constants and data structures used in ARexx are provided as INCLUDE files on the program distribution disk. These should be reviewed carefully before attempting to use the library functions.

Some of the library functions are not documented here. These private entry points are reserved for the internal use of the interpreter and should not be called from external programs.

C-1 Functional Groups

The library functions can be grouped into *Conversion*, *Input/Output*, *Resource Management*, and *String Manipulation* functions.

Data Conversion. These functions provide many of the common data-conversion requirements.

Input/Output. Two levels of I/O support are provided. The low level functions use DOS filehandles directly, while the higher-level functions use linked lists of **IoBuff** structures and support logical file names.

Resource. These functions allocate, release, or otherwise manage the data structures used with ARexx.

String Functions. All data in ARexx are managed as strings. These functions provide some of the more common string-manipulation operations.

Table C.1 ARexx Systems Library Functions

Name	Functional Group	Description
AddClipNode	Resource	Allocate a Clip node
ClearMem	Resource	Clear a block of memory
ClearRexxMsg	Resource	Release argstrings from message
CloseF	Input/Output	Close a file buffer
ClosePublicPort	Resource	Close a port resource node
CmpString	String	Compare string structures for equality
CreateArgstring	Resource	Create an argstring structure
CreateDOSPkt	Input/Output	Create a DOS StandardPacket
CreateRexxMsg	Resource	Create a message packet
CurrentEnv	Resource	Get current storage environment
CVa2i	Conversion	ASCII to integer
CVc2x	Conversion	Character to Hex or Binary digits
CVi2a	Conversion	Integer to ASCII
CVi2arg	Conversion	Integer to ASCII argstring
CVi2az	Conversion	Integer to ASCII, leading zeroes
CVs2i	Conversion	String structure to integer
CVx2c	Conversion	Hex or binary digits to binary
DeleteArgstring	Resource	Release an argstring structure
DeleteDOSPkt	Input/Output	Release a DOS StandardPacket
DeleteRexxMsg	Resource	Release a message packet
DOSRead	Input/Output	Read from a DOS filehandle
DOSWrite	Input/Output	Write to a DOS filehandle
ErrorMsg	Conversion	Get error message from error code
ExistF	Input/Output	Check whether a DOS file exists
FillRexxMsg	Resource	Convert and install argstrings
FindDevice	Input/Output	Locate a DOS device node
FindRsrcNode	Resource	Locate a resource node
FreePort	Resource	Close a message port
FreeSpace	Resource	Release internal memory
GetSpace	Resource	Allocate internal memory
InitList	Resource	Initialize a list header
InitPort	Resource	Initialize a message port
IsRexxMsg	Resource	Test a message packet
LengthArgstring	Resource	Get length of argstring
ListNames	Resource	Copy node names to an argstring
OpenF	Input/Output	Open a file buffer
OpenPublicPort	Resource	Allocate and open a port resource node
QueueF	Input/Output	Queue a line in a file buffer
ReadF	Input/Output	Read from a file buffer
ReadStr	Input/Output	Read a string from a file buffer
RemClipNode	Resource	Release a Clip node
RemRsrcList	Resource	Release a resource list
RemRsrcNode	Resource	Release a resource node

Table C.1 Library Functions (cont.)

Name	Functional Group	Description
SeekF	Input/Output	Reposition a file buffer
StackF	Input/Output	Stack a line in a file buffer
StcToken	String	Break out a token
StrcmpN	String	Compare strings
StrcpyA	String	Copy a string, converting to ASCII
StrcpyN	String	Copy a string
StrcpyU	String	Copy a string, converting to uppercase
StrflipN	String	Reverse characters in a string
Strlen	String	Find length of a string
ToUpper	Conversion	ASCII to uppercase
WriteF	Input/Output	Write to a file buffer

C-2 Library Functions

The following descriptions of the ARexx Systems Library functions are listed alphabetically. The required arguments and register assignments are shown in parentheses after the function name. Multiple returns are shown in parentheses on the left-hand side of the call.

AddClipNode()– allocate and link a Clip node

```
Usage: node = AddClipNode(list,name,length,value)
        D0                 A0   A1    D0     D1
        A0
       (CCR)
```

Allocates and links a Clip node into the specified list. Clip nodes are resource nodes containing a name and value string, and include an "auto-delete" function for simple maintenance. The *list* argument must point to a properly-initialized EXEC list header. The *name* argument points to a null-terminated name string, the *value* argument is a pointer to a storage area, and the *length* argument is its length in bytes. The returned value is a pointer to the allocated node, or 0 if the allocation failed.

The **RemClipNode()** function is installed as the "auto-delete" function for each node. Clip nodes can be intermixed with other resource nodes in a list and then released with a single call to **RemRsrcList()**.
See Also: **RemClipNode(), RemRsrcList(), RemRsrcNode()**

AddRsrcNode()– allocate and link a resource node

```
Usage: node = AddRsrcNode(list,name,length)
        D0                 A0   A1    D0
        A0
       (CCR)
```

Allocates and links a resource node (a **RexxRsrc** structure) to the specified *list*. The *name* argument is a pointer to a null-terminated string, a copy of which is installed in the node structure. The length argument is the total length for the node; this length is saved within

the node so that it may be released later. The returned value is a pointer to the allocated node, or 0 if the allocation failed.
See Also: **RemRsrcList(), RemRsrcNode()**

ClearMem()– clear a block of memory
```
Usage: ClearMem(address,length)
                  A0       D0
```

Clears a block of memory beginning at the given *address* for the specified *length* in bytes. The address must be word-aligned and the length must be a multiple of 4 bytes; all structures allocated by ARexx meet these restrictions. Register A0 is preserved.

ClearRexxMsg()– release argument strings
```
Usage: ClearRexxMsg(msgptr,count)
                       A0     D0
```

Releases one or more argstrings from a message packet and clears the corresponding slots. The *count* argument specifies the number of argument slots to clear, and can be set to less than 16 to reserve some to the slots for private use. No action is taken if the slot already contains a zero value.
See Also: **FillRexxMsg()**

CloseF()– close a file buffer
```
Usage: boolean = CloseF(IoBuff)
         D0               A0
```

Releases the **IoBuff** structure and closes the associated DOS file. `CloseF()` is the "auto-delete" function for the **IoBuff** structure, so an entire list of file buffers can be closed with a single call to `RemRsrcList()`.

ClosePublicPort() – close a port resource node
```
Usage: ClosePublicPort(node)
                         A0
```

Unlinks and closes the message port and releases the resource node structure. The node must have been allocated by the `OpenPublicPort()` function.
See Also: **OpenPublicPort()**

CmpString()– compare string structures for equality
```
Usage: test = CmpString(ss1,ss2)
        D0               A0  A1
       (CCR)
```

The arguments *ss1* and *ss2* must be pointers to ARexx string structures and are compared for equality. String structures include the length and hash code of the string, so the actual strings are not compared unless the lengths and hash codes match. The return value sets the CCR and will be -1 (**True**) if the strings match and 0 (**False**) otherwise.

CreateArgstring()– create an argument string structure
```
Usage: argstring = CreateArgstring(string,length)
         D0                        A0      D0
         A0
        (CCR)
```

Allocates a **RexxArg** structure and copies the supplied string into it. The *argstring* return is a pointer to the string buffer of the structure, and can be treated like an ordinary string pointer. The **RexxArg** structure stores the structure size and string length at negative offsets to the string pointer. The string pointer can be set to NULL if only an uninitialized structure is required.
See Also: **DeleteArgstring()**

CreateDOSPkt()– allocate and initialize a DOS standardPacket.
```
Usage: packet = CreateDOSPkt()
         D0
         A0
        (CCR)
```

Allocates a DOS **StandardPacket** structure and initializes it by interlinking the EXEC message and the DOS packet substructures. No replyport is installed in either the message or the packet, as these fields are generally filled in just before the message is sent.
See Also: **DeleteDOSPkt()**

CreateRexxMsg()– allocate an ARexx message packet
```
Usage: msgptr = CreateRexxMsg(replyport,extension,host)
         D0                   A0          A1       D0
         A0
        (CCR)
```

This functions allocates an ARexx message packet from the system free memory list. The message packet consists of a standard EXEC message structure extended to include space for function arguments, returned results, and internal defaults. The *replyport* argument points to a public or private message port and must be supplied, as it is required to return the message packet to the sender. The *extension* and *host* arguments are pointers to null-terminated strings that provide values for the default file extension and the initial host address, respectively. Additional override fields in the extended packet may be filled in after the packet has been allocated.

The interpreter preserves all of the fields in the message packet except for the primary and secondary result fields `rm_Result1` and `rm_Result2`.
See Also: **DeleteRexxMsg()**

CVa2i()– convert from ASCII to integer
```
Usage: (digits,value) = CVa2i(buffer)
         D0      D1             A0
```

Converts the *buffer* of ASCII characters to a signed long integer value. The scan proceeds

until a non-digit character is found or until an overflow is detected. The function returns both the number of digits scanned and the converted value.

CVc2x()– convert (unpack) from character string to hex or binary digits.

```
Usage: error = CVc2x(outbuff,string,length,mode)
         D0              A0      A1      D0     D1
```

Converts the *string* argument to a string of hex (0-9, A-F) or binary (0,1) digits.

CVi2a()– convert from integer to ASCII

```
Usage: (length,pointer) = CVi2a(buffer,value,digits)
          D0      A0                A0      D0     D1
```

Converts the signed integer *value* argument to ASCII characters using the supplied *buffer* pointer. The *digits* argument specifies the maximum number of characters that will be copied to the buffer. The returned *length* is the actual number of characters copied. The *pointer* return is the new buffer pointer.
See Also: **CVi2az()**

CVi2arg()– convert from integer to argstring

```
Usage: argstring = CVi2arg(value,digits)
          D0                  D0     D1
          A0
         (CCR)
```

Converts the signed long integer *value* argument to ASCII characters, and installs them in an argstring (a **RexxArg** structure). The returned value is an argstring pointer or 0 if the allocation failed. The allocated structure can be released using `DeleteArgstring()`.

CVi2az()– convert from integer to ASCII with leading zeroes

```
Usage: (length,pointer) = CVi2az(buffer,value,digits)
         D0      A0                 A0      D0     D1
```

Converts the signed long integer *value* argument to ASCII characters in the supplied *buffer*, with leading zeroes to fill out the requested number of *digits*. This function is identical to CVi2a except that leading zeroes are supplied.
See Also: **CVi2a()**

CVs2i()– convert from string structure to integer

```
Usage: (error,value) = CVs2i(ss)
          D0     D1           A0
```

The *ss* argument must be a pointer to a string structure. It is converted to a signed long integer *value* return. The *error* return code is 47 ("Arithmetic conversion error") if the string is not a valid number.

CVx2c()– convert from hex or binary digits to (packed) string
```
Usage: error = CVx2c(outbuff,string,length,mode)
         D0              A0      A1      D0     D1
```

Converts the *string* argument of hex (0-9,A-F) or binary (0,1) digits to the packed binary representation. The *mode* argument specifies the (hex or binary) conversion mode, and must be set to -1 for hex strings or 0 for binary strings. Blank characters may be embedded in the string for readability, but only at byte boundaries. The *error* return code is 47 if the string is not a valid hex or binary string.

CurrentEnv()– return the current storage environment
```
Usage: envptr = CurrentEnv(rxtptr)
         D0                  A0
```

Returns a pointer to the current storage environment associated with an executing ARexx program. The *rxtptr* argument is a pointer to the **RexxTask** structure, and may be obtained from the message packet sent to an external application.

DeleteArgstring()– delete (release) an argstring structure
```
Usage: DeleteArgstring(argstring)
                         A0
```

Releases an argstring (**RexxArg**) structure. The **RexxArg** structure contains the total allocated length at a negative offset from the *argstring* pointer.
See Also: **CreateArgstring()**

DeleteDOSPkt()– release a DOS StandardPacket structure.
```
Usage: DeleteDOSPkt(message)
                      A0
```

Releases a DOS **StandardPacket** structure, typically obtained by a prior call to `CreateDOSPkt()`.
See Also: **CreateDOSPkt()**

DeleteRexxMsg()– delete (release) an ARexx message packet
```
Usage: DeleteRexxMsg(packet)
                       A0
```

Releases an ARexx message packet to the system free-memory list. The internal `MN_LENGTH` field is used as the total size of the memory block to be released, so this function can be used to release any message packet that contains the total length in this field. Any embedded argument strings must be released before calling `DeleteRexxMsg()`.
See Also: **CreateRexxMsg()**

DOSRead()– read from a DOS file

```
Usage: count = DOSRead(filehandle,buffer,length)
         D0                A0        A1     D0
        (CCR)
```

Reads one or more characters from a DOS filehandle into the supplied buffer. The *length* argument specifies the maximum number of characters that will be read. The returned *count* is the actual number of bytes transferred, or -1 if an error occurred.

DOSWrite()– write to a DOS file

```
Usage: count = DOSWrite(filehandle,buffer,length)
         D0                 A0        A1      D0
        (CCR)
```

Writes a buffer of the specified length to a DOS filehandle. The returned *count* is the actual number of bytes written, or -1 if an error occurred.

ErrorMsg()– find the message associated with an error code

```
Usage: (boolean,ss) = ErrorMsg(code)
          D0    A0              D0
```

Returns the error message (as a pointer to a string structure) associated with the specified ARexx error code. The boolean return will be -1 if the supplied code was a valid ARexx error code, and 0 otherwise.

ExistF()– check whether an external file exists

```
Usage: boolean = ExistF(filename)
         D0                A0
        (CCR)
```

Tests whether an external file currently exists by attempting to obtain a read lock on the file. The boolean return indicates whether the operation succeeded, and the lock is released.

FillRexxMsg()– convert and install arguments in message packet.

```
Usage: boolean = FillRexxMsg(msgptr,count,mask)
         D0                   A0     D0     D1
        (CCR)
```

This function can be used to convert and install up to 16 argument strings in a **RexxMsg** structure. The message packet must be allocated and the argument fields of interest set to either a pointer to a null-terminated string or an integer value. The *count* argument specifies the number of fields, beginning with ARG0, to be converted into argstrings and installed into the argument slot. Bits 0-15 of the *mask* argument specify whether the corresponding argument is a string pointer (bit clear) or an integer value (bit set).

The *count* argument is normally set to the exact number of strings to be passed. By setting this count to less than 16, a number of the slots can be reserved for private uses.

The returned value is -1 (**True**) if all of the arguments were successfully converted. In the event of an allocation failure, all of the partial results are released and a value of 0 is returned.
See Also: **ClearRexxMsg()**

FindDevice()– check whether a DOS device exists.

```
Usage: device = FindDevice(devicename,type)
         D0                    A0        D0
         A0
        (CCR)
```

Scans the DOS DeviceList for a device node of the specified type matching the null-terminated name string. The acceptable values for the *type* argument are the constants `DLT_DEVICE`, `DLT_DIRECTORY`, or `DLT_VOLUME` defined in the DOS INCLUDE files. Device names are converted to uppercase before checking for a match. The returned value is a pointer to the matched device node, or 0 if the device was not found.

FindRsrcNode()– locate a resource node with the given name.

```
Usage: node = FindRsrcNode(list,name,type)
        D0                   A0   A1   D0
        A0
       (CCR)
```

Searchs the specified list for the first node of the selected type with the given name. The *list* argument must be a pointer to a properly-initialized EXEC list header. The *name* argument is a pointer to a null-terminated string. If the *type* argument is 0, all nodes are selected; otherwise, the supplied type must match the `LN_TYPE` field of the node. The returned value is a pointer to the node or 0 if no matching node was found.

FreePort()– release resources associated with a message port

```
Usage: FreePort(port)
                 A0
```

This function deallocates the signal bit associated with a message port and marks the port as "closed." The task calling `FreePort()` must be the same one that initialized the port, since signal bit allocations are specific to a task. The memory space associated with the port is not released.
See Also: **InitPort()**

FreeSpace()– releases space to the internal memory allocator.

```
Usage: FreeSpace(envptr,block,length)
                   A0     A1     D0
```

Returns a block of memory to the internal allocator, which must have been obtained from a call to `GetSpace()`. The *envptr* argument is a pointer to the base or current storage

environment.
See Also: **CurrentEnv(), GetSpace()**

GetSpace()– allocate memory using the internal allocator.

```
Usage:  block = GetSpace(envptr,length)
          D0                A0      D0
          A0
         (CCR)
```

Allocates a block of memory using the internal allocator. The memory is obtained from an internal pool managed by the interpreter and is returned to the operating system when the ARexx program terminates. The *envptr* argument is a pointer to the base or current storage environment for the program.

The internal allocator must be used to allocate strings for use as values for symbols, and is convenient for obtaining small blocks of memory whose lifetime will not exceed that of the ARexx program.
See Also: **CurrentEnv(), FreeSpace()**

InitList()– initialize a list header

```
Usage:  InitList(list)
                  A0
```

Initializes an EXEC list header structure.

InitPort()– initialize a previously-allocated message port.

```
Usage:  (signal,port) = InitPort(port,name)
          D0     A1                A0   A1
```

Initializes a message port structure for which memory space has been previously allocated, typically as part of a larger structure or as static storage in a program. It installs the task ID (of the task calling the function) into the `MP_SIGTASK` field and allocates a signal bit. The *name* parameter must be a pointer to a null-terminated string. The *signal* return is the signal bit that was allocated for the port. In the event that a signal could not be assigned, a value of -1 is returned.

Note that the port is not linked into the system Ports List. If the port is to be made public, this can be done after the function returns. The port address is returned in the correct register (A1) for a subsequent call to the EXEC function `AddPort()`.
See Also: **FreePort()**

IsRexxMsg()– check whether a message came from ARexx.

```
Usage: boolean = IsRexxMsg(msgptr)
         D0                 A0
```

Tests whether the message packet specified by the *msgptr* argument came from an ARexx program. ARexx marks its messages with a pointer to a static string "REXX" in the

LN_NAME field. The returned value is either -1 (**True**) if the message came from ARexx or 0 (**False**) otherwise.

IsSymbol()– check whether a string is a valid symbol.

```
Usage: (code,length) = IsSymbol(string)
         D0    D1                 A0
```

Scans the supplied string pointer for ARexx symbol characters. The *code* return is the symbol type if a symbol was found, or 0 if the string did not start with a symbol character. The *length* return is the total length of the symbol.

ListNames()– build a string of names from a list.

```
Usage:  argstring = ListNames(list,separator)
          D0                  A0   D0[0:7]
          A0
         (CCR)
```

Scans the specified list and copies the name strings into an argstring. The *list* argument must be a pointer to an initialized EXEC list header. The *separator* argument is the character, possibly a null, to be placed as a delimiter between the node names.

The list is traversed inside a Forbid() exclusion and so may be used with shared or system lists. The returned argstring can be released using `DeleteArgstring()` after the names are no longer needed.
See Also: **DeleteArgstring()**

LockRexxBase()– lock a shared resource.

```
Usage:  LockRexxBase(resource)
                       D0
```

Secures the specified resource in the ARexx Systems Library base for read access. The *resource* argument is a manifest constant for the required resource, or zero to lock the entire structure.

Note that write access to shared resources is normally mediated by the ARexx resident process, which operates at an elevated priority to gain exclusive access. Locking a resource should not be attempted from a process operating at a priority higher than the resident process.
See Also: **UnlockRexxBase()**

OpenF()– open a file buffer

```
Usage: IoBuff = OpenF(list,filename,mode,logical)
          D0           A0     A1     D0    D1
          A0
         (CCR)
```

Attempts to open an external file in the specified mode, which should be one of the constants `RXIO_READ`, `RXIO_WRITE`, or `RXIO_APPEND` defined in the ARexx INCLUDE files. If successful,

an **IoBuff** structure is allocated and linked into the specified list. The *list* argument must be a pointer to a properly-initialized EXEC list header.

The optional *logical* argument is the logical name for the file, and must be either a pointer to a null-terminated string or zero (`NULL`) if a name is not required.
See Also: **CloseF()**

OpenPublicPort()– open a public message port

```
Usage: node = OpenPublicPort(list,name)
        D0                    A0   A1
        A0
       (CCR)
```

Allocates a message port as an "auto-delete" resource node and links it into the specified list. The *list* argument must point to a properly initialized EXEC list header. The message port is initialized with the given name and linked into the system Ports List.
See Also: **ClosePublicPort()**

QueueF()– queue a line to a file buffer.

```
Usage: count = QueueF(IoBuff,buffer,length)
         D0             A0      A1      D0
```

Queues a buffer of characters in the stream associated with the **IoBuff** structure. The stream must be managed by a DOS handler that supports the `ACTION_QUEUE` packet.

Queued lines are placed in "firtst-in, first-out" order and are immediately available to be read from the stream. The *buffer* argument is a pointer to a string of characters, and the *length* specifies the number of characters to be queued. The return value is the actual count of characters or -1 if an error occurred.
See Also: **StackF()**

ReadF()– read characters from a file buffer

```
Usage: count = ReadF(IoBuff,buffer,length)
         D0             A0      A1      D0
        (CCR)
```

Reads one or more characters from the file specified by the **IoBuff** pointer. The *buffer* argument is a pointer to a storage area, and the *length* argument specifies the maximum number of characters to be read. The return value is the actual number of characters read, or -1 if an error occurred.

ReadStr()– read a string from a file

```
Usage: (count,pointer) = ReadStr(IoBuff,buffer,length)
         D0    A1                 A0      A1      D0
```

Reads characters from the file specified by the **IoBuff** pointer until a "newline" character is found. The "newline" is not included in the returned string. The return value is the actual number of characters read, or -1 if an error occurred.

See Also: **ReadF()**

RemClipNode()– unlink and deallocate a list Clip node.
```
Usage: RemClipNode(node)
                   A0
```

Unlinks and releases the specified Clip node. The function is the "auto-delete" function for Clip nodes, and will be called automatically by `RemRsrcNode()` or `RemRsrcList()`.
See Also: **AddClipNode(), RemRsrcList(), RemRsrcNode()**

RemRsrcList()– unlink and deallocate a list of resource nodes
```
Usage: RemRsrcList(list)
                   A0
```

Scans the supplied list and releases any nodes found. The list must consist of resource nodes (**RexxRsrc** structures), which contain information to allow automatic cleanup and deletion.
See Also: **RemRsrcNode()**

RemRsrcNode()– unlink and deallocate a resource node
```
Usage: RemRsrcNode(node)
                   A0
```

Unlinks and releases the specified resource node, including the name string if one is present. If an "auto-delete" function has been specified in the node, it is called to perform any required resource deallocation before the node is released.
See Also: **RemRsrcList()**

SeekF()– seek to the specified position in a file.
```
Usage: position = SeekF(IoBuff,offset,anchor)
         D0             A0     D0     D1
```

Seeks to a new position in the file is specified by the *IoBuff* pointer. The position is given by the *offset* argument, a byte offset relative to the supplied *anchor* argument. The anchor may specify the beginning (-1), the current position (0), or the end of the file (1). The return value is the new position relative to the beginning of the file.

StackF()– stack a line to a file buffer.
```
Usage: count = StackF(IoBuff,buffer,length)
         D0            A0     A1     D0
```

Stacks a buffer of characters in the stream associated with the **IoBuff** structure. The *buffer* argument is a pointer to a string of characters, and the *length* specifies the number of characters to be stacked. The return value is the actual count of characters or -1 if an error occurred.

Stacked lines are placed in "last-in, first-out" order and are immediately available to be read from the stream. The stream must be managed by a DOS handler that supports the ACTION_STACK packet.
See Also: **QueueF()**

StcToken()– break out the next token from a string
```
Usage: (quote,length,scan,token) = StcToken(string)
         D0      D1     A0   A1                  A0
```

Scans a null-terminated string to select the next token delimited by "white space," and returns a pointer to the start of the token. The *quote* return will be an ASCII single or double quote if the token was quoted and 0 otherwise; white space characters are ignored within quoted strings. The *length* return is the total length of the token, including any quote characters. The *scan* return is advanced beyond the current token to prepare for the next call.

StrcpyA()– copy a string, converting to ASCII
```
Usage: hash = StrcpyA(destination,source,length)
        D0                 A0         A1      D0
```

Copies the *source* string to the *destination* area, converting the characters to ASCII by clearing the high-order bit of each byte. The *length* of the string (which may include embedded nulls) is considered as a 2-byte unsigned integer. so the string is limited in length to 65,535 bytes. The *hash* return is the internal hash byte for the copied string.
See Also: **StrcpyN(), StrcpyU**

StrcpyN()– copy a string
```
Usage: hash = StrcpyN(destination,source,length)
        D0                 A0         A1      D0
```

Copies the *source* string to the *destination* area. The *length* of the string (which may include embedded nulls) is considered as a 2-byte unsigned integer. The *hash* return is the internal hash byte for the copied string.
See Also: **StrcpyA(), StrcpyU**

StrcpyU()– copy a string, converting to uppercase
```
Usage: hash = StrcpyU(destination,source,length)
        D0                 A0         A1      D0
```

Copies the *source* string to the *destination* area, converting to uppercase alphabetics. The *length* of the string (which may include embedded nulls) is considered as a 2-byte unsigned integer. The *hash* return is the internal hash byte for the copied string.
See Also: **StrcpyA(), StrcpyN**

StrflipN()– reverse the characters in a string
```
Usage: StrflipN(string,length)
                  A0      D0
```

Reverses the sequence of characters in a string. The conversion is performed in place.

Strlen()– find the length of a null-terminated string
```
Usage: length = Strlen(string)
         D0              A0
        (CCR)
```

Returns the number of characters in a null-terminated string. Register A0 is preserved, and the CCR is set for the returned length.

StrcmpN()– compare the values of strings
```
Usage: test = StrcmpN(string1,string2,length)
        D0              A0      A1      D0
       (CCR)
```

The *string1* and *string2* arguments are compared for the specified number of characters. The comparison proceeds character-by-character until a difference is found or the maximum number of characters have been examined. The returned value is -1 if the first string was less, 1 if the first string was greater, and 0 if the strings match exactly. The CCR register is set for the returned value.

ToUpper()– translate an ASCII character to uppercase
```
Usage: upper = ToUpper(character)
        D0              D0
```

Converts an ASCII character to uppercase. Only register D0 is affected.

UnlockRexxBase()– unlock a shared resource.
```
Usage:  UnlockRexxBase(resource)
                        D0
```

Releases the specified resource, or all resources if the argument is zero. Every call to `LockRexxBase()` should be followed eventually by a call to `UnlockRexxBase()` for the same resource.
See Also: **LockRexxBaseF()**

WriteF()– write characters to a file buffer
```
Usage: count = WriteF(IoBuff,buffer,length)
         D0             A0      A1      D0
        (CCR)
```

Writes a buffer of characters of the specified length to the file associated with the **IoBuff** pointer. The *buffer* argument is a pointer to a storage area, and the *length* argument

specifies the number of characters to be written. The returned value is the actual number of characters written or -1 if an error occurred.
See Also: **CloseF(), OpenF(), ReadF()**

Appendix D

The ARexx Support Library

The ARexx language system is distributed with an external function library that provides a number of Amiga-specific functions. It is a standard Amiga shared library named `rexxsupport.library` and should reside in the system `LIBS:` directory. Unlike the Systems Library described in the previous Appendix, the support library functions are callable from with ARexx programs.

The support library was designed to supplement the generic Built-In functions with functions specific to the Amiga. This library will be expanded in future releases, and users are encouraged to submit suggestions for additional functions.

The Support Library must be added to the global Library List before it can be accessed by ARexx programs. This can be done using the Built-In function `ADDLIB()` or by direct communication with the resident process. The library name must be specified as `rexxsupport.library`, the query function offset is -30, and the version number is 0. The search priority can be set to 0 or whatever value is appropriate.

ALLOCMEM()

Usage: `ALLOCMEM(`*length,[attribute]*`)`
Allocates a block of memory of the specified length from the system free-memory pool and returns its address as a 4-byte string. The optional *attribute* parameter must be a standard EXEC memory allocation flag, supplied as a 4-byte string. The default attribute is for "PUBLIC" memory (not cleared).

This function should be used whenever memory is allocated for use by external programs. It is the user's responsibility to release the memory space when it is no longer needed.
See Also: `FREEMEM()`
Example:

```
say c2x(allocmem(1000))       ==> 00050000
```

CLOSEPORT()

Usage: `CLOSEPORT(`*name*`)`
Closes the message port specified by the *name* argument, which must have been allocated by a call to `OPENPORT()` within the current ARexx program. Any messages received but not yet `REPLY`ed are automatically returned with the return code set to 10.
See Also: `OPENPORT()`
Example:

```
call closeport myport
```

FREEMEM()

Usage: FREEMEM(*address,length*)
Releases a block of memory of the given length to the system freelist. The *address* parameter is a four-byte string, typically obtained by a prior call to ALLOCMEM(). FREEMEM() cannot be used to release memory allocated using GETSPACE(), the ARexx internal memory allocator. The returned value is a boolean success flag.
See Also: ALLOCMEM()
Example:

```
say freemem('00042000'x,32)      ==> 1
```

GETARG()

Usage: GETARG(*packet,[n]*)
Extracts a command, function name, or argument string from a message packet. The *packet* argument must be a 4-byte address obtained from a prior call to GETPKT(). The optional *n* argument specifies the slot containing the string to be extracted, and must be less than or equal to the actual argument count for the packet. Commands and function names are always in slot 0; function packets may have argument strings in slots 1–15.
Examples:

```
command  = getarg(packet)
function = getarg(packet,0)  /* name string  */
arg1     = getarg(packet,1)  /* 1st argument */
```

GETPKT()

Usage: GETPKT(*name*)
Checks the message port specified by the *name* argument to see whether any messages are available. The named message port must have been opened by a prior call to OPENPORT() within the current ARexx program. The returned value is the 4-byte address of the first message packet, or '0000 0000'x if no packets were available.

The function returns immediately whether or not a packet is enqueued at the message port. Programs should never be designed to "busy-loop" on a message port. If there is no useful work to be done until the next message packet arrives, the program should call WAITPKT() and allow other tasks to proceed.
See Also: WAITPKT()
Example:

```
packet = getpkt('MyPort')
```

OPENPORT()

Usage: OPENPORT(*name*)
Creates a public message port with the given name. The returned value is the 4-byte address of the Port Resource structure or '0000 0000'x if the port could not be opened or initialized. An initialization failure will occur if another port of the same name already exists, or if a signal bit couldn't be allocated.

The message port is allocated as a Port Resource node and is linked into the program's global data structure. Ports are automatically closed when the program exits, and any pending messages are returned to the sender.
See Also: `CLOSEPORT()`
Example:

```
myport = openport("MyPort")
```

REPLY()

Usage: `REPLY(`*packet,rc*`)`
Returns a message packet to the sender, with the primary result field set to the value given by the *rc* argument. The secondary result is cleared. The *packet* argument must be supplied as a 4-byte address, and the *rc* argument must be a whole number.
Example:

```
call reply packet,10            /* error return */
```

SHOWDIR()

Usage: `SHOWDIR(`*directory,['All' | 'File' | 'Dir']*`)`
Returns the contents of the specified directory as a string of names separated by blanks. The second parameter is an option keyword that selects whether all entries, only files, or only subdirectories will be included.
Example:

```
say showdir("df1:c")            ==> rx ts te hi tco tcc
```

SHOWLIST()

Usage: `SHOWLIST({'D' | 'L' | 'P' | 'R' | 'W'},`*[name]*`)`
The first argument is an option keyword to select a system list; the options currently supported are `Devices`, `Libraries`, `Ports`, `Ready`, and `Waiting`. If only the first parameter is supplied, the function scans the selected list and returns the node names in a string separated by blanks. If the *name* parameter is supplied, the boolean return indicates whether the specified list contains a node of that name. The name matching is case-sensitive.

The list is scanned with task switching forbidden so as to provide an accurate snapshot of the list at that time.
Example:

```
say showlist('P')               ==> REXX MyCon
say showlist('P','REXX')        ==> 1
```

STATEF()

Usage: `STATEF(`*filename*`)`
Returns a string containing information about an external file. The string is formatted as
"`{DIR | FILE}` *length blocks protection comment.*"
The *length* token gives the file length in bytes, and the *block* token specifies the file length in blocks.
Example:

```
say statef("libs:rexxsupport.library")
/* would give "FILE 1880 4 RWED  "      */
```

WAITPKT()

Usage: `WAITPKT(`*name*`)`
Waits for a message to be received at the specified (named) port, which must have been opened by a call to `OPENPORT()` within the current ARexx program. The returned boolean value indicates whether a message packet is available at the port. Normally the returned value will be 1 (**True**), since the function waits until an event occurs at the message port.

The packet must then be removed by a call to `GETPKT()`, and should be returned eventually using the `REPLY()` function. Any message packets received but not returned when an ARexx program exits are automatically `REPLY`ed with the return code set to 10.
Example:

```
call waitpkt 'MyPort'   /* wait awhile */
```

Appendix E

Distribution Files

This appendix lists the directories of the standard ARexx distribution disk. The contents of some of the directories may change from time to time, so your disk may not show exactly the same files. Most notably, the `:rexx` directory will expand as more program examples are included in it.

The second section of the Appendix lists the HEADER files that define the constants and data structures used with ARexx. All of these files are available in the :INCLUDE directory, but are listed here for convenience in studying the structures.

E-1 Directories

The files are listed below as they would be using the system `dir` command. For example, "dir df1:c opt a" would list the contents of the `:c` directory on disk drive 1.

The :C Directory

This directory contains the command utilities used with ARexx. These files should be copied to your system `C:` directory when you install the program.

```
c (dir)
   hi                          loadlib
   rexxmast                    rx
   rxc                         rxset
   tcc                         tco
   te                          ts
```

The :INCLUDE Directory

This directory has the INCLUDE and HEADER files used for assembly language and "C" programming, respectively. These files contain the structure definitions necessary to build an interface to ARexx.

```
include (dir)
   errors.h                    rexxio.h
   rxslib.h                    storage.h
   errors.i                    rexxio.i
   rxslib.i                    storage.i
```

The :LIBS Directory

These are the library files for the language interpreter and the Support Library functions. Both files should be copied to your system LIBS: directory when you install ARexx.

```
libs (dir)
  rexxsupport.library          rexxsyslib.library
```

The :REXX Directory

The :rexx directory contains example programs to illustrate various features of the language. New files will be added from time to time, and users are welcome to contribute files to be distributed in this way.

```
rexx (dir)
  bigif.rexx                   break.rexx
  builtin.rexx                 calc.rexx
  cmdtest.rexx                 fact.rexx
  factw.rexx                   haltme.rexx
  hosttest.rexx                iftest.rexx
  marquis.rexx                 nesttest.rexx
  paver.rexx                   potpourri.rexx
  rslib.rexx                   select.rexx
  sigtest.rexx                 support.rexx
  test1.rexx                   timer.rexx
```

The :TOOLS Directory

These files are intended for software developers, and include examples of interfacing to ARexx. The file rexxtest is of particular interest; it calls the ARexx interpreter directly, and can be run under a debugger to aid with developing new function libraries.

```
tools (dir)
  hosttest                     hosttest.asm
  loadlib.asm                  rexxtest
  rexxtest.asm                 rxoffsets.o
```

Miscellaneous Files

```
.info                          Install-ARexx
README                         Start-ARexx
```

E-2 Listings of Header Files

This section of the chapter consists of listings of the header files contained in the `:include` directory.

storage.h

This is the main header file and contains definitions for all of the important data structures used by ARexx.

```
/* === rexx/storage.h ==================================================
 *
 * Copyright (c) 1986, 1987 by William S. Hawes (All Rights Reserved)
 *
 * ====================================================================
 * Header file to define ARexx data structures.
 */

#ifndef REXX_STORAGE_H
#define REXX_STORAGE_H

#ifndef EXEC_TYPES_H
#include "exec/types.h"
#endif
#ifndef EXEC_NODES_H
#include "exec/nodes.h"
#endif
#ifndef EXEC_LISTS_H
#include "exec/lists.h"
#endif
#ifndef EXEC_PORTS_H
#include "exec/ports.h"
#endif
#ifndef EXEC_LIBRARIES_H
#include "exec/libraries.h"
#endif

/* The NexxStr structure is used to maintain the internal strings in REXX.
 * It includes the buffer area for the string and associated attributes.
 * This is actually a variable-length structure; it is allocated for a
 * specific length string, and the length is never modified thereafter
 * (since it's used for recycling).
 */
```

storage.h (cont.)

```
struct NexxStr {
   LONG     ns_Ivalue;              /* integer value                */
   UWORD    ns_Length;              /* length in bytes (excl null)  */
   UBYTE    ns_Flags;               /* attribute flags              */
   UBYTE    ns_Hash;                /* hash code                    */
   BYTE     ns_Buff[8];             /* buffer area for strings      */
   };                               /* size: 16 bytes (minimum)     */

#define NXADDLEN 9                  /* offset plus null byte        */
#define IVALUE(nsPtr) (nsPtr->ns_Ivalue)

/* String attribute flag bit definitions                           */
#define NSB_KEEP     0              /* permanent string?            */
#define NSB_STRING   1              /* string form valid?           */
#define NSB_NOTNUM   2              /* non-numeric?                 */
#define NSB_NUMBER   3              /* a valid number?              */
#define NSB_BINARY   4              /* integer value saved?         */
#define NSB_FLOAT    5              /* floating point format?       */
#define NSB_EXT      6              /* an external string?          */
#define NSB_SOURCE   7              /* part of the program source?  */

/* The flag form of the string attributes                          */
#define NSF_KEEP     (1 << NSB_KEEP  )
#define NSF_STRING   (1 << NSB_STRING)
#define NSF_NOTNUM   (1 << NSB_NOTNUM)
#define NSF_NUMBER   (1 << NSB_NUMBER)
#define NSF_BINARY   (1 << NSB_BINARY)
#define NSF_FLOAT    (1 << NSB_FLOAT )
#define NSF_EXT      (1 << NSB_EXT   )
#define NSF_SOURCE   (1 << NSB_SOURCE)

* Combinations of flags
#define NSF_INTNUM   (NSF_NUMBER | NSF_BINARY | NSF_STRING)
#define NSF_DPNUM    (NSF_NUMBER | NSF_FLOAT)
#define NSF_ALPHA    (NSF_NOTNUM | NSF_STRING)
#define NSF_OWNED    (NSF_SOURCE | NSF_EXT | NSF_KEEP)
#define KEEPSTR      (NSF_STRING | NSF_SOURCE | NSF_NOTNUM)
#define KEEPNUM      (NSF_STRING | NSF_SOURCE | NSF_NUMBER | NSF_BINARY)
```

storage.h (cont.)

```
/* The RexxArg structure is identical to the NexxStr structure, but
 * is allocated from system memory rather than from internal storage.
 * This structure is used for passing arguments to external programs.
 * It is usually passed as an "argstring", a pointer to the string buffer.
 */

struct RexxArg {
   LONG     ra_Size;                  /* total allocated length        */
   UWORD    ra_Length;                /* length of string              */
   UBYTE    ra_Flags;                 /* attribute flags               */
   UBYTE    ra_Hash;                  /* hash code                     */
   BYTE     ra_Buff[8];               /* buffer area                   */
   };                                 /* size: 16 bytes (minimum)      */

/* The RexxMsg structure is used for all communications with Rexx programs.
 * It is an EXEC message with a parameter block appended.
 */
struct RexxMsg {
   struct Message rm_Node;            /* EXEC message structure        */
   APTR     rm_TaskBlock;             /* pointer to global structure   */
   APTR     rm_LibBase;               /* library base                  */
   LONG     rm_Action;                /* command (action) code         */
   LONG     rm_Result1;               /* primary result (return code)  */
   LONG     rm_Result2;               /* secondary result              */
   STRPTR   rm_Args[16];              /* argument block (ARG0-ARG15)   */
   struct MsgPort *rm_PassPort;       /* forwarding port               */
   STRPTR   rm_CommAddr;              /* host address (port name)      */
   STRPTR   rm_FileExt;               /* file extension                */
   LONG     rm_Stdin;                 /* input stream (filehandle)     */
   LONG     rm_Stdout;                /* output stream (filehandle)    */
   LONG     rm_avail;                 /* future expansion              */
   };                                 /* size: 128 bytes               */
/* Field definitions                                                   */
#define ARG0(rmp) (rmp->rm_Args[0])   /* start of argblock             */
#define ARG1(rmp) (rmp->rm_Args[1])   /* first argument                */
#define ARG2(rmp) (rmp->rm_Args[2])   /* second argument               */

#define MAXRMARG  15                  /* maximum arguments             */

/* Command (action) codes for message packets                          */
#define RXCOMM    $01000000           /* a command-level invocation    */
#define RXFUNC    $02000000           /* a function call               */
#define RXCLOSE   $03000000           /* close the port                */
#define RXQUERY   $04000000           /* query for information         */
#define RXADDFH   $07000000           /* add a function host           */
```

```
#define RXADDLIB  $08000000           /* add a function library        */
#define RXREMLIB  $09000000           /* remove a function library     */
#define RXADDCON  $0A000000           /* add/update a ClipList string  */
#define RXREMCON  $0B000000           /* remove a ClipList string      */
#define RXTCOPN   $0C000000           /* open the trace console        */
#define RXTCCLS   $0D000000           /* close the trace console       */

/* Command modifier flag bits                                         */
#define RXFB_NOIO    16               /* suppress I/O inheritance?     */
#define RXFB_RESULT  17               /* result string expected?       */
#define RXFB_STRING  18               /* program is a "string file"?   */
#define RXFB_TOKEN   19               /* tokenize the command line?    */
#define RXFB_NONRET  20               /* a "no-return" message?        */

/* Modifier flags                                                     */
#define RXFF_RESULT  (1 << RXFB_RESULT)
#define RXFF_STRING  (1 << RXFB_STRING)
#define RXFF_TOKEN   (1 << RXFB_TOKEN )
#define RXFF_NONRET  (1 << RXFB_NONRET)
#define RXCODEMASK   $FF000000
#define RXARGMASK    $0000000F

/* The RexxRsrc structure is used to manage global resources.
 * The name string for each node is created as a RexxArg structure,
 * and the total size of the node is saved in the "rr_Size" field.
 * Functions are provided to allocate and release resource nodes.
 * If special deletion operations are required, an offset and base can
 * be provided in "rr_Func" and "rr_Base", respectively.  This function
 * will be called with the base in register A6 and the node in A0.
 */
struct RexxRsrc {
   struct Node rr_Node;
   WORD     rr_Func;                 /* "auto-delete" offset          */
   APTR     rr_Base;                 /* "auto-delete" base            */
   LONG     rr_Size;                 /* total size of node            */
   LONG     rr_Arg1;                 /* available ...                 */
   LONG     rr_Arg2;                 /* available ...                 */
   };                                /* size: 32 bytes                */
/* Resource node types                                                */
#define RRT_ANY      0                /* any node type ...             */
#define RRT_LIB      1                /* a function library            */
#define RRT_PORT     2                /* a public port                 */
#define RRT_FILE     3                /* a file IoBuff                 */
#define RRT_HOST     4                /* a function host               */
#define RRT_CLIP     5                /* a Clip List node              */
```

storage.h (cont.)

```
/* The RexxTask structure holds the fields used by REXX to communicate with
 * external processes, including the client task.  It includes the global
 * data structure (and the base environment).  The structure is passed to
 * the newly-created task in its "wake-up" message.
 */

#define GLOBALSZ  200                   /* total size of GlobalData      */
struct RexxTask {
   BYTE     rt_Global[GLOBALSZ];        /* global data structure         */
   struct MsgPort rt_MsgPort;           /* global message port           */
   UBYTE    rt_Flags;                   /* task flag bits                */
   BYTE     rt_SigBit;                  /* signal bit                    */

   APTR     rt_ClientID;                /* the client's task ID
   APTR     rt_MsgPkt;                  /* the packet being processed
   APTR     rt_TaskID;                  /* our task ID
   APTR     rt_RexxPort;                /* the REXX public port

   APTR     rt_ErrTrap;                 /* Error trap address
   APTR     rt_StackPtr;                /* stack pointer for traps

   struct List rt_Header1;              /* Environment list              */
   struct List rt_Header2;              /* Memory freelist               */
   struct List rt_Header3;              /* Memory allocation list        */
   struct List rt_Header4;              /* Files list                    */
   struct List rt_Header5;              /* Message Ports List            */
   };

/* Definitions for RexxTask flag bits                                     */
#define RTFB_TRACE   0                  /* external trace flag           */
#define RTFB_HALT    1                  /* external halt flag            */
#define RTFB_SUSP    2                  /* suspend task?                 */
#define RTFB_TCUSE   3                  /* trace console in use?         */
#define RTFB_WAIT    6                  /* waiting for reply?            */
#define RTFB_CLOSE   7                  /* task completed?               */

/* Definitions for memory allocation constants                            */
#define MEMQUANT  16                    /* quantum of memory space       */
#define MEMMASK   $FFFFFFF0             /* mask for rounding the size    */

#define MEMQUICK  (1 << 0 )             /* EXEC flags: MEMF_PUBLIC       */
#define MEMCLEAR  (1 << 16)             /* EXEC flags: MEMF_CLEAR        */
```

storage.h (cont.)

```
/* The SrcNode is a temporary structure used to hold values destined for a
 * segment array.  It is also used to maintain the memory freelist.
 */

struct SrcNode {
   struct SrcNode *sn_Succ;          /* next node                    */
   struct SrcNode *sn_Pred;
   APTR     sn_Ptr;                  /* pointer value                */
   LONG     sn_Size;                 /* size of object               */
   };                                /* size: 16 bytes               */
#endif
```

rxslib.h

This file defines the library base for the ARexx Systems Library.

```
/* === rexx/rxslib.h ======================================================
 *
 * Copyright (c) 1986, 1987 by William S. Hawes (All Rights Reserved)
 *
 * ========================================================================
 * The header file for the REXX Systems Library
 */

#ifndef REXX_RXSLIB_H
#define REXX_RXSLIB_H

#ifndef REXX_STORAGE_H
#include "rexx/storage.h"
#endif

/* Some macro definitions                                                 */

#define RXSNAME  "rexxsyslib.library"
#define RXSID    "rexxsyslib 1.0 (23 AUG 87)"
#define RXSDIR   "REXX"
#define RXSTNAME "ARexx"

/* The REXX systems library structure.  This should be considered as  */
/* semi-private and read-only, except for documented exceptions.      */

struct RxsLib {
   struct Library rl_Node;            /* EXEC library node           */
   UBYTE    rl_Flags;                 /* global flags                */
   UBYTE    rl_pad;
   APTR     rl_SysBase;               /* EXEC library base           */
   APTR     rl_DOSBase;               /* DOS library base            */
   APTR     rl_IeeeDPBase;            /* IEEE DP math library base   */
   LONG     rl_SegList;               /* library seglist             */
   LONG     rl_MaxAlloc;              /* maximum memory allocation   */
   LONG     rl_Chunk;                 /* allocation quantum          */
   LONG     rl_MaxNest;               /* maximum expression nesting  */
   struct NexxStr *rl_NULL;           /* static string: NULL         */
   struct NexxStr *rl_FALSE;          /* static string: FALSE        */
   struct NexxStr *rl_TRUE;           /* static string: TRUE         */
   struct NexxStr *rl_REXX;           /* static string: REXX         */
   struct NexxStr *rl_COMMAND;        /* static string: COMMAND
   struct NexxStr *rl_STDIN;          /* static string: STDIN
   struct NexxStr *rl_STDOUT;         /* static string: STDOUT
   struct NexxStr *rl_STDERR;         /* static string: STDERR
```

```
    STRPTR    rl_Version;           /* version/configuration string*/
    STRPTR    rl_TaskName;          /* name string for tasks       */
    LONG      rl_TaskPri;           /* starting priority           */
    LONG      rl_TaskSeg;           /* startup seglist             */
    LONG      rl_StackSize;         /* stack size                  */
    STRPTR    rl_RexxDir;           /* REXX directory              */
    STRPTR    rl_CTABLE;            /* character attribute table   */
    struct NexxStr *rl_Notice;      /* copyright notice            */

    struct MsgPort rl_RexxPort;     /* REXX public port            */
    UWORD     rl_ReadLock;          /* lock count                  */
    LONG      rl_TraceFH;           /* global trace console        */
    struct List rl_TaskList;        /* REXX task list              */
    WORD      rl_NumTask;           /* task count                  */
    struct List rl_LibList;         /* Library List header         */
    WORD      rl_NumLib;            /* library count               */
    struct List rl_ClipList;        /* ClipList header             */
    WORD      rl_NumClip;           /* clip node count             */
    struct List rl_MsgList;         /* pending messages            */
    WORD      rl_NumMsg;            /* pending count               */
    };

/* Global flag bit definitions for RexxMaster                      */
#define RLFB_TRACE RTFB_TRACE       /* interactive tracing?        */
#define RLFB_HALT  RTFB_HALT        /* halt execution?             */
#define RLFB_SUSP  RTFB_SUSP        /* suspend execution?          */
#define RLFB_TCUSE RTFB_TCUSE       /* trace console in use?       */
#define RLFB_TCOPN 4                /* trace console open?         */
#define RLFB_STOP  6                /* deny further invocations    */
#define RLFB_CLOSE 7                /* close the master            */

#define RLFMASK    0x07             /* passed flags                */

        ; Initialization constants

#define RXSVERS    2                /* main version                */
#define RXSREV     1                /* revision                    */
#define RXSALLOC   0x800000         /* maximum allocation          */
#define RXSCHUNK   1024             /* allocation quantum          */
#define RXSNEST    32               /* expression nesting limit    */
#define RXSTPRI    0                /* task priority               */
#define RXSSTACK   4096             /* stack size                  */
#define RXSLISTH   4                /* number of list headers      */
```

rxslib.h (cont.)

```
/* Character attribute flag bits used in REXX.  Defined only for      */
/* ASCII characters (range 0-127).                                    */

#define CTB_SPACE   0                 /* white space characters       */
#define CTB_DIGIT   1                 /* decimal digits 0-9           */
#define CTB_ALPHA   2                 /* alphabetic characters        */
#define CTB_REXXSYM 3                 /* REXX symbol characters       */
#define CTB_REXXOPR 4                 /* REXX operator characters     */
#define CTB_REXXSPC 5                 /* REXX special symbols         */
#define CTB_UPPER   6                 /* UPPERCASE alphabetic         */
#define CTB_LOWER   7                 /* lowercase alphabetic         */

/* Attribute flags                                                    */
#define CTF_SPACE   (1 << CTB_SPACE)
#define CTF_DIGIT   (1 << CTB_DIGIT)
#define CTF_ALPHA   (1 << CTB_ALPHA)
#define CTF_REXXSYM (1 << CTB_REXXSYM)
#define CTF_REXXOPR (1 << CTB_REXXOPR)
#define CTF_REXXSPC (1 << CTB_REXXSPC)
#define CTF_UPPER   (1 << CTB_UPPER)
#define CTF_LOWER   (1 << CTB_LOWER)

#endif
```

rexxio.h

This file defines the data structures used for buffered I/O. ARexx uses linked lists of **IoBuff** structures to keep track of the files it opens. Each **IoBuff** node is allocated as an "auto-delete" structure and can be closed and released by a call to either CloseF() or RemRsrcNode(). An entire list of files can be closed with a call to RemRsrcList().

```
/* === rexx/rexxio.h ==================================================
 *
 * Copyright (c) 1986, 1987 by William S. Hawes (All Rights Reserved)
 *
 * ====================================================================
 * Header file for ARexx Input/Output related structures
 */

#ifndef REXX_REXXIO_H
#define REXX_REXXIO_H

#ifndef REXX_STORAGE_H
#include "rexx/storage.h"
#endif

#define RXBUFFSZ  204                   /* buffer length                */

/* The IoBuff is a resource node used to maintain the File List.  Nodes are
 * allocated and linked into the list whenever a file is opened.
 */

struct IoBuff {
   struct RexxRsrc iobNode;             /* structure for files/strings  */
   APTR     iobRpt;                     /* read/write pointer           */
   LONG     iobRct;                     /* character count              */
   LONG     iobDFH;                     /* DOS filehandle               */
   APTR     iobLock;                    /* DOS lock                     */
   LONG     iobBct;                     /* buffer length                */
   BYTE     iobArea[RXBUFFSZ];          /* buffer area                  */
   };                                   /* size: 256 bytes              */

/* Access mode definitions                                             */
#define RXIO_EXIST   -1                 /* an external filehandle       */
#define RXIO_STRF    0                  /* a "string file"              */
#define RXIO_READ    1                  /* read-only access             */
#define RXIO_WRITE   2                  /* write mode                   */
#define RXIO_APPEND  3                  /* append mode (existing file)  */
```

rexxio.h (cont.)

```
/* Offset anchors for SeekF()                                          */
#define RXIO_BEGIN   -1              /* relative to start              */
#define RXIO_CURR    0               /* relative to current position   */
#define RXIO_END     1               /* relative to end                */

/* The Library List contains just plain resource nodes.               */

#define LLOFFSET(rrp) (rrp->rr_Arg1)   /* "Query" offset               */
#define LLVERS(rrp)   (rrp->rr_Arg2)   /* library version              */

/* The RexxClipNode structure is used to maintain the Clip List.  The
 * value string is stored as an argstring in the rr_Arg1 field.
 */

#define CLVALUE(rrp) ((STRPTR) rrp->rr_Arg1)

/* A message port structure, maintained as a resource node.
 * The ReplyList holds packets that have been received but haven't been
 * replied.
 */

struct RexxMsgPort {
   struct RexxRsrc rmp_Node;         /* linkage node                  */
   struct MsgPort  rmp_Port;         /* the message port              */
   struct List     rmp_ReplyList;    /* messages awaiting reply       */
   };

/* DOS Device types                                                   */
#define DT_DEV    0                  /* a device                      */
#define DT_DIR    1                  /* an ASSIGNed directory         */
#define DT_VOL    2                  /* a volume                      */

/* Private DOS packet types                                           */
#define ACTION_STACK 2002            /* stack a line                  */
#define ACTION_QUEUE 2003            /* queue a line                  */
#endif
```

errors.h

This file contains the definitions for all of the error messages issued by the ARexx interpreter.

```
/* == errors.h ==========================================================
 *
 * Copyright (c) 1987 by William S. Hawes (All Rights Reserved)
 *
 * ======================================================================
 * Definitions for ARexx error codes
 */

#define ERRC_MSG  0                     /*  error code offset           */
#define ERR10_001 (ERRC_MSG+1)          /*  program not found           */
#define ERR10_002 (ERRC_MSG+2)          /*  execution halted            */
#define ERR10_003 (ERRC_MSG+3)          /*  no memory available         */
#define ERR10_004 (ERRC_MSG+4)          /*  invalid character in program*/
#define ERR10_005 (ERRC_MSG+5)          /*  unmatched quote             */
#define ERR10_006 (ERRC_MSG+6)          /*  unterminated comment        */
#define ERR10_007 (ERRC_MSG+7)          /*  clause too long             */
#define ERR10_008 (ERRC_MSG+8)          /*  unrecognized token          */
#define ERR10_009 (ERRC_MSG+9)          /*  symbol or string too long   */

#define ERR10_010 (ERRC_MSG+10)         /*  invalid message packet      */
#define ERR10_011 (ERRC_MSG+11)         /*  command string error        */
#define ERR10_012 (ERRC_MSG+12)         /*  error return from function  */
#define ERR10_013 (ERRC_MSG+13)         /*  host environment not found  */
#define ERR10_014 (ERRC_MSG+14)         /*  required library not found  */
#define ERR10_015 (ERRC_MSG+15)         /*  function not found          */
#define ERR10_016 (ERRC_MSG+16)         /*  no return value             */
#define ERR10_017 (ERRC_MSG+17)         /*  wrong number of arguments   */
#define ERR10_018 (ERRC_MSG+18)         /*  invalid argument to function*/
#define ERR10_019 (ERRC_MSG+19)         /*  invalid PROCEDURE           */

#define ERR10_020 (ERRC_MSG+20)         /*  unexpected THEN/ELSE        */
#define ERR10_021 (ERRC_MSG+21)         /*  unexpected WHEN/OTHERWISE   */
#define ERR10_022 (ERRC_MSG+22)         /*  unexpected LEAVE or ITERATE */
#define ERR10_023 (ERRC_MSG+23)         /*  invalid statement in SELECT */
#define ERR10_024 (ERRC_MSG+24)         /*  missing THEN clauses        */
#define ERR10_025 (ERRC_MSG+25)         /*  missing OTHERWISE           */
#define ERR10_026 (ERRC_MSG+26)         /*  missing or unexpected END   */
#define ERR10_027 (ERRC_MSG+27)         /*  symbol mismatch on END      */
#define ERR10_028 (ERRC_MSG+28)         /*  invalid DO syntax           */
#define ERR10_029 (ERRC_MSG+29)         /*  incomplete DO/IF/SELECT     */
```

errors.h (cont.)

```
#define ERR10_030 (ERRC_MSG+30)        /*  label not found             */
#define ERR10_031 (ERRC_MSG+31)        /*  symbol expected             */
#define ERR10_032 (ERRC_MSG+32)        /*  string or symbol expected   */
#define ERR10_033 (ERRC_MSG+33)        /*  invalid sub-keyword         */
#define ERR10_034 (ERRC_MSG+34)        /*  required keyword missing    */
#define ERR10_035 (ERRC_MSG+35)        /*  extraneous characters       */
#define ERR10_036 (ERRC_MSG+36)        /*  sub-keyword conflict        */
#define ERR10_037 (ERRC_MSG+37)        /*  invalid template            */
#define ERR10_038 (ERRC_MSG+38)        /*  invalid TRACE request       */
#define ERR10_039 (ERRC_MSG+39)        /*  uninitialized variable      */

#define ERR10_040 (ERRC_MSG+40)        /*  invalid variable name       */
#define ERR10_041 (ERRC_MSG+41)        /*  invalid expression          */
#define ERR10_042 (ERRC_MSG+42)        /*  unbalanced parentheses      */
#define ERR10_043 (ERRC_MSG+43)        /*  nesting level exceeded      */
#define ERR10_044 (ERRC_MSG+44)        /*  invalid expression result   */
#define ERR10_045 (ERRC_MSG+45)        /*  expression required         */
#define ERR10_046 (ERRC_MSG+46)        /*  boolean value not 0 or 1    */
#define ERR10_047 (ERRC_MSG+47)        /*  arithmetic conversion error */
#define ERR10_048 (ERRC_MSG+48)        /*  invalid operand             */

/* Return Codes for general use ...                                   */
#define RC_FAIL   -1                   /*  something's wrong           */
#define RC_OK      0                   /*  success                     */
#define RC_WARN    5                   /*  warning only                */
#define RC_ERROR  10                   /*  something's wrong           */
#define RC_FATAL  20                   /*  complete or severe failure  */
```

Glossary

Allocation. A grant of a system resource, such as memory space. Programs designed to run in a multitasking environment generally use dynamic allocation to avoid tying up system resources.

AmigaDOS. The higher-level part of the Amiga operating system that supports the filing system and input/output operations.

Argstring. An "argument string" structure used to pass data to an ARexx program. The structure is passed as a pointer to the buffer area containing the string data, and can be treated as a pointer to a null-terminated string.

Argument. A data item passed to a function, sometimes called a parameter.

Clause. A group of one or more tokens forming a "sentence" in a language. The clause is the smallest executable language fragment.

Command Line Interface (CLI). A program that accepts input from the user and runs programs based on the entered command. The CLI generally refers to the command interpreter supplied with the Amiga, but other command "shells" may be used instead.

Concatenation. An operation in which two strings are joined or "chained together." ARexx provides two concatenation operators, one of which joins strings directly and the other of which embeds a blank between the operands.

EXEC. The multitasking kernel of the Amiga's operating system. EXEC provides the task scheduling, interrupt handling, and message-passing primitives used to support ARexx.

Function Host. A program that manages a public message port for receiving function invocation messages. The message port may be the same one used for command messages.

Function Library. A collection of functions callable from ARexx and managed as an Amiga shared library. Each function library includes an entry point to associate a function name with the code to be called.

Host Address. The name of the public message port associated with a host application. The host address is used as the unique identifier for the host, and should be unique within the system message ports list. Within an ARexx program the host address identifies the external host to which commands will be sent.

Host Application. An executable program that provides a suitable command interface to receive ARexx commands. Most host applications will also provide a means to invokde macro programs from within the application.

Interrupt. An event that alters the normal flow of control in a program. Interrupts in ARexx refer to events within the program execution and are distinct from the hardware-level interrupts managed by the Amiga EXEC system.

Macro Program. A program that implements a complex "macro" operation from a series of "micro" commands.

Message Packet. A data structure used to pass information between tasks. A message packet is allocated and initialized by one task and then sent to another task's message port. After the recipient has processed the message, it "replies" the message to the *replyport* associated with the message.

Message Port. A data structure used as the rendezvous point for message passing. A message port provides the anchor for a list of message packets and identifies the task to be signalled when a message arrives.

Multitasking. The ability to run more than one program at a time. More precisely, multitasking permits the resources of the computer to be shared among many tasks without forcing any task to be aware of the others.

Process. An extension to an EXEC task structure that provides the data fields required to use AmigaDOS functions. All ARexx programs run as AmigaDOS processes.

Replyport. A message port designated to receive a returning message packet. Each message packet includes a field that specifies its reply port.

Resident Process. The program responsible for launching ARexx programs and for managing various resources used by ARexx. It is structured as a host application and opens a public message port named "REXX."

Shared Library. A collection of executable code and data managed as a resource by the EXEC operating system. As the name "shared" implies, the code and data in a library can be used by more than one task.

Storage Environment. The collection of data values forming the current state of an ARexx program. Storage environments are strictly nested and only one environment is current at any time.

Task. An entity consisting of executable code and a data structure managed by the EXEC operating system. The task is the smallest program unit that can be scheduled and run separately.

Token. The elementary words or atoms of a language. A token can be considered as a string of one or more characters forming the smallest unit of the language.

Typeless. Data items having no assumed structure or usage. ARexx treats all data as typeless character strings and checks for specific characteristics only when required by an operation.

Index

Suggested List Price: $49.95 US

ARexx Version 1.0

ARexx is a multitasking implementation of the REXX language, an elegant high-level language designed for macro-processing and general programming tasks. Its clean, simple syntax makes REXX easy to learn ... an ideal "first language." And the powerful language features will appeal to experienced programmers as well!

- Interpreted operation — no compile-link-run steps
- Exceptional String-Handling Facilities
- Built-In Source-Level Debugger
- Over 75 Euilt-In Functions
- Supports External Function Libraries
- Compact, Reentrant Code — Only 32K

ARexx macro programs can interact with other software products that include an ARexx command interface, allowing you to extend and customize your software and to build integrated applications. The growing list of software products supporting the ARexx command interface includes TxEd-Plus from MicroSmiths, C.A.P.E.68K from Inovatronics, and AmigaTEX from Radical Eye Software.

Look also for WShell, our companion Amiga software product. WShell is a CLI-compatible command shell with the features you've always wanted in a command environment — command aliases, resident and built-in commands, prompt string/window titlebar variables, concurrent piping, and many more. With its ARexx command interface, WShell provides transparent support for REXX macros as well as the standard "execute" scripts (including support for the new "script bit".)

Developed and supported by:
William S. Hawes
P.O. Box 308
Maynard, MA 01754

System requirements: Amiga 500/1000/2000 with V1.2 OS