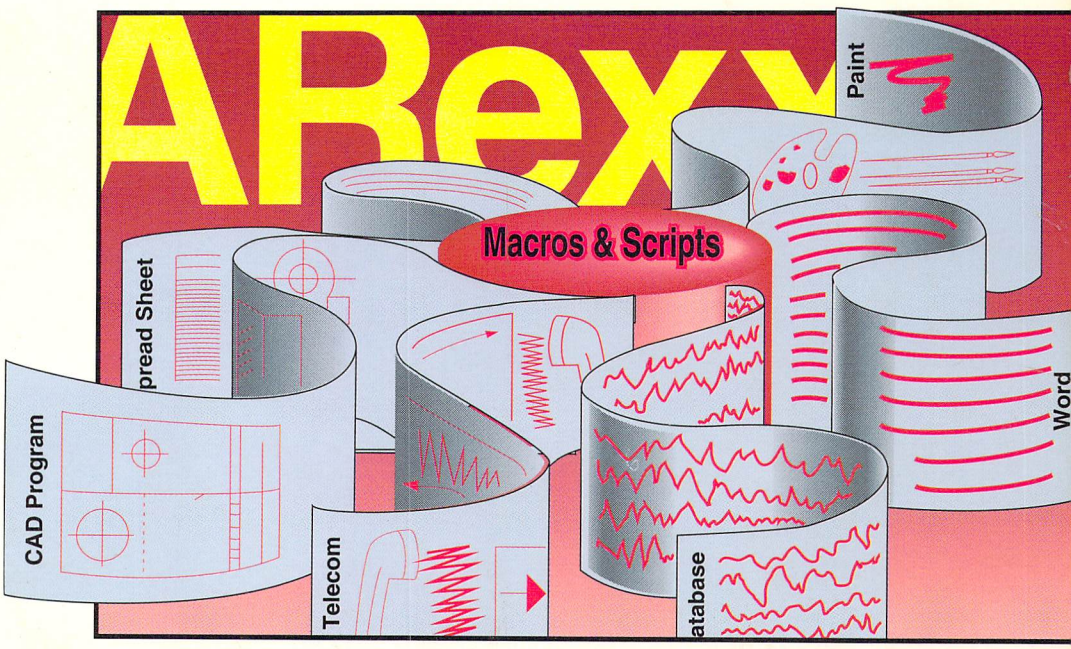
Includes Companion Diskette

Using ARexx on the Amiga®

Guide to using the ARexx programming language

by Zamara and Sullivan

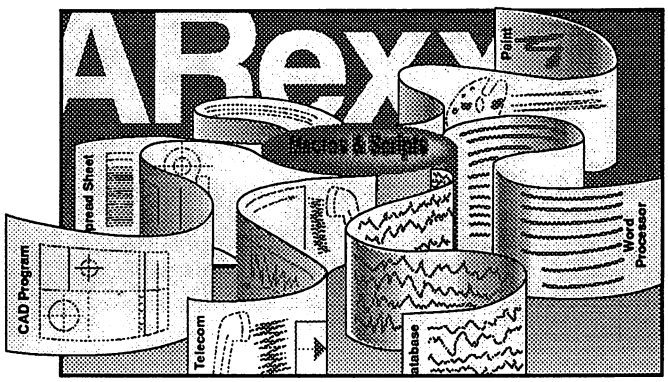


Includes ready-to-use companion diskette

Abacus

Using ARexx on the Amiga

by Chris Zamara and Nick Sullivan



Published by
Abacus

Abacus
5370 52nd Street SE
Grand Rapids, MI 49512

Chris Zamara and Nick Sullivan

Editors: Jim D'Haem, Scott Slaughter, Robbin Markley

Zamara Chris, 1963-
Arexx programming on the Amiga / Chris Zamara, Nick Sullivan.
p. cm.
Includes index.
ISBN 1-55755-114-6 : $34.95
1. Amiga (Computer)--Programming. 2. ARexx (Computer program language) I. Sullivan, Nick. 1951- . II. Title.
QA76.8A177Z36 1991
005.265--dc20 91-30049
CIP

Printed in U.S.A.
10 9 8 7 6 5 4 3 2

Preface

Although this book may seem to be a specialized text about a single programming language, it is for all Amiga users who want to get more from their computer. This is because ARexx is more than just a programming language - it's now a standard part of the Amiga's operating system, and is used by a large number of software packages to add powerful user-programmability. ARexx is not just for programmers: it's for 'power users' and anyone with serious applications for their Amiga: graphics, animation, word processing, multimedia, spreadsheets, 3D modeling, and more.

You don't have to be an Amiga expert to use this book: it starts off assuming very little about how much background you may have with programming, using computers, and using the Amiga in particular. You will find it helpful, however, to have some acquaintance with the Amiga's basic operations. If you're a beginner and aren't familiar with the basics of the Workbench, copying files, and typing in simple Shell commands, we recommend that you look through the documentation supplied with your Amiga. You'll need to know these basics before using most programs or doing any serious work with the computer.

You can use ARexx and this book even if you don't intend to learn about writing your own ARexx programs. You will get the most out of ARexx and many of your software applications, however, if you give ARexx programming a try - ARexx is much easier to use than other languages. If you are completely new to programming, you can still learn ARexx programming from this book, although it will take some determination. If you've never encountered programming in any language before, we would recommend that you read a 'learning programming' textbook as support material.

This book is very much a 'hands on' text. Although you can learn the concepts of ARexx just by reading it, the ideas will be much more firmly cemented in place if you actually try the given examples and perform your own experiments. For this reason, it is best if you are sitting near your computer when reading the book to learn about ARexx for the first time. Once you understand the basics and start writing your own ARexx programs, you'll want to keep the book nearby for the complete easy-to-access reference at the end.

The terminology used throughout the book is consistent with that used in the official ARexx manual by William S. Hawes. In some

cases, this differs from the terminology used in the *The REXX Language: A Practical Approach to Programming* by M. F. Cowlishaw, which defines the REXX language on which ARexx is based. The most notable difference is in the use of the term *instruction*, which Cowlishaw defines as any clause other than null clauses and label clauses (including assignments and external host commands), while Hawes (and this book) uses "instruction" to mean what Cowlishaw calls a *keyword instruction*.

We hope that learning and using ARexx is an enjoyable experience. We have tried to use interesting examples wherever possible, and have included real-world programs that you'll actually find useful. If you have fun while you're discovering and learning about ARexx along with this book, you'll learn a lot more, and learn it faster. It is our hope that this book accomplishes that goal.

Acknowledgements

Though this book, like all others ever written, is in some sense a conspiracy of thousands with the authors merely the ringleaders, we would like to single out for thanks the special contributions of some of our chief accomplices.

We begin with Bill Hawes, to whom, as the author of ARexx, this book owes the very possibility of its existence. Bill's creative adaptation of the IBM script language REXX to the multitasking Amiga environment was one of the decisive actions that shaped Amiga history. On a more immediate level, Bill's generous and patient assistance with technical questions was invaluable to us throughout the writing of this book.

Of the several people whose ideas influenced the final form of both the book and the accompanying disk, we would like to mention particularly Mark R. Brown of INFO magazine and long-time Amiga activist Larry Phillips. Will Murphy, Brad DuTemple and the rest of the staff at Moebius Computers in Victoria were unfailingly helpful in answering questions and helping us locate needed materials.

In the preparation and proof-reading of the manuscript, we would like to acknowledge the help of Corwin Sullivan for his careful verification of the programming examples, and of our editors at Abacus, led by Jim D'Haem, for imposing typographical order on the sometimes chaotic text files we thrust upon them.

Jim Oldfield at Abacus deserves special mention both for helping to initiate the project (in what now seems the distant past) and for his superb efficiency in the role of liaison with the software manufacturers whose products are discussed in Section III of the book.

We would also like to express our gratitude to the manufacturers themselves for their co-operation in providing source materials for Section III. Though they are too numerous to name individually here, we direct the reader's attention to Appendix D (Vendors and Products) for a listing with complete address information.

Finally, we would like to thank the community of Amiga users for their growing acceptance and support of ARexx as a standard script and macro language, and Commodore-Amiga Inc. both for recognizing the importance of ARexx and for the Amiga computer itself.

Nick Sullivan - Chris Zamara

September 1991, Victoria, BC

Contents

SECTION I
GETTING ACQUAINTED

Chapter 1
Introduction

What is ARexx?

ARexx (pronounced "AY-REX") is a simple programming language for the Amiga that anyone can learn. ARexx lets you write simple "scripts" (programs) that define a task for the computer to perform.

A script is just a text file containing a list of instructions that ARexx understands. These scripts, which you can either write yourself or obtain from other sources, can let you accomplish many everyday tasks automatically that could involve hours of laborious manual operation of software applications.

Because of the way ARexx can communicate with other programs running at the same time in the computer (thanks to the Amiga's multitasking operating system), you can use **macros** written in ARexx that will work with specific software applications.

Learning about ARexx will not only give you the ability to create programs to solve specific problems, but can make your wordprocessor, paint program, spreadsheet, 3D modelling software and many other applications much more powerful and automatic.

You can even use ARexx programs to link the operations of several programs to perform tasks that no one program can accomplish by itself.

ARexx itself is a program, but unlike other programs that may display windows, menus and gadgets on the screen, ARexx is "invisible".

When you run the ARexx "master" program, ARexx sits in the background, ready to run any scripts that may come its way. You may tell ARexx to run scripts directly or you may run them as macros from software applications that support ARexx.

In this book, you'll learn how to write programs in ARexx and how to use ARexx with some of the more popular software applications that support it.

You'll also be provided with some useful ready-to-use macros that can add extra features to software you may already own —even if you don't do any ARexx programming at all.

History of ARexx

REXX

ARexx is so named because it is an implementation of the REXX language for the Amiga. REXX is a language definition developed over a number of years by Mike Cowlishaw at IBM.

Cowlishaw began in 1979 with the idea to create a language that was easy to program in and was "designed for people, not machines". The language evolved as the newest versions were distributed over IBM's massive VNET network consisting of over 1,000 mainframe installations.

VNET users tested and used the latest versions of the language, sending suggestions and comments to Cowlishaw by electronic mail. The amount of testing and refinement that went on was staggering.

In Cowlishaw's book, *The REXX Language,* he reports that at peak periods he received over 350 pieces of electronic mail per day! An informal language committee spontaneously appeared, communicating over the network and discussing the language's evolution.

It was largely this kind of large-scale cooperation and user input that shaped the language, guided along all the while by the author's fundamental concepts.

Fundamental REXX concepts

From the start, REXX was to be easy to program in, highly readable, use natural data typing and be effective in manipulating the kinds of symbols people use most often, like words, names and numbers.

The language also had to be good at reporting errors accurately and, perhaps most of all, had to try to fulfill the users' expectations as much as possible: any behavior that might be surprising to the programmer was carefully weeded out.

On the system level, the language was designed to be system-independent, working on a variety of platforms and to be easily adaptable to future expansion. It was also designed with support for communicating with external "host environments", so that REXX

could be used as a standard macro language for operating systems, editors and other applications.

Cowlishaw calls this important feature "a primary concept of the language". (You'll see why when you begin working with ARexx macros!)

REXX implementations

As the language developed and gained many users, it became more and more officially recognized. In 1983, the IBM System/370 implementation became part of CMS (Conversational Monitor System), a user interface for IBM mainframes.

In 1985, a version of REXX was released for MS-DOS/PC-DOS personal computers. In 1987, REXX became the standard Procedures Language for all IBM Systems Application Architecture (SAA) operating systems, which encompasses a large number of systems.

The language is still evolving, but more slowly than before: the standard is too firmly entrenched for major changes to be made at this point. Discussions are underway, however, for exciting new versions like an object-oriented REXX.

The official REXX language definition is completely documented in the book by M. F. Cowlishaw, *The REXX Language: A Practical Approach to Programming* (Prentice-Hall, 1985; 2nd edition, 1990). This book is also the source for the above historical information about REXX.

REXX for the Amiga

When the Amiga came out in 1985, it was and remained for many years the most advanced personal computer available. Not only did it have incredible graphics and sound capabilities, but it had true multitasking, allowing applications to effortlessly share the machine without any special consideration by the programmer.

With a true multitasking computer, the REXX language clearly defined and the recent release of REXX on other personal computers, the stage was set for an Amiga version of REXX.

ARexx was created by William Hawes and released in 1987. ARexx followed the official REXX language definition very closely, but added a new dimension to ARexx's external 'host environment' commands.

Since the Amiga supported true multitasking, any other program running in the system could theoretically be used as a 'host' for these external commands. This made ARexx the ideal macro language, since it could control any program that had ARexx support built in. This

might include text editors, spreadsheets or any other application that might have its own specialized language.

William Hawes developed the entire ARexx system, wrote the manual and distributed and supported the product. The official ARexx package has always been available from William Hawes himself as well as the usual retail outlets.

Like REXX before it, ARexx steadily built a loyal following of users, becoming the standard macro language on the Amiga.

ARexx and AmigaDOS release 2

As it became evident that ARexx was not only the *de facto* standard on the Amiga, but also a very good standard indeed, Commodore made the wise decision of adopting ARexx as an official part of the operating system with the release of the 2.0 version.

ARexx is not an integral part of the system – it's not in the computer's ROM (Read-Only Memory) and the standard Shell does not recognize ARexx scripts especially – but the ARexx software is included on the standard system Workbench disk.

When you boot an Amiga under Workbench 2.0, ARexx is ready to use. Now that ARexx is officially part of the system and software developers can be assured that most users will have it, we can expect to see more and more applications with built-in ARexx support.

ARexx and the Amiga

In retrospect, it is easy to see that ARexx is the ideal macro language for the Amiga and the inclusion of ARexx with the latest release of the operating system makes perfect sense. But how did this wide acceptance come about?

The adoption of any standard—especially one involving a third-party commercial product—is extremely difficult. Unless a lot of software supports ARexx, the language won't be very popular; if not many people buy the language, there isn't much incentive for software developers to add ARexx support to their products.

Looking at the Catch-22 situation, it seems almost miraculous that ARexx has succeeded and become the widely accepted and supported standard that it has. It was not just luck that brought this about, however.

ARexx didn't have a big advertising budget or hype behind it. It was introduced quietly, at first being known to software developers rather than the general user community. The design and implementation of ARexx, not its promotion, is what contributed most to its success. Some of these factors were:

The REXX language standard

The REXX language standard is powerful yet easy to program in. This makes it ideal for the kind of 'quick and dirty' scripts that are put together to solve specific problems.

Small demands on memory

ARexx was implemented using very efficient code, making the language quite small. The entire system, once loaded into the Amiga, takes up less than 40 kilobytes of RAM.

For comparison, consider that Commodore's *AmigaVision* program, shipped with the Amiga 3000 computer, is over 600k! ARexx's small size helped greatly in making the language accepted, since the system has to be available at all times and every extra kilobyte of memory that it uses is one kilobyte less available for other programs to run in. Perhaps the fact that the original ARexx release was developed on a non-expanded (512k) Amiga had something to do with this preoccupation with small code size!

Shared library for multitasking

The implementation of ARexx as a "shared library" means that more than one ARexx program can run at the same time without loading more than one copy of the ARexx system into memory. Again, this keeps memory demands on the system very low.

Good behavior for high reliability

ARexx has always been very "well-behaved", making it trusted by those that use it. The most important aspect of this good behavior is that the program doesn't 'crash' the computer (cause a software error and subsequent reboot) under any circumstances.

This is a very important quality for any software to have and is something that isn't reflected in written specifications. When people have confidence in running a certain piece of software and never worry about the system crashing as a result of it, they tend to use it more often.

Other aspects of ARexx's good behavior are its friendliness towards multitasking (it doesn't slow down other programs when it's not being used) and its careful return of all the system memory that it uses.

Expandability

ARexx itself is small, but is designed in a flexible way so that its functionality can be easily increased. Through its support of *function libraries* and *function hosts*, the ARexx language can be extended to any degree. This makes the basic language system a good platform for more powerful language versions in the future.

The future vision

Perhaps the greatest factor of all in ARexx's success was the inspiration that many software developers found by thinking "What if... ?". What if all software supported the ARexx standard fully? What if all users had ARexx and knew how to use it?

If people could fully automate the control of any software and link the functionality of different programs into one, wouldn't that put the Amiga far ahead of any other personal computer? Isn't this what the Amiga—and the world—really needs? This is the kind of exciting talk that went on as the developers contemplated the future of ARexx and the Amiga, and many of them decided to make it happen.

Although not many users owned ARexx in the beginning, many developers supported it in their products as an investment in this future. Partly because of their faith and vision—and now Commodore's—this future is coming true.

The missing link: introductory documentation

What has been holding ARexx back? Why isn't it even more generally used and accepted? Probably the lack of information and good introductory documentation had a lot to do with it.

As good as the ARexx software was, the documentation, however complete, was never very easy to understand or learn from for the average user. Fortunately, this problem is being corrected, partly due to books such as the one you're reading now.

What is a Programming Language?

A programming language lets you define a task for the computer to perform—a program—in a way that makes sense and is easy to understand from a human's point of view.

Using a programming language to create a computer program is much simpler than if you had to use the the primitive 'machine' instructions that the computer's hardware understands. There are several programming languages that can be used on the Amiga: AmigaBASIC, other versions of BASIC and AmigaCOMAL for simple programming and languages like C, Modula-2 and Assembler (which

is very close to 'machine' language) for more complicated or high-performance applications.

With any programming language that you use, the list of commands and instructions that you create (called *source code*) must be converted by an *interpreter* or a *compiler* program into the more basic machine instructions (the *object code*) before the computer can run your program.

The wordprocessor or paint program that you use, for example, may have been originally written in C or assembly language. The source code for the program was converted into machine instructions by a compiler (or assembler, in the case of assembly language), which results in the program that resides on your disk drive.

ARexx's specialty: external control

ARexx is a true programming language, but it differs from other programming languages like BASIC in a number of important ways.

Languages like BASIC are self-contained: The program that you write must do everything by itself. If you want your program to allow the user to edit text, for example, you have to write the code for a text-editing facility. ARexx, on the other hand, has the unique ability to communicate directly with other programs running at the same time.

In ARexx, the same problem might be handled by using special instructions that are understood by the wordprocessor program that is currently being used.

Interpreter vs. compiler

We mentioned the *interpreter* and *compiler* above. Although it may seem like a rather technical detail, it is useful to know that ARexx is an interpreter. This means that each line of source code in a program is interpreted by ARexx one at a time as the program is executed.

A compiler, on the other hand, translates the entire program all at once and the translated version of the program is used thereafter. Interpreters are generally easier to use than compilers and make it easier to find and correct errors in your program. Interpreters also have language features not available in compilers (like ARexx's powerful INTERPRET instruction).

The drawback is mostly in performance: programs written in interpreted languages like ARexx cannot execute their instructions as quickly as compiled programs. ARexx can be used to solve any programming problem, but real-time applications like video games or high-speed data sampling are not good candidates for ARexx programs.

What Can ARexx Be Used For?

ARexx programs can be used in a number of ways:

Stand-alone scripts

An ARexx program or script, can be used in many everyday applications just as a BASIC program or an AmigaDOS script (such as the "Startup-Sequence") might be used. ARexx scripts, however, are often shorter and simpler to write than programs written in these other languages.

ARexx has powerful features for dealing with textual information, making it ideal for applications that involve reading and processing data from text files. Once you become familiar with ARexx programming, you'll often be able to type in a small script to suit your needs exactly instead of adapting your problem to a spreadsheet, database or other pre-existing software.

Program macros

Since ARexx can control many software applications, ARexx scripts can be used as "macros" to perform specific operations within an application. The ability to control the functions of an application in this way using a general purpose language like ARexx allows you to add powerful, custom made features to a program.

Using ARexx to create macros also has the advantage of standardization: you use the same language to write macros in a number of different programs.

Programs that allow ARexx macros can also be easily extended even if you don't write the macros yourself. By obtaining macros from other sources, you can increase the power of your existing software applications without purchasing an upgraded version.

Controlling multiple programs

Some complicated processes may involve the use of several programs: You might render images with a 3D modelling program, then put together many such images into an animation using an animation package, for example.

A procedure like this may involve a number of separate steps with both software packages that would have to be repeated for each frame of the animation. ARexx's ability to control external programs allows you to write a program that would coordinate operations between the modelling and animation programs (both of which would be running at the same time).

Theoretically, any number of programs could participate in an operation like this (providing your computer has enough memory to run them all), all controlled by a single ARexx script.

Built-in Instructions and External Commands

Because ARexx is a self-contained programming language and has the ability to control external programs, ARexx programs can contain two kinds of instructions:

ARexx instructions and functions

Built-in *instructions* and *functions* can be used in any ARexx script, since they are part of the basic ARexx language. ARexx programs using only these built-in instructions and functions can be run regardless of what software is operating; no external programs are affected.

External software commands

External *commands* are understood by the current "command host", which is the external program ARexx is currently communicating with (it might be a spreadsheet program, for example). These commands are interpreted not by ARexx itself, but by the external program.

It is these commands that let you control the functions of a software application using an ARexx program instead of manually selecting menus, entering data, etc. These commands are often unique to a particular software package, so ARexx scripts that use them are not generic: such scripts are generally designed to work with a specific software application.

Scripts and macros

Stand-alone scripts can be created using only the built-in instructions and functions. Scripts that are used as program macros or for controlling multiple programs will combine the use of built-in instructions and functions with software-specific commands.

Macros are often supplied with software applications that support ARexx; additional macros can often be purchased as an add-on product. In many cases, macros are available for free from the public domain. Once you learn to program in ARexx, however, you'll probably be happiest using the macros that suit your needs exactly: those that you've written yourself.

A Few Examples

The idea of 'adding power' to your software might sound tantalizing, but is admittedly vague. This book—and the accompanying disk—provide you with some samples of ARexx scripts and macros that can be used with popular software products. Here are a few examples of how ARexx can be used with some commercially available products:

Database

The *Microfiche Filer Plus* (Software Visions) database program can be fully controlled via ARexx macros. Your database can be fully automated by writing ARexx programs, just as you would write a program for a language-driven database like Ashton Tate's *DBase III.*

This gives you the ability to generate special reports from the information in the database, to create a large number of records with data from a text file or generated by the program, to perform selective search and replace operations on a large number of records, etc.

Unlike databases with their own control programs, using ARexx in this way means you don't have to learn a new language to use the program: once you know ARexx, you just have to learn the commands specific to the database and you're ready to write database programs.

Spreadsheet

The *Advantage* spreadsheet (Gold Disk) program provides ARexx macros to search and replace formulas, automatically lay out large tables, swap any two spreadsheet columns and automatically fill in a range of cells with the names of months.

Because *Advantage* has full ARexx support, many other operations on a spreadsheet that might be tedious can be automated by writing simple scripts and using them as macros.

Hypermedia

In the hypermedia program *HyperBook* (Gold Disk), you can lay out pages containing text or graphical objects. If you were creating a hyperbook with dozens or hundreds of pages, each using different text and formatting (colors, position, etc.), it could take quite a while to do by hand.

Since *HyperBook* has full ARexx support, a simple ARexx macro could read text from a file and create each page with the text formatted accordingly. Once the script was written (a few minutes' work), the entire process would all take place automatically.

An example of a hyperbook created in this way, along with the HyperBook 'reader' program to let you look at it, is included on the companion disk.

Paint program

ARexx scripts can be used to control the paint program *DigiPaint 3* (NewTek). Any common graphical effect that you can accomplish by a series of steps can be programmed in a script.

You might have a script to scale the current 'brush' to a specific size and overlay it onto the picture using a specific density and shading style. Rather than select all of these operations each time, you could let a script do the work.

This is especially useful for repetitive operations: your work of art might involve stamping down a hundred such brushes in a grid pattern across the picture. Very tedious and inaccurate by hand, the same procedure is easy, quick and accurate using an ARexx script.

Telecom-munications

Baud Bandit (Progressive Peripherals and Software) and *A-TALK III* (Oxxi) are examples of telecommunications programs that are completely controllable with ARexx macros. Using macros with these programs can allow you to automate a procedure like logging into an online service, downloading your mail and logging off.

On a larger scale, an ARexx script can be used to create a customized bulletin board system so that remote users can log onto your system and exchange information and files.

CAD Programs

With an ARexx-controllable CAD (Computer-Aided Design) program like *IntroCAD Plus* from Progressive Peripherals and Software, you can use ARexx macros to define complex figures mathematically that would take hours of painstaking work to create manually.

How This Book Is Organized

This book is designed as your complete guide to using ARexx on the Amiga. You will be introduced to setting up and using ARexx, generic ARexx programming and specific ARexx applications with popular software products.

The book is divided into four main sections:

Section I - Getting Acquainted

This contains all introductory material, including how to install and set up the ARexx system. It also dives in with some simple examples of ARexx programs just to give you a feel for the language and "get your feet wet".

More experienced users or those that know a bit about ARexx can skim over this section, but most users should find it extremely beneficial for getting acquainted.

Section II - ARexx Programming

This is where you learn how to program in ARexx. No previous programming experience is assumed and all examples use purely generic ARexx programming: no other software is required.

Basic rules of syntax and program construction are given in the early chapters, with more specific uses of ARexx instructions and functions given in the later ones.

If you're not interested in writing ARexx programs, but merely wish to learn about what the language is and how you can use the provided macros and scripts with your software, you can skip this entire section.

We would recommend you at least give ARexx programming a try, however: it may not be as difficult as you think!

Section III - Controlling Applications

While Section II covered the basic ARexx instructions and functions, in this section we cover ARexx as it is used in a number of popular software products. You'll see examples of scripts and macros and learn how to use them in each of the programs.

You can type the macros in from the book to really 'get your hands dirty,' or simply load them from the supplied disk. Command summaries of the main software packages are also listed for easy reference.

This section is useful to owners of the software products mentioned, as well as to owners of similar products or those considering purchasing an ARexx-compatible product.

Section IV - Reference

Section IV is the reference section, which you'll probably find to be the most useful part of the book after you've read the first three sections.

If you do any ARexx programming, you'll continually find yourself reaching to this section to look up the various built-in instructions and functions, what they do and how to use them.

The instruction/function reference portion of the section is organized in two parts, so that you can browse through similar functions by category or quickly look up any instruction or function alphabetically.

The section is placed at the end of the book so that you can easily thumb through it.

Chapter 2
The ARexx System

Fundamentals of the ARexx System

Although it is a powerful, general-purpose programming language, the ARexx system actually consists of just a few relatively small programs. Here is a brief overview of the disk files that are absolutely vital to using ARexx:

RexxMast: The resident process

This program must be run before ARexx can be used. Once *RexxMast* is running in the system, ARexx scripts can be run using the 'RX' program and ARexx macros can be used from within software applications.

rexxsyslib.library: The ARexx systems library

This file contains most of the software required by ARexx and must be present in the system 'libs:' directory at the time *RexxMast* is run.

The RX program

The small 'RX' program is used as an AmigaDOS Shell command to run ARexx scripts (your ARexx programs), which are stored as text files. For example, the command 'rx hello' would execute the ARexx script on disk called 'hello'.

Alternatively, you can create a Workbench icon for a script that uses RX to run the script when the icon is double-clicked. RX simply passes your ARexx script to the ARexx resident process, which you started by running *RexxMast*.

RX can also be used to directly execute short ARexx programs that can fit on a single line. You'll learn more about using this command from a Shell or CLI window in this section of the book.

ARexx scripts and macros

RexxMast, *RX* and *rexxsyslib.library* are part of the ARexx system itself. Since ARexx is a programming language, it needs programs to do anything useful; these ARexx programs (called *scripts* and *macros*) are provided by you.

You can write your own ARexx scripts and macros to perform specific tasks—after reading this book, you should have a good idea how to do this.

You can also use prewritten scripts and macros written by someone else to help with common operations.

Installing ARexx

ARexx itself is a software program that remains resident in your system, springing into action when needed. While some programs consist of just a single disk file that you can double-click from the Workbench or run from an AmigaDOS Shell window, the ARexx system consists of a few files that work together.

These files must be on your standard boot disk (the Workbench disk you start off with, also referred to as the 'system' or 'SYS:' disk) if you want the ARexx system to be available every time you reboot.

If you have purchased the ARexx package and haven't 'installed' these files yet, this section will guide you through the simple process. Even if you have ARexx already set up in your system, the guide below may answer some of your questions about exactly what the installation process consists of.

AmigaDOS 2.0 If you have the major Amiga operating system upgrade, V2.0, running on your Amiga, then ARexx is already a part of your system.

The *RX* program and other ARexx programs are included on the standard Workbench disk under V2.0 (in the 'Rexxc' directory) and the 'RexxMast' program in the 'System' drawer is automatically run by the Startup-Sequence when the system boots up. Additional ARexx support files are in the 'libs' directory.

The remainder of this *Installation* section assumes that you have purchased the ARexx package separately and you have the standard ARexx release disk.

ARexx versions

The original ARexx release was called 'Version 1.0'—the version number should be printed right on the disk label. Since then, version 1.1 has been released, which contains a number of improvements, including better performance and some new functions.

Although for the most part there is little difference, this book assumes the version 1.1 release. If you purchased ARexx version 1.0, you should be able to bring the original disk to your dealer for an upgrade to 1.1.

Installing using the AmigaDOS Shell

The easiest way to run programs and copy files is by using the Workbench GUI (Graphical User Interface) familiar to most Amiga users.

If you prefer the command-driven 'Shell' window (or the old-style 'CLI' Command-Line Interface), you can use the *Copy* command to transfer the basic set of required files (commands and libraries) from the ARexx disk to your system disk:

```
Shell> copy ARexx_Disk:c c:
Shell> copy ARexx_Disk:libs libs:
```

You can now start the ARexx *resident process* to make the ARexx language available:

```
Shell> RexxMast
```

There are also some additional library files in the 'libraries' directory that can be installed optionally (copied to 'libs:'). These are explained in the discussion of the 'Install-ARexx' script below.

Installing from Workbench

If you have looked at the ARexx release disk from Workbench (by double-clicking the disk icon), you have probably already found that running the system and installing it on your boot disk is simply a matter of double-clicking the 'Install-ARexx' icon.

This icon first executes an AmigaDOS script (a file called 'Start-ARexx' consisting of a list of AmigaDOS commands) that runs ARexx from the release disk, making the ARexx language available in the system.

It then executes an ARexx script (called 'Install-ARexx') using the 'RX' program, since ARexx is now operating. 'Installation' in this case simply means copying a few files from the ARexx disk to various directories on your standard 'SYS:' disk (the Workbench disk or hard

drive partition that you boot from). The basic ARexx system requires about 40k of free space on your boot disk.

The 'Install-ARexx' script

As stated above, the ARexx installation script simply copies a few files. However, it's not quite that simple: you'll have to answer a few questions to control the level of installation that is performed. Here's a breakdown of exactly what the script does and how to answer the questions:

- Copies the **commands** in the 'C' directory on the ARexx disk to the 'C:' system directory, where commands are normally kept. A complete list of these commands can be found in *Chapter 2 - The Entire System*, below.

- Copies the **library files** in the ARexx disk 'libs' directory to the system 'libs:' directory. These are the basic files for the ARexx system, consisting of *rexxsyslib.library* and *rexxsupport.library*.

- Displays a message in the console window asking, "Install optional libraries in LIBS:? (Y/N)". You must respond with 'Y' (followed by RETURN) if you want the files copied or 'N' otherwise.

  These **optional libraries** add extra functions to ARexx, allowing you to write advanced programs more easily. They provide graphics, math and other capabilities not built into the ARexx core language.

  If you can spare an additional 60k or so on your boot disk and you think you might want to write complex scripts using advanced math functions or Intuition operating system calls, then respond with 'Y'.

  You may also come across scripts that require these libraries; some of the ARexx scripts supplied with *IntroCAD Plus*, for example, require the math library. If you don't have the space on your boot disk, don't worry about it. None of the examples in this book will require these libraries and you can write many powerful ARexx programs without them.

- Displays another message asking, "Install icons in SYS:? (Y/N)". If you respond with 'Y', the programs 'RexxMast' (for starting ARexx) and 'RXC' (for removing ARexx) will be copied to the 'root' of your boot disk, along with their '.info' icon files.

  This will allow you to start ARexx—and remove it, if desired—by double-clicking these icons from Workbench. If you want this convenience, respond with 'Y'. It is not required, however, since you can start (or remove) ARexx with a simple 'RexxMast' Shell command or have *RexxMast* run automatically at boot time.

Using ARexx without installing the files

If your Workbench disk is so full that there's not enough room even for the basic 40k ARexx installation or you want to use ARexx occasionally but don't want to commit valuable Workbench disk space to its files, you can use the system without doing the installation.

Simply running the 'RexxMast' program won't work, since it will expect to find the ARexx system library in the SYS disk's 'libs' directory (or wherever 'libs:' is assigned). Fortunately, you can preload the system library so that RexxMast will work. This can be done from the Workbench or through a Shell command:

From Workbench: Preloading the library and running RexxMast can be done from the Workbench like this:

- Double-click the 'LoadLib' icon on the ARexx disk
- Double-click the 'RexxMast' icon on the ARexx disk.

The minimal ARexx system will now be available for use. Any scripts using functions in the extra 'support' library will not work, however. This won't be a problem for most scripts and for most of the examples found in this book.

From the Shell: Executing the 'Start-ARexx' AmigaDOS script on the ARexx disk will give you access to the full ARexx system, including the support library, without copying any files to your system disk. From a shell prompt, just type the following command:

```
Shell> execute ARexx_Disk:Start-ARexx
```

Starting ARexx after rebooting

The installation process described above—using either the Workbench or Shell method—needs only to be done once for a given boot disk.

The installation as described also makes certain that ARexx is set up and running, ready to use. The next time you reboot, however, ARexx will not be running although the necessary files are available on the system disk. That is because *RexxMast*, the resident process, has not been started.

To start ARexx, just run the RexxMast program, either by typing 'RexxMast' at a Shell console window or by double-clicking the 'RexxMast' icon. If you chose the optional 'Install icons' procedure from the installation script, the icon will be in the root of your system boot disk.

Permanent installation

If you want ARexx to run automatically for you whenever you reboot, you can execute the 'RexxMast' command in the system 'Startup-Sequence'. The Startup-Sequence is an AmigaDOS script file that contains a list of AmigaDOS commands executed when the system is initialized (after a reboot).

Running the *RexxMast* command in the startup-sequence is not a necessary step, but is a good idea. It will save you the trouble of always remembering to run *RexxMast* before you use an ARexx-driven program or execute an ARexx script.

Don't worry that running *RexxMast* every time will make unnecessary demands on your system's memory or other resources. ARexx is very economical and unless you are running *very* close to the upper limit of your system's available memory, having ARexx running will not have any noticeable effect on system performance or capacity.

Adding *RexxMast* to the Startup-Sequence

To modify your Startup-Sequence, you will need to load it into a text editor, add the *RexxMast* command, then save the file. When you do this, make sure you are not working with the only copy of your system's Workbench disk!

You should always work from backup copies of this disk and have the original stored away safely in unmodified form.

Information about using text editors in general can be found in Appendix A. To describe this procedure, we will assume the use of *Ed*, the simple editor provided in the 'C' directory of the standard Workbench disk. This editor is not the best, but for this simple task it will do well enough.

You've probably used some sort of text editor before, perhaps in the form of a wordprocessor. In any case, following the instructions below should result in a successfully modified Startup-Sequence:

- You will need to use the AmigaDOS Shell (or the CLI, if you have a version of the operating system earlier than 1.3). If no Shell window is running, double-click the 'Shell' icon on the Workbench disk.

- At the Shell prompt (when the Shell window is active), run *Ed* and load the Startup-Sequence with the following command:

  ```
  Shell> ed s:Startup-Sequence
  ```

  After a bit of disk activity, *Ed*'s window will open and the text from the Startup-Sequence file will be displayed.

- A good place to put in the *RexxMast* command is right near the end of the file, so that this command will be executed as one of the last items in the system initialization procedure. Move to the end of the file by using the *Ed* 'bottom' command 'ESC b': press the 'Esc' key (at the top left of the keyboard), then press 'b' and hit RETURN. The cursor will move to the end of the file.

- The last line in the Startup-Sequence is normally 'endcli >nil:'. This final command removes the initial Shell window, exposing the Workbench.

If you've already modified your Startup-Sequence and removed this command, just put the 'RexxMast' on the very last line. If the 'endcli' command is still there, we'll put the RexxMast command on the line just before it, as follows:

- Hold down the SHIFT key and press cursor-left; the cursor will move to the start of the 'endcli' line. Now press RETURN to create a blank line for the new command and move the cursor up into the blank line. Type 'rexxmast >nil:' without pressing RETURN after it. The last two lines in the file should now look similar to this:

```
rexxmast >nil:
endcli >nil:
```

(The '>nil:' after the command tells the Amiga to 'throw away' all output from these commands, so that the display is not cluttered by unnecessary text.)

- Save the file and close down *Ed* with the 'exit' command 'ESC x': press the 'Esc' key, then press 'x' and RETURN. The file will be saved back to the system disk and *Ed* will exit, closing its window. You will be returned to the Shell window.

The REXX: assign

One other step that can be performed for a complete ARexx installation is to assign the logical device 'REXX:' to the directory containing your ARexx scripts.

When you execute an ARexx script using the *RX* program, it first looks in your *current directory* for the script, then looks in 'REXX:', if the assignment has been made. If this doesn't make much sense to you right now, don't worry. It's not vital to using ARexx and will be covered in more detail in upcoming chapters.

If you understand the AmigaDOS Shell and the *Assign* command, however, you may want to make the REXX assignment in your Startup-Sequence right after the RexxMast command. Assuming you will be putting ARexx scripts in the 'S:' directory (which is supposed to be for scripts, after all), you could add the command 'assign rexx: s:' to the Startup-Sequence.

Hard disks and the Startup-Sequence

If you are using a hard disk and your Startup-Sequence is similar to 'Startup-Sequence.hd' from an old Workbench version 1.2 disk, you should edit the Startup-Sequence on the floppy disk instead of 's:Startup-Sequence'.

The 'Startup-Sequence.hd' file on these early Workbench versions did not execute the Startup-Sequence on the hard drive. If there is no Startup-Sequence in 's:' (*Ed* shows a blank window and says 'creating new file') or adding the RexxMast command has no effect, you are probably in this situation. With the Workbench boot disk in drive DF0:, use the command 'ed df0:s/Startup-Sequence' instead of 'ed s:Startup-Sequence' as shown above.

The Entire System

You don't have to know all the details about the various ARexx files to use the language, but it is good to have a general idea of the various components involved in the whole system.

If you are making a customized boot disk for your system, for example, you might want to know the minimum number of files that you require for your specific needs.

Programs

The *RexxMast* and *RX* programs are the most important ones in the ARexx system, but they are not the only ones. Here are the other programs that are part of the ARexx system. You may never need to use many of these. These commands are explained in detail in Appendix B of this book, but here's a brief overview:

RexxMast	The ARexx *resident process;* makes ARexx available.
RX	Run a script or give an ARexx command.
HI	Halt all active scripts immediately.
RXC	Shut down and remove ARexx from the system.

RXLIB	Add an ARexx function library or show library list.
RXSET	Set a global 'clip' variable.
TS	Start trace mode.
TE	End trace mode.
TCO	Open 'global tracing console'.
TCC	Close global tracing console.
WaitForPort	Wait for message at given host address.
LoadLib	Preloads Amiga libraries for ARexx (ARexx disk only).

The above programs are found in the 'C' directory on the ARexx release disk.

On the Workbench 2.0 disk, all the above commands except for *RexxMast* and *LoadLib* are in the 'Rexxc' directory. *RexxMast* is in the 'System' drawer, where it can be run from Workbench by double-clicking its icon. This is usually unnecessary with V2.0, since the 'RexxMast' command is executed automatically in the Startup-Sequence. *LoadLib* is not included in the 2.0 release.

Libraries

There are two 'library' files use by ARexx, which are found in the 'libs' directory of the ARexx release disk or the Workbench 2.0 disk:

libs/rexxsyslib.library	The main ARexx system
libs/rexxsupport.library	Some extra ARexx functions

The *rexxsyslib.library* file is required before ARexx can be used. It contains most of the basic ARexx software that is responsible for running basic scripts.

The *rexxsupport* library is used by some scripts, but is not vital to simply get ARexx started. It is recommended that you have both of these libraries available in the system's 'libs:' directory; the installation process will put them there.

Another library that ARexx uses is *mathieeedoubbas.library*. This library is part of the Amiga operating system and is in the 'libs' directory of the standard Workbench release disk.

This file must be in your system 'libs:' directory in order for ARexx to work. If you've removed this file from your Workbench disk because you thought you didn't need it, you'll have to put it back again before you can use ARexx.

Optional libraries

Other libraries may be used to extend ARexx's available functions. Such libraries will often be loaded by the scripts that use them and you will usually know if a script needs a special ARexx library.

A few of these optional libraries are supplied on the ARexx release disk in the 'libraries' directory. The *rexxarplib* provides a number of useful functions to ARexx scripts, including a file requester and access to the Amiga's Intuition operating system routines.

The *rexxmathlib* provides a number of common math functions, including trigonometric and exponential functions, which are not available in the basic ARexx package. The installation script on the ARexx disk will ask whether you want these optional libraries copied to the 'libs:' directory of your boot disk.

If you have enough room and you think you might want to use these features in future ARexx scripts, say 'Y' to copy them (see the above *Installation* section).

Read the documentation files provided with these libraries to learn how to use the functions. (These libraries are not part of the ARexx language but extensions written by an ARexx user; they are not documented in this book.)

Using optional libraries

Some ARexx programs may make use of the optional math or ARP libraries and will expect them to be available in the system 'libs:' directory. Before a library can be used, it must be *added* to ARexx's list of libraries.

In many cases, an ARexx program that needs an optional library will add the library itself (assuming it is in the 'libs:' directory). In some

cases, however, an ARexx program might assume that a library like 'rexxmathlib.library' has already been added.

This will usually be noted in the documentation for the ARexx program or the application that uses it, such as the ARexx macros provided with *IntroCAD* from Progressive Peripherals and Software.

If you are using a program that assumes the presence of the math library, you must first add the library using ARexx's RXLIB program.

This can be done with the following Shell command, assuming the optional 'rexxmathlib.library' has been installed in your 'libs:' directory.

```
Shell> rxlib rexxmathlib.library 0 -30
```

Similarly, if an ARexx program assumes the presence of the 'rexxsupport.library,' you can add the library like this:

```
Shell> rxlib rexxsupport.library 0 -30
```

The RXLIB program is covered in detail, along with the other ARexx programs, in Appendix B of this book.

How ARexx Is Accessed

Now you know what ARexx consists of and how to make it ready to go every time you turn on your computer. You might be wondering at this point how you actually use the ARexx system in sessions at your computer.

In a way, that's what this entire book is about. However, this section talks about the fundamental three-way interaction among ARexx, the system and you.

The resident process

All ARexx programs are executed by the resident process, the ARexx program that is always running in the computer. When you run the *RexxMast* program (or it is automatically run by the Startup-Sequence), the resident process is started.

Unless the resident process is explicitly removed (using the *RXC* program), it will remain resident and ready to go until the computer is turned off or rebooted.

Creating an ARexx script

An ARexx script is a text file containing ARexx instructions; you'll learn all about writing ARexx scripts in Section II of the book.

You can create a script using a text editor such as *Ed* (supplied in the C directory of the standard Workbench disk) or *MEmacs* (supplied on the system 'Extras' disk). There are many other text editors that can be used to do the job; see Appendix A for more information about text editors.

The text file is saved to disk (or RAM:), where it can be read by the *RX* program and passed on to the ARexx resident process to be executed.

Executing a script using the Shell

An easy way to execute an ARexx script is to use the *RX* program to run the program from an AmigaDOS Shell. If you don't have a Shell window up, you must first double-click the 'Shell' icon on the Workbench disk. (If you have a version of AmigaDOS earlier than 1.3, use the 'CLI' icon instead. CLI stands for 'Command Line Interface' and is a simpler precursor to the more convenient Shell supplied with versions 1.3 and later.)

The Shell window that comes up can be used for running any ARexx scripts, and also for using AmigaDOS commands and any other programs. You can keep a Shell window open while running other software; there is usually no need to close the window.

You can also open another Shell window to type commands into if the first one is 'busy' waiting for a program to complete.

An ARexx script can be executed with a simple Shell command. Unlike ordinary programs which can be executed by simply typing the program's name, ARexx scripts must be executed using ARexx's 'RX' program. For example, if your ARexx program was saved as a file called 'myscript,' (in the Shell's current directory) you would run the program like this:

```
Shell> rx myscript
```

Scripts can be written so that they use additional information supplied on the command line. A command using such a script might look like this:

```
Shell> rx myscript 10 Charlie -12.4
```

In the above example, the 'myscript' script was run and was supplied with the information '10', 'Charlie' and '-12.4'. What this information is used for depends on the script.

Since you don't know how to write an ARexx script yet, the above example doesn't do you much good! This is just an introduction. In Chapter 4, the process of running a script from the Shell is explained in detail, using real examples.

Executing a script using the Workbench

The RX program can also be used to run an ARexx script from the Workbench, so that the script will appear as an icon and can be run by double-clicking, just like the icon of a regular program. This is done by simply saving a 'Project' icon along with the script and changing the icon's 'default tool' to 'RX'.

To do this, you will need to use a text editor that will save an icon along with the text file. In most text editors, this feature can be turned on as an option. Once the file is saved, the icon's imagery can be modified using the Icon Editor tool from the system 'Extras' disk.

Before you can use the icon to run the script, you must change the icon's 'default tool' to run the 'RX' program. Changing an icon's default tool is a standard Workbench procedure, explained in the Workbench section of your Amiga manual. The basic steps to change the default tool to 'RX' are as follows:

- Select the script's icon by clicking on it once.
- Select *Info* from the *Workbench* menu (with Workbench Version 2.0, it is *Information* in the *Icons* menu). The icon information window appears.
- Click on the 'default tool' box and type 'C:RX', assuming you've installed RX in your 'C:' directory as in the normal installation procedure. If you are using Workbench 2.0, just type 'RX'.
- Click 'Save' on the bottom left of the requester to make the change to the icon.

From now on, double-clicking on the icon will automatically run the associated script as an ARexx program.

Everyday use of scripts

So running ARexx scripts is straightforward, but when and why are they normally used? Once you learn ARexx, you'll probably write quick scripts to solve a number of problems that come up all the time. You might have solved these problems in the past by doing a lot of manual work or trying to use a program that wasn't quite suitable for the task. Or you may have not done the job at all, perhaps doing the work manually or getting someone else to do it.

This book is filled with examples of useful ARexx scripts, but a sneak peek at a single example might be useful at this point:

If you use a modem and telecommunications software, you have probably made use of Bulletin Boards or other online information services and have seen the large amount of software available to download. Suppose you've just completed an exceptionally long online session and you've downloaded dozens of files. Most of these files are 'archives,' containing a number of files compressed and unified using an archiving program like ARC, ZOO, LHARC or others. For each of the files you've downloaded, you have to create a directory and extract the files in the archive using the appropriate utility. This could take a long time, especially since the extraction process itself can be very slow. The more files you've downloaded and the bigger they are, the longer it will take.

The alternative is to write an ARexx script that will do all the work automatically, creating the directories and calling up the appropriate archive utilities accordingly. Not only is the automated process less work for you, it can go on in the background while you do something else, so that even if it takes hours, they won't be hours of *your time.*

This is a random example, simply chosen because the problem was a real one that happened to come up as the author was preparing this section of the book; the script to do the job is included on the companion disk.

Using ARexx from within an application

You can make full use of ARexx even if you never use the Shell or Workbench. A software application like a spreadsheet, database or wordprocessor can use ARexx programs called *macros* to define useful

operations made up of a number of basic operations available in the program. Applications that support ARexx communicate directly with the ARexx resident process in the same way as the RX program does, and usually provide specialized commands to the ARexx language.

With most programs that support ARexx, you are able to load macros from disk and execute macros from within the application itself. You may even be able to create new macros and edit existing ones. This means you never have to type a Shell command or use icons on the Workbench to use ARexx. Many applications will even try to run the *RexxMast* program if it hasn't been run yet, freeing you from one more concern.

With sophisticated ARexx-supporting software applications, it is possible for you to make heavy use of ARexx without even knowing that you're doing so. Using pre-written macros means ARexx is just functioning as an 'engine' beneath the user interface of the software application. If you *do* know about the application's use of ARexx, however and you know a bit about ARexx programming, you will be able to control the application in new and powerful ways.

Chapter 3
Simple Software Control

Simple ARexx Commands

In the previous chapter, you saw how to run an ARexx script using the RX program in a Shell window. You can also use RX to give a single line of ARexx instructions directly, without running a separate script.

For example, type the following command at the Shell prompt (open a Shell window if necessary). When typing it in, note the use of the double-quote character around the entire command and the single-quote character around the text inside:

```
Shell> rx "say 'Hello, this is ARexx speaking.'"
```

The text "Hello, this is ARexx speaking" should be printed out in the console window. You have just used ARexx's SAY instruction to print some text.

Talking to AmigaDOS

Now try the following ARexx command:

```
Shell> rx "address command 'Dir'"
```

You should see the list of files in your current directory, as if you had just typed the AmigaDOS 'Dir' command by itself. What has just happened is that you have given an AmigaDOS command from within an ARexx program.

Not only can ARexx programs communicate with software that has ARexx support built-in, they can also be used to run AmigaDOS commands or any other program that might be available on the currently mounted disk volumes.

The 'address command' instruction as it is used above tells ARexx to pass the following text on the line—in this case 'Dir'—to the AmigaDOS environment, to be executed as if it were a command typed by the user at a Shell window.

ARexx didn't do anything with the 'Dir' command itself; it just 'passed the buck'. The ability to pass the buck like this is what gives ARexx its usefulness as a macro language for controlling external software. The

special case of controlling AmigaDOS is useful in that it lets you use ARexx scripts as a substitute for AmigaDOS scripts.

If the 'Dir' command was used without first giving the instruction 'ADDRESS COMMAND', ARexx would have looked for an ARexx script called 'Dir' or 'Dir.rexx' and executed that script as an ARexx program. The ADDRESS COMMAND instruction diverted the command from ARexx to AmigaDOS.

The ADDRESS instruction

Using the ADDRESS instruction tells ARexx the name of a *host* program to send commands to. When a host has been specified, any time you use a *command* in an ARexx script (a symbol that isn't part of the built-in set of ARexx instructions), the command is sent to the current 'host' application for interpretation.

A host can be any program with an ARexx 'command interface,' and each host is identified by its own unique name (called the *host address*). Using the ADDRESS instruction with a particular host address will set that host as the destination for all future commands.

The COMMAND host name used above—as in 'ADDRESS COMMAND'—is a special case used for AmigaDOS commands. To send commands from ARexx to an external host application, you must know its host name.

For example, NewTek's *DigiPaint 3* has an ARexx command interface and its host name happens to be the same as the name of the program. (The host name for an application will be given in the application's documentation, usually in the section on the ARexx interface.)

To send commands to *DigiPaint 3* from ARexx, you would need to first use the following ARexx instruction:

```
address 'DigiPaint'
```

Any future commands not recognized as valid ARexx instructions would be sent to *DigiPaint 3* for interpretation.

Like many applications that support ARexx, *DigiPaint 3* has commands that allow control of all the same features of the program that can be controlled manually through the menus and control panels. All the commands are documented in the *DigiPaint 3* manual.

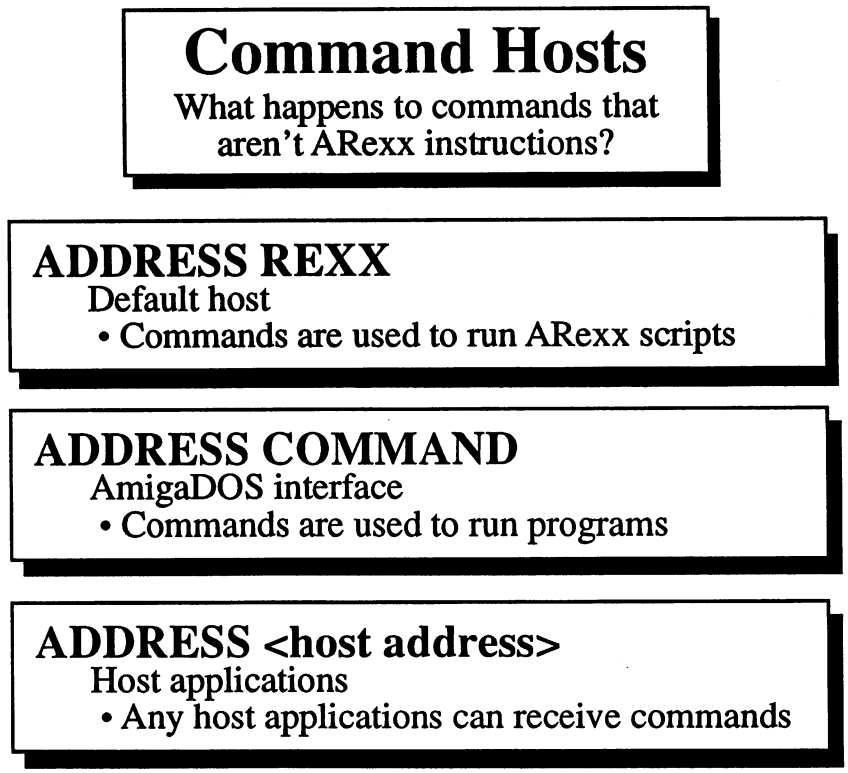


Figure 3-1: Command Hosts

Controlling a Sample Host

The best way to get a feel for how you can control a host application using ARexx is to try a few simple examples. On the companion disk, we've provided a simplified paint program with an ARexx command interface.

This program, called "ARexxPaint," is not meant as a serious application program for real work. However, it makes a good example of an ARexx host.

The *ARexxPaint* host application

Run the paint program from the disk by double-clicking its icon from Workbench or by typing the following command from the Shell:

```
Shell> run UsingARexx:ARexxPaint
```

ARexxPaint opens its window on the Workbench screen instead of using a custom screen; this makes it easier for you to type Shell commands and see the program's display at the same time.

Notice the simple capabilities of the program: using the menus you can draw lines, squares, circles, filled shapes and change colors. Experiment with the program a bit to see its limited set of capabilities. Using the program in this way is not really important, however, since you'll soon be controlling it with commands from ARexx.

Commanding the host

Arrange the Shell Window so that it fits below the *ARexxPaint* window. Activate the Shell window and type the following command:

```
Shell> rx "address 'ARexxPaint' 'BOX' 30 20 200 70"
```

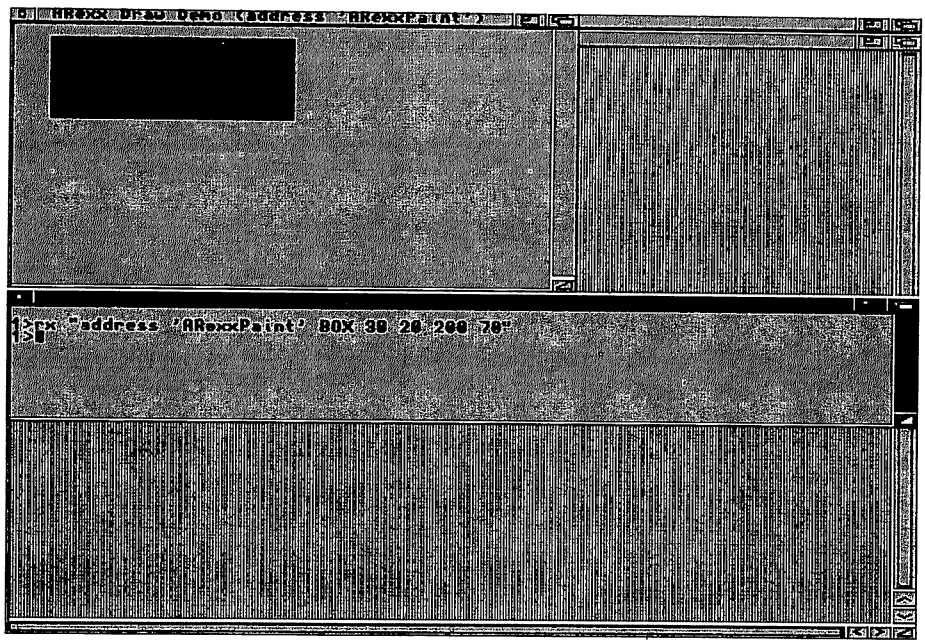


Figure 3-2: ARexxPaint window and Shell on Workbench screen

After you press RETURN on this command, a box should appear on the *ARexxPaint* display window. The BOX command was sent to *ARexxPaint*, where it was interpreted and resulted in a box of the given position and size being drawn.

You could have achieved the same end result by selecting the box tool and drawing with the program by hand, but the ARexx interface is like a 'back door' that lets you access the program's capabilities using commands.

Notice the ADDRESS instruction: the host address is 'ARexxPaint', which happens to be the same as the program's name on disk. This is not the case with all programs: the authors of the program could have chosen 'Paint program ARexx host port' or 'Limburger' as host addresses just as easily.

As long as you know the host address for an application, it doesn't really matter what it actually is. The only time a problem might arise is if two hosts use the same address, which is an unlikely occurrence.

Try experimenting with a few more commands at this point. Here is a list of the commands understood by *ARexxPaint*, along with the information (the *arguments*) that each command requires:

```
BOX left top width height

CIRCLE x y radius

LINE x1 y1 x2 y2

COLOR n (0 to 3)

CLEAR
```

Using a script

The above example shows how ARexx can send commands to an external host, but it doesn't really give you an idea of why this concept is useful: after all, you could have just drawn the box yourself! Well, suppose you want to draw something like the following drawing instead:

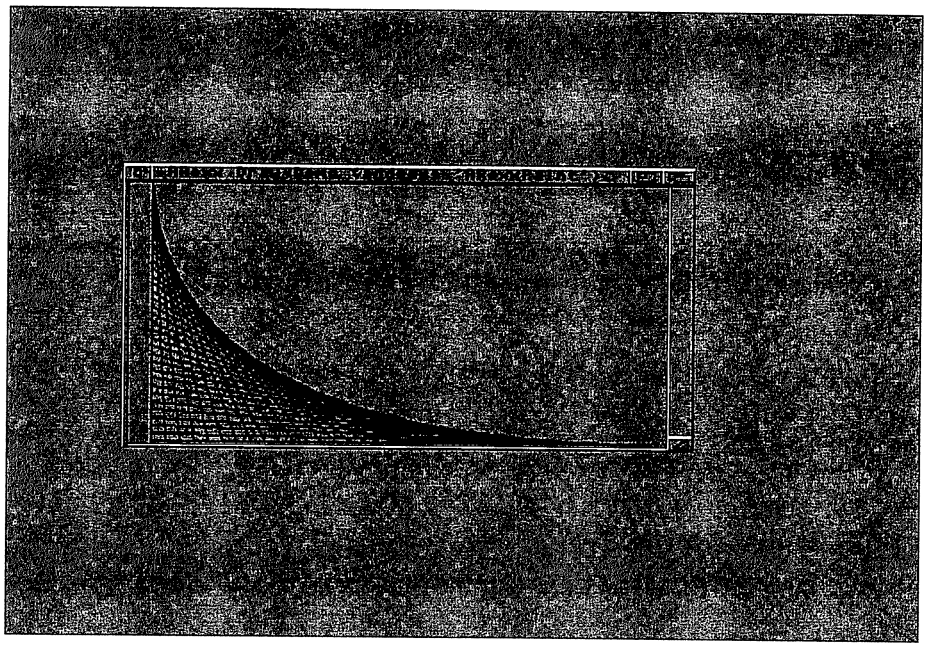


Figure 3-3

This might be time-consuming to do by hand and almost impossible to do mathematically accurately. Mathematical accuracy is, of course, what computers are good for, so why not let your computer do the work?

This would be tricky to accomplish with a paint program that only allows manual control. Even with the primitive *ARexxPaint*, however, it's a cinch, simply because of the program's ARexx command interface. The following simple ARexx script does the trick:

```
/* ARexxPaint fancy line drawing */
address 'ARexxPaint'
do position=0 to 175 by 5
    'line' 20 position 20+position*2 175
    end
```

If you like, you can test this by typing the script into a text editor, saving it and executing it using the RX program. Or, you can just take our word for it for now, since you'll learn all about writing scripts in the next section of the book.

For now, the important thing to note is the difference between the ARexx instructions in the script and the command understood by *ARexxPaint*. The 'line' command makes no sense to ARexx and is just passed on to the current command host.

The ARexx ADDRESS instruction has set the command host address to 'ARexxPaint', so that program gets the 'line' command and tries to interpret it. As it happens, 'line' is a valid *ARexxPaint* command and is followed by valid information that defines the start and end points of the line to be drawn.

The ARexx script uses variables to control the values of these start and end points and arranges to choose values that result in the pattern of lines that you see. The combination of ARexx instructions and *ARexxPaint* commands in a script lets you use *ARexxPaint* in a powerful, script-driven manner.

The above script along with several other examples are provided on the companion disk in the 'ARexxPaint' drawer. You can run these using the RX program or double-click on their icons from Workbench. It is easy to see from these examples that ARexx scripts can provide capabilities for controlling a program that are not possible or practical otherwise.

ARexx at Work

Once you learn a bit of ARexx programming, you'll probably find yourself writing ARexx scripts to handle a large number of everyday general programming problems.

Some examples of these might be:

- Go through a text file containing a list of expenses and sum those in each of several categories (telephone, auto, home repair, etc.).
- Search through all the text files in a specified directory and find all the occurrences of a few selected keywords.
- Transfer all the files on a certain disk that are older than a specified date to another disk.
- Read through a free-format text file containing names and addresses and print mailing labels for all your friends that live out of town. In other words, read a database file of any format and generate a report containing important information from selected records in any desired format.

Using macros

The above examples, and others like them, are applications that any programming language could be applied to, although ARexx is highly suitable because of its adept text-handling, easy syntax and special features. Perhaps most of your ARexx use, however, will be in the form of macros within applications, since this is ARexx's unique ability not available in other languages.

Here are a few ways that you might use ARexx macros in different applications that provide an ARexx command interface:

Word processing

- In your wordprocessor, you could define a macro to format text or a list of information in a standard way.

For example, you might type the basic text for a standard business letter and let your macro put in the date, the letterhead and set the text formatting codes to make the letter appear printed in your preferred format.

Another macro might add a column of figures or compute the result of mathematical expressions in the text. If you are running a spelling checker or thesaurus with ARexx support, you could link these programs directly to your wordprocessor.

ProWrite from New Horizons is an example of a wordprocessor with extensive ARexx support. You can define *ProWrite* macros to perform many operations on the text in a document.

Telecom-munications

- If you access a number of online services and bulletin boards using a telecommunications program that has ARexx support, you can use macros to automate any predictable sequence of events.

For example, you may have a macro to dial a particular BBS (and keep trying if the line is busy), download any new mail that may have been sent to you, then log off.

You could extend this with a 'master macro' that does this to all the BBS's on a list, perhaps while you sleep at night. If you subscribe to an online financial service, another macro might get the latest quote on a particular stock or list of stocks.

Macros could also be used to send standard sequences of text like your name or a special 'symbol' made up of an arrangement of standard text characters.

Examples of telecommunications programs with ARexx support are *Baud Bandit* and *A-TALK III.*

Home control

- The software for the BSR home control system has an ARexx interface.

The BSR hardware lets you turn lamps or appliances on and off under computer control; the ARexx interface in the software means that can be done under the control of an ARexx program. You might use a 'night' macro to turn off unnecessary lights and turn on the burglar alarm.

Database

- Using ARexx as a database command language is generally easier and more powerful than a specialized language built into the database application.

ARexx database macros could be used to generate reports or could be complex enough to create an entire specialized database application.

SuperBase 4 and *MicroFiche Filer Plus* are examples of databases that provide full ARexx support.

Spelling checker or thesaurus

- If you are using a text editor or wordprocessor with support for ARexx macros and are also running a spelling checker or thesaurus program with ARexx support, you can easily check a word even if the program you're using does not have built-in features for spelling checking or thesaurus look-ups.

Even if your wordprocessor has these features, you may wish to use the services of a program with a more extensive library of words or perhaps use a dictionary in a different language.

ASDG's *Cygnus Ed Professional* is a text editor with ARexx support and Meridian Software's *Zing!Spell* is a spelling checker with an ARexx command interface.

The Electric Thesaurus by Softwood is a thesaurus program with an ARexx interface. *ProWrite* is a wordprocessor with full ARexx support and although it already has its own built-in spelling checker and thesaurus, you can use macros to access *The Electric Thesaurus* in case you wish to use its word list, based on Roget II's thesaurus.

ARexx Support

What to look for

To be able to get the full benefit of your ARexx skills, you should have as much software as possible with built-in ARexx support.

"ARexx support" is not a specific term, and can mean very different things with different applications. When purchasing software with an eye towards its ARexx support, there are a few factors to keep in mind.

At the very least, you should look for some mention of ARexx in a product's advertisement or on the package. The words "Full ARexx support" are often used to indicate a fairly extensive ARexx command interface.

Some applications (notably user-programmable authoring systems such as Inovatronics' *CanDo*) allow you to send and receive ARexx commands, but don't actually add any of their own commands.

In these applications, you can use ARexx in your current project, but you can't use ARexx macros to control the application itself. Look for some mention of "additional ARexx commands" if you want some control of the application itself. In general, the more ARexx commands that an application supports, the more control you'll have over it.

Some applications provide ARexx commands to select menu options and perform other user-interface operations. This is often all you need: a paint program with such capabilities would be fully ARexx-programmable (a specific example is NewTek's *DigiPaint 3*).

Other programs that deal with textual data, like wordprocessors, text editors, databases and appointment calendars, should also provide ARexx access to that data. This is more important in data intensive applications like a database.

However, even a wordprocessor should allow you to actually read the text in the document from an ARexx program, allowing you to perform sorts or other operations that deal with the text itself. An example of this capability is *ProWrite*'s EXTRACT command.

See if you can find out this information about a product before you make your purchase decision, by asking a knowledgeable salesperson or by contacting the manufacturer directly.

Going one step further, some applications provide capabilities available in ARexx commands that aren't part of the regular user interface. This will mostly be found on products that use ARexx heavily as a scripting or macro language.

A look at the ARexx command reference portion of a product's documentation should give you an idea of the extent of the ARexx support in this area. Examples of such ARexx extensions might be putting up a file or font requester, getting user input or importing data from text files.

Where to get macros

Most applications with ARexx macro support will come with a few sample macros on the release disk. In some cases, these may be useful macros designed to meet a specific need or add features that the program itself lacks. Other products may simply include these macros as demonstrations of some of the possibilities with the available ARexx commands in the program.

Additional macros are available for some products as separate products. The macro packages may be general-purpose macros to add features to the program or they may make up an extension of the application that tailors it to a specific need.

Another plentiful source of ARexx macros are the various public-domain distribution sources: The data libraries of online services and bulletin boards and PD disk collections like the "Fish Disks" from Fred Fish. Many users who have created macros to fill a specific need have decided to share them with others—some of these may be quite specialized, but may serve as a useful basis for similar applications.

When you begin programming in ARexx, you will no doubt write macros to handle a variety of tasks. You'll probably find that nothing suits your needs as well as a self-written program, since you have complete control over what it does. Even so, there's no reason you can't share your creations with others! A good pool of freely distributable macros will benefit all ARexx users and will provide lots of ideas for future innovations.

To write your own macros, you'll have to learn the basics of the ARexx programming language. The next section of the book will teach you ARexx programming without controlling any external software.

Section III then moves on to applying ARexx to specific software applications.

SECTION II
AREXX
PROGRAMMING

Chapter 4
Simple Programming

In this chapter you are going to begin to learn how to write ARexx scripts. Scripts are programs, so writing scripts is a form of programming. A very friendly form, however: if you are new to programming it will prove easier than you probably expect.

To give you some idea of what's coming up, here is an objective for the chapter: To learn enough about ARexx to let you write a script that:

- Asks the user to enter his or her name.
- Reads in the name when the user types it.
- Outputs a sentence including the name the user supplied.

Here is a sample session with this script:

```
Please type your name
Nick
Coincidentally, my name is Nick also.
```

This will be the culmination of the chapter, a target to keep in mind as we develop the more elementary scripts that lead up to it. Even so, it will not be an earth-shaking script by most standards and may not in itself establish your reputation as a programming guru. But in learning enough about ARexx to write this script, you will master many of the fundamentals of the language.

Creating a Simple Script

In Chapter 2 we learned how to run an ARexx script using the *rx* program. We saw that a script is just an ordinary text file containing ARexx instructions.

In this section we will actually create, test and modify a very simple ARexx program, using methods that can be applied later on in more ambitious projects.

Entering the script

The first step is to run your text editor as described in Appendix A. Since you're creating a new file rather than modifying an old one, when you bring up the editor with a command like:

```
run memacs rexx:first.rexx
```

the editor's text area should be empty. Ready? Start by entering this:

```
/* First ARexx script */
```

This line is a *comment*. In general, comments are for your benefit only and will be ignored by ARexx. The comment on the first line of a script, however, is special—the sole exception to the rule.

ARexx insists that *every script file begins with a comment*; in its absence, ARexx will rather misleadingly report 'Program not found' when you try to run the script. Try to school yourself to think of checking for a missing comment if you see the 'Program not found' message, rather than waste your time in search of exotic reasons for ARexx's failure to find the script file.

The symbols '/*' and '*/' are like special brackets that mark the beginning and end of the comment. The comment may start and end on the same line or may extend over several lines: once ARexx has encountered the opening '/*', it will ignore all subsequent text up to the '*/'. (This is true in the first-line comment: only the existence of the comment is demanded; the text it contains, if any, is immaterial.)

Comments may occur anywhere in a script. Their main purpose is to document the program's usage, logic and flow, so that the programmer will understand the code as readily in six months' or a year's time as when it was written. Abundant, helpful, commentary is one of the hallmarks of good programming.

Now enter the second line of your new script:

```
say "Hello world!"
```

That line completes this very short script. Save the file and (assuming you named the script *rexx:first.rexx* as we suggested) test it by typing this in your CLI window:

```
rx first
```

Running ARexx Scripts

Remember that the *rx* program automatically tries the *.rexx* extension on the script name you give it and checks the *rexx:* logical directory as well as the current directory.

Given the full pathname *rexx:first.rexx*, therefore, these alternate forms would have worked just as well as the one above:

```
rx rexx:first
rx first.rexx
rx rexx:first.rexx
```

If everything went well, running your script looked like this. (We use 'Shell>' to indicate your AmigaDOS shell or CLI prompt):

```
Shell> rx first
Hello world!
```

Is that what happened? If not, the problem is probably one of these:

- ARexx is not properly installed. If you followed the instructions in Chapter 2 on installing ARexx, this shouldn't be a problem, but if you're not sure, go back and redo the procedure given there for verifying the correctness of the installation.

- Your script is improperly named. To verify that the name is correct, try this:

```
Shell> type rexx:first.rexx
/* First ARexx script */
say "Hello world!"
```

- You entered the script incorrectly. Typing the file as just described will tell you if this is the problem.

- Out of memory or a disk/hardware error (although either is quite unlikely). However, if you're really stuck, try rebooting and *then* rerun the script.

- If that doesn't do it, we suggest you seek the advice of some knowledgeable friend, your dealer or a member of your user group.

When you have the script working properly, we can move on to try some more scripts knowing that the first step has been taken: You're now programming in ARexx!

Script names

You should make it a rule to end your ARexx script file names with the suffix *.rexx*. Since ARexx itself does not otherwise care what your scripts are called, you could say that the choice of name is entirely arbitrary.

Nevertheless, care in choosing script names is amply repaid later when you scan a directory listing of *rexx:* and actually remember what each script does from its title alone, without having to examine a listing of its contents.

Avoid names like *myscript, program1* and even *foo,* a favorite name among programmers since the dawn of time. Do not be tempted by *temp* or seduced by *stuff.*

Try to make the file name do something useful. We called the script above *first*—an adequate if not inspired name. Perhaps *hello_world* would have been a better choice (notice that an underscore is generally used instead of a space in script names) or simply *hello.*

To facilitate referring to this book, a good system might be to name the scripts given here something like *bk104,* that is, the script discussed on page 104 of the book.

Such a system would also have the advantage of making the book's scripts readily identifiable later on when you want to purge or archive them.

Expressions and the SAY instruction

The line:

```
say "Hello world!"
```

in our first script is an example of an *instruction,* one of the several types of *clause* in the ARexx language. By the way, the comment line:

```
/* First ARexx script */
```

is an example of a *null clause,* which merely means that ARexx ignores it. An entirely blank line is also considered to be a null clause.

We will show you the remaining clause types over the course of this and the next few chapters. For now, however, let us look more closely at instruction clauses, specifically those that begin with 'say'.

ARexx regards any clause beginning with any of a rather small set of *keywords* and not followed by either a colon or an equals sign, to be an instruction clause or, more simply, an *instruction.*

Instructions are classified by their keywords. For instance, the instruction:

```
say "Hello world!"
```

begins with the keyword SAY and is an example of a 'SAY instruction'.

All the keywords recognized by ARexx are documented in the Reference Section at the end of the book. Just in case you're curious and to give you a more general idea of the language, here's a complete list:

ADDRESS	ARG	BREAK
CALL	DO	DROP
ECHO	ELSE	END
EXIT	IF	INTERPRET
ITERATE	LEAVE	NOP
NUMERIC	OPTIONS	OTHERWISE
PARSE	PROCEDURE	PULL
PUSH	QUEUE	RETURN
SAY	SELECT	SHELL
SIGNAL	WHEN	

Letter case

It doesn't matter whether or not you use upper case (capital letters) when spelling an ARexx instruction keyword, since ARexx automatically converts all text to upper case before acting upon it, except for text within single or double quotation marks.

Whether you capitalize the keywords in a script is therefore just a matter of personal taste. In this book, we use lower case for keywords in listings, but we use upper case for clarity when talking about keywords (such as SAY) in ordinary text.

And speaking of SAY, let's now put theory to one side again and get the feel of ARexx by trying a few more examples with this very useful instruction.

Experimenting with SAY

Here is a short script that uses the SAY instruction to test various properties of ARexx. After you've had a chance to enter and run this script we'll discuss each line in detail. First, however, try to form your own theories about what makes each SAY instruction behave the way it does.

```
/* SAY experiments */
say "Hello world!"
say 'Hello world!'
say "Hello" 'world!'
say "'Hello'" '"world!"'
say "Hello" || "world!"
say Hello World
say "12345"
say "12345 + 22222"
say 12345 + 22222
say "12345" + "22222"
```

Single and double-quotes

When you are typing this script into your text editor, be particularly careful with the two kinds of quotation mark: the single and the double. Both of these are on the key between the semicolon and the return keys.

The vertical bars (known as 'or-bars') in the sixth line of the script are located on the shifted 'backslash' key, which is immediately to the left of the back-space key.

The results

Save the script as *rexx:saytest.rexx* or an appropriate name of your own choosing. Run it, then examine the results that each SAY instruction produced. Here's what should happen (apart from the line numbering, added to aid our discussion):

```
   Shell> rx saytest
1. Hello world!
2. Hello world!
3. Hello world!
4. 'Hello' "world!"
5. Helloworld!
```

```
 6. HELLO WORLD
 7. 12345
 8. 12345 + 22222
 9. 34567
10. 34567
```

Line 1 The output on line 1 is exactly the same as the output for our *first.rexx* program—not surprising, since the SAY instruction that produced it is also identical. Notice the following:

- All the text between the double quotes was displayed in your Shell window
- The quotes themselves were not displayed
- A line feed character was output after the text. In other words, subsequent output does not begin in the next available character position, but from the beginning of the next screen line.

Line 2 Line 2 of the output is the same as line 1. The SAY instruction differs only in that single quotes are used instead of double quotes.

As you see, ARexx treats both types in exactly the same way. Text between quotes (single or double) is an example of a *string*, which essentially just means a bunch of characters to be treated as text.

Line 3 Line 3 of the output is again the same as lines 1 and 2, but this time the SAY instruction is different in an important way: instead of one string in quotes, this time we have two strings side by side, separated by a space.

Joining strings together like this is often called *string concatenation* and in ARexx it's no more difficult than putting the strings next to each other the way you'd like them to appear in the final result.

As simple and natural as this kind of string concatenation is, it is actually a form of ARexx *expression*: a collection of *terms* and *operators* that ARexx will analyze and process to determine a *result*.

In this case, the terms are the strings "Hello" and 'world!' and the operator is the blank space between them. When ARexx encounters this operator, it joins, or *concatenates*, the terms on either side of it, inserting a blank space between them.

We'll be delving much more deeply into expressions, and describing many more operators in the next chapter.

Line 4 Line 4 is a duplicate of line 3, except that it demonstrates one vital point: Single quotes have no special meaning within a string delimited with double quotes and vice versa.

This lets you use either type of quote as part of a string, simply by surrounding the string with the quote of the other type. Another way to use a quote character as part of a string is to use two quotes in a row, as in 'can"t' to produce the string *can't*.

Line 5 Line 5 is almost a duplicate of line 3. The difference is that this time, instead of the space operator, we use another concatenation operator, formed by two or-bars. The effect, as you see, is to concatenate the two strings with no intervening space.

Line 6 Did you figure out why the output in line 6 appeared in upper case?

The answer goes back to something we mentioned earlier in discussing keywords: your entire ARexx program, except for strings, is converted to upper case as it is read in. And here, since there are no quotes this time around the text, 'Hello world!' appears in capital letters.

Line 7 The instruction that creates the output of line 7 is only superficially different from most of its predecessors. Instead of 'Hello world!,' the quoted string now reads '12345'. Nothing else has changed.

Line 8 The same is true in line 8. Here the string is '12345 + 22222', which resembles an arithmetic problem but is in reality just another text string, thanks to the quotes around it. What would happen if we took them away?

Line 9 That question is answered in line 9. Then suddenly, instead of a string we have another *expression*, consisting of two terms (12345 and 22222) and one operator (the plus sign). As before, ARexx processes the expression first and returns the result. You might expect the result to be the arithmetic sum of the two numbers, and so it is.

Line 10 Although we hope that this test program has been informative, so far there has been nothing about the results that would greatly surprise someone who was initially ignorant of ARexx but was familiar with another computer language, such as C or BASIC. Line 10 of the output, however, would almost certainly be unexpected. Take another look at the SAY instruction that generated line 10:

```
say "12345" + "22222"
```

This instruction combines two *string* terms with an *arithmetic* operator; for that reason, our C programmer might well expect ARexx to regard it as erroneous.

The BASIC programmer might recall that in BASIC the plus sign doubles as a string concatenation operator and so would predict the output '1234522222'. As we see, both are mistaken.

The real result:

```
10. 34567
```

Typelessness

illustrates a very important characteristic of ARexx, known as *typelessness*.

What does this mean? Simply that ARexx treats any particular piece of data as either a number or a string, depending on the context in which it is used.

In the present example, the presence of the plus sign means that the expression makes sense *only* if the two terms are treated as numbers, although they are not numbers to begin with.

Therefore ARexx quietly converts them before evaluating the expression. Of course, not all strings *can* be converted to numbers. Given an instruction such as:

```
say "12345" + "helicopter"
```

for instance, even ARexx will report an *Arithmetic conversion error* and stop executing the script.

Breaking long program lines

By the way, you may wonder whether it would be possible to enter a program line that is too wide for your text editor window. This may happen when you want to use a very long string, for example.

Most text editors can handle this situation with horizontal scrolling, but yours may not or you may prefer to keep the whole width of your script visible at all times. You can continue an ARexx instruction over multiple screen lines by using a comma as the last character of each line (except the last line).

The comma causes ARexx to regard the next line as an unbroken continuation of the current line.

Multiple instructions on a single line

The opposite capability—putting multiple instructions on one line—is also available: just separate each pair of neighboring instructions with a semicolon. Usually this is warranted only when invoking *rx* with a 'string file' from the CLI, like this:

```
Shell>rx "r=4.8; pi=3.14159; say pi * r * r"
72.382236
```

You *have* to do it this way with string files since the variables you set in one use of *rx* are not remembered for subsequent uses.

In scripts, you're better off to stick with one instruction per line in most instances. In some cases however, using multiple instructions may bring out the script logic a little more clearly, as is arguably the case in this fragment:

```
a = 3; say a
b = 4; say b
```

More on expressions

Numeric conversions

We have just seen that ARexx treats strings as numbers when it is necessary (and possible) to do so. The invisible conversions that underpin typelessness work both ways, however. Numbers are very often treated as strings. The only difference is that this kind of conversion is always possible. Try running this little script:

```
/* More experiments with SAY */
say 1+1 "wrongs don't make a right."
say 1+1 "plus" 0+2 "equals" 7-3"."
say "Quote marks like "" can be embedded in strings."
say "One way to con" || "cat" || "enate."
say "Another way to con"'cat'"enate."
```

You might call this one *rexx:saytest2.rexx*. Here's what you'll see when you run it:

```
   Shell> rx saytest2
1. 2 wrongs don't make a right.
2. 2 plus 2 equals 4.
3. Quote marks like " can be embedded in strings.
4. One way to concatenate.
5. Another way to concatenate.
```

Line 1 The '2' in line 1 is clearly the result of the numerical operation '1+1' and so must have itself have been a number when ARexx first computed it. But the output '2 wrongs don't make a right.' is just as clearly a string, the result of applying the *space* concatenation operator to the two terms, one of which must have been the string '2'. This demonstrates that an implicit conversion from a number to a string has taken place.

Line 2 Line 2 uses the same effect, but introduces two new wrinkles:

- Instead of two string terms connected by one space operator, we have here several (6) string terms and several (5) operators. You can extend expressions in this way with no practical limit.
- When two strings adjoin with no intervening spaces, the result is the same as if the '||' concatenation operator had been applied. This effect was used to make the period follow directly after the '4' with no space.

Line 3 Line 3 shows a special feature of quotes that overrides the 'null concatenation operator' just described: quote characters can be included in a quoted string by using two of them in a row. This applies to single quotes as well as double.

Line 4 Line 4 gives another instance of the use of the now-familiar or-bar concatenation operator.

Line 5 Line 5 uses the null concatenation operator, avoiding the 'quotes within quotes' feature of line 3 by using double quotes for the first and last strings and single quotes for the one in the middle.

Simple Variables

Think back now to the task we set at the beginning of the chapter, namely to write an ARexx script that would allow the user to enter his or her name, then produce some output incorporating the name.

You can probably guess that the output is going to use SAY in some fashion. At the same time, you might wonder how we can SAY something—the user's name—that was unknown at the time of writing the script. Certainly there is nothing in what we have covered so far that would make this possible. Some ingredient must be missing.

That secret ingredient is the ability to use *variables*. You might remember variables from high school algebra, but even if that's a painful memory, don't be alarmed: there'll be no algebra here.

In computer languages, a *variable* is really a little piece of the computer's memory where a value (such as a number or a string) can be stored and the *variable name* is used to symbolize whatever value might currently be stored at the variable's location.

More concretely, let's suppose there's a variable with the classic name X and that its current value is 5. Then:

```
say X
```

should output the number 5. If we changed the value of X to another number, such as 10 or to a string, such as 'John', the same SAY instruction would produce correspondingly changed output. And the instruction:

```
say "The value of X is" X
```

would produce one of the following:

- The value of X is 5.
- The value of X is 10.
- The value of X is John.

Variables can also be used in expressions in just the same way as the *constants* we have been using up to now. Some examples:

```
say X+2
say "My name is" X
say "They call me '" || X || "'"
```

Assignment Clauses

We now have some idea of how to use the values stored in variables. If this knowledge is to be useful, however, we're clearly going to need some way of *setting* those values beforehand.

Giving a value to a variable is most often accomplished with an *assignment clause*. In the next little script, we have used comments to show which lines produce output (numbered 1 to 7) and which lines are assignment clauses.

Remember that comments have no effect on the operation of the script: ARexx ignores them altogether.

```
/* Assignment */
say X       /* 1 */
say x       /* 2 */
X = 5       /* assignment */
say X       /* 3 */
say x       /* 4 */
say X + X   /* 5 */
X = X + 1   /* assignment */
say X       /* 6 */
X = "John"  /* assignment */
say "They call me '" || X || "'"  /* 7 */
```

Running this script, which you might call *rexx:assign.rexx*, should look like this:

```
   Shell> rx assign
1. X
2. X
3. 5
4. 5
5. 10
6. 6
7. They call me 'John'
```

Line 1

In line 1, we output the value of X even before we have assigned any value to it. Other computer languages handle this differently:

- Some would regard it as an error and would halt execution of the script.
- The value of X would be undefined—essentially random.
- A standard default value, such as zero, would be given to the variable as soon as it came into existence.

ARexx's method of setting an initial value for—of *initializing*—a variable is different again. The default value for all ARexx variables is *the name of the variable itself.* This explains line 1 of the output from our test script.

Line 2

You may recall from earlier in this chapter that ARexx converts all text in a script that is not inside quotes to upper case. Variable names are no exception.

From ARexx's point of view, therefore, the second line of our test script is exactly the same as the first and produces on line 2 the same output as on line 1.

Line 3

Line 3 shows that the assignment clause 'X = 5' has had the desired effect of giving X a new value.

As you see, the structure of an assignment clause is very simple. It consists of a variable name followed by an equals sign followed by a value of some kind. Here the value is the numerical constant 5, but string-valued constants and expressions yielding either type of value are also valid, as shown in subsequent lines.

Line 4

Line 4 simply underscores the earlier demonstration that X is X even if we refer to it in lower case: X and x are one and the same variable.

Line 5

In line 5 we use our variable in a simple expression. Every time the variable name occurs in the expression, the corresponding value is substituted for it.

The instruction that produced line 5 therefore reduces to:

```
say 5 + 5
```

On another occasion, however, when X has a different value, the same script line might have a totally different meaning, such as:

```
say 101.3 + 101.3
```

or:

```
say -4 + -4
```

or even the erroneous, script-halting:

```
say "wrong" + "wrong"
```

It all depends on the value of X.

There is no finality about assigning a value to a variable. You can make new assignments to the same variable as often as you like, whether of the same or a different type of value.

The new value simply overwrites the previous one, which is lost forever. Line 6 of our output reveals the new value of X after the assignment:

```
X = X + 1
```

This line both uses X in an expression (at which point it still has its old value of 5) and as the recipient of an assignment, giving it its new value of 6.

Line 7 shows that X can be given a string value ("John") and can be used as such in an expression, just like a string constant.

Script arguments

A more specialized method for setting the value of a variable uses PARSE ARG, one of several varieties of the PARSE instruction, which you are shortly going to meet in a somewhat different context.

The effect of PARSE ARG is to copy the text given to the script as *command-line arguments*; into one or more variables. Although we will not be discussing PARSE in full detail until Chapter 10, this use is so important that we'll show you briefly how to use it right now.

Here's a very simple script for you to try. You could call it *rexx:args.rexx*:

```
/* parse arg #1 */
parse arg x
say '(' x ')'
```

This script reads whatever you type in at the command line into the variable X, then displays what it read. Here are some trial runs:

```
Shell> rx args Hello
( Hello )
Shell> rx args rexx:args.rexx
( rexx:args.rexx )
Shell> rx args
(  )
Shell> rx args one two three
( one two three )
```

As you see, everything on the command line after the script name is transferred to the variable. In the final example, it would be more convenient to read each word into a variable of its own.

This is very simple to accomplish:

```
/* parse arg #2 */
parse arg x y z .
say '(' x ')'
say '(' y ')'
say '(' z ')'
```

Notice the period at the end of the line containing the PARSE instruction. It plays a somewhat subtle but important role that will be explained in Chapter 10.

For now, you should use it whenever you want to use PARSE with multiple variables. And now to test:

```
Shell> rx args Hello
( Hello )
(  )
(  )
Shell> rx args one two three
( one )
( two )
( three )
```

Variable names

Choosing a name

The names of the variables in the examples so far have been single letters like X. Variable names in real life, however, should generally be longer and more descriptive.

X would be a good choice of name in relatively few programming situations: it might be useful in a script involving an X/Y coordinate system provided there could be no doubt as to what X was the X-coordinate of. If there were possible ambiguity, a more precise name, such as *Cursor_X*, should be preferred.

In general, variable names should be chosen to promote the readability of the script. Of these two possible versions of the same instruction, for example, there can be no doubt which makes its point more clearly:

```
1)      total = principal + interest
2)      t = p + i
```

One can carry this to extremes, of course. Although the intent of the next line is very plain and perfectly legal in ARexx (and many other languages, for that matter), most programmers would prefer to be more concise than:

```
Total_cow_weight = Weight_of_cow_1 + Weight_of_cow_2
```

The issue in naming variables, as we pointed out earlier with regard to comments, is communication, whether with some other person who may one day be trying to understand the script you have written or with yourself, months or years from now when you are trying to locate a fault (a 'bug') in the script or adapt it to some new purpose.

Allowable names

In contrast to the foregoing guidelines, the *technical* rules governing ARexx variable names are very simple. They resemble those for most computer languages.

Variable names are formed from any of the characters in the ranges a-z, A-Z and 0-9, plus any of: $?!_ (dollar sign, question mark, exclamation mark and underscore). A digit is not allowed as the first character of the name.

As a general practice, we recommend that you avoid punctuation characters other than the underscore (normally used as a stand-in for the space character), since few other languages allow them. Another sign, the period, has a special significance in ARexx variable names: it is used in *compound variables*, a subject we take up in Chapter 6.

User Input

If you remember, we proposed at the start of this chapter a specification for a simple script. We now know most of what we need to write that script, but we're still missing one item: A method of getting a line of input from the user.

PARSE PULL

And this is where we come back to PARSE. Again, we'll postpone the detailed discussion and theory until Chapter 10. Here we'll just describe the one aspect of this many-faceted instruction that our design requires.

To read a line of input from the user, use an instruction of this form:

```
parse pull <variable>
```

where '<variable>' represents a variable name, as in:

```
parse pull name
```

User interaction

ARexx will suspend execution of the script when it comes to this line and wait for the user to enter one line of text. As soon as the user hits RETURN, the text he or she has typed will be stored as a string in the named variable, just as though it had been placed there explicitly by assignment, and the program will continue.

As a rule, the script should indicate to the user what sort of input is required, using SAY (as we will do here) or the OPTIONS PROMPT instruction (covered in Chapter 10).

All but the most casual, disposable sorts of scripts should do some checking to make sure that the input line is valid. That kind of checking, however, requires language features we haven't yet covered and so is omitted for now.

PULL

Instead of PARSE PULL, you can optionally use PULL all by itself. The only difference is that the input will be translated to upper case. This makes no difference when the input is a number and may in fact simplify matters when the input is a string to be used in a comparison (we'll be discussing comparisons next chapter). When the string is to be redisplayed, however, you usually want the original case to be preserved, requiring PARSE PULL. (Incidentally, the same considerations apply to PARSE ARG, for which you can substitute just ARG if you don't mind getting the arguments in upper case.)

The final script

The following script, which you might call *rexx:name.rexx*, is the culmination of this chapter, the realization of our design objective. We hope you find its simplicity reassuring; indeed, many useful scripts are not much longer or more intricate than this.

```
/* First look at PARSE */
say "Please enter your name:"
parse pull name
say "Coincidentally, my name is" name "also."
```

Chapter 5
Numbers, Strings and Operators

Experimenting with Expressions

In Chapter 4 we briefly discussed *expressions*, including *terms*, *operators* and *variables*. These concepts are common to all computer languages, though the details of how they work vary markedly from one language to another, often in interesting and characteristic ways. In this chapter we are going to examine expressions much more thoroughly, beginning with a closer look at how ARexx handles numeric values and some peculiarities you may encounter when working with numbers.

Values

Every expression has a *value*. In fact, the whole point of most expressions is to arrive at that value so it can be used in some way: assigning it to a variable, perhaps, or displaying it to the user, or using it as an input to some other expression. In this chapter, therefore, we will primarily be concerned with values of various kinds and with performing various experiments in expression evaluation. As you know from the previous chapter, it's very easy to see the result of an ARexx expression: you simply plug it into a SAY instruction, like this:

```
say <expression>
```

For experimenting with expressions, however, even this is too much work: you must either create a little program to test the expression, as we often did in the last chapter, or use the *rx* program from the Shell and deal with a sometimes inconvenient initial quote:

```
rx "say <expression>
```

The Dialog.rexx script

Both these methods require just enough fussy typing to discourage unfettered experimentation. To get around this, we recommend you use the *Dialog.rexx* script given in the box on page 67. Here's what will happen when you run this script:

```
Shell> rx dialog
->
```

The '->' symbol is a prompt. Type in any valid ARexx expression at the prompt and the result of the expression will be displayed on the next line. There is no need to preface it with SAY—that's taken care of by the *Dialog* script.

To enter an instruction or an assignment clause, rather than an expression, just begin the line with a period. AmigaDOS commands can be executed this way with *.address command '<command>'*.

A dialogue with *Dialog*

If you enter something erroneous, *Dialog* will tell you the nature of the error. To quit *Dialog* and return to the AmigaDOS Shell, enter *bye* or *.Exit*. Here is a brief sample session to illustrate these points:

```
Shell> rx dialog
->3+3
6
->.x=3
->.y=5
->x+y
8
->.say x+y
8
->x+"atom"
*** Error: Arithmetic conversion error
.address command 'echo "Is there an echo in here?"'
Is there an echo in here?
->bye
Shell>
```

Throughout this chapter, examples will presented in the form of snippets from a *Dialog* session. You can recognize them by the '->' prompt in lines like these:

```
->47//3
2
```

We encourage you to type into *Dialog* not just the given examples, but as many others of your own as you can think of. Becoming conversant with expressions is one of the most important steps in learning to program in any language, ARexx included.

Dialog.rexx

```
/* Interactive ARexx expression processor */
rc = 0                  /* set error var to 'no error' */
options prompt "->" /* prompt used by parse pull   */
error: signal on error
                        /* jump here on command errors */
   if rc ~= 0 then do
      say "Command error: RC="rc
      rc = 0
      end
syntax: signal on syntax
                       /* jump here on syntax errors  */
   if rc ~= 0 then do
      say "*** Error:" errortext(rc)
      rc = 0
      end
do forever
   parse pull line                 /* collect input  */
   select
      when upper(line) = "BYE" then exit
      /* Other commands may be added as
                                     WHEN clauses here */
      otherwise
         if left(line,1) = "." then
            interpret substr(line,2)/* instruction */
         else
            interpret say line      /* expression  */
      end
   end
```

Numbers and Strings

In Chapter 4 we learned that ARexx knows how to deal with both numeric and string values and that in contrast to most languages it even allows these two *data types* to be used interchangeably in many contexts. Now the time has come to look more closely at certain matters we have dealt with only briefly so far.

ARexx's typelessness is achieved by storing all values, both numbers and text, in string form. Thus the number '29' would be stored by ARexx as the pair of characters '2' and '9'. That may sound natural enough, but in fact it is not the approach taken by languages whose primary goal is computing efficiency. ARexx's bias is towards making things easier for the programmer rather than optimizing the performance of calculations, so the string method of storage is for it the logical choice. It is a choice that carries with it some interesting and perhaps unexpected consequences.

Comparing numbers

The first arises when you try to answer the innocent-sounding question: 'When are two numbers equal?' Checking the equality of two numbers is a very common operation in computer programs, ARexx scripts included. However, because ARexx numbers are stored as strings, there are several possible views. For instance, which of the following numeric strings is equal to the string '3'?

```
a) "3"
b) "  3  "
c) "+3.0"
```

String *a* is clearly equal; in fact, it is identical and you can't get much more equal than that. What about string *b*? As a character sequence it is clearly not identical to '3', however the difference is only that it contains some extra spaces. We might decide it would be logical for ARexx to regard those spaces as irrelevant and consider the two strings equal all the same. String *c* looks quite different from '3', so different that no reasonable character-by-character comparison could conclude that they are equal. If they were compared as *numbers*, however, after first converting them from the string to the internal numeric form, they should be regarded as equal after all.

As we shall see in our discussion of *relational operators* later on in this chapter, ARexx provides two ways of comparing values. One method would allow only the first of the three strings above as a match for the test string; the other method would also admit both of the other strings as a match.

String storage and precision

There is another interesting and unexpected distinction between the two ways of representing numbers. Whereas the precision of the numeric representation is restricted (by default, at least) to nine digits and the range of values that may be represented is limited (though large), the string representation is capable of essentially unlimited precision. This is because the numeric representation uses a fixed small amount of memory to store each number, but strings can consist of as many characters as may be needed.

Allowable range for numbers

The range of values allowed by the numeric representation is more than 300 orders of magnitude (equivalent to 1 followed by 300 zeroes) above and below zero. Given a precision of only nine digits, however, it is clear that even within that range some numbers cannot be exactly represented. (Repeating decimals and irrational numbers, of course, cannot be represented exactly using *any* finite precision.)

Limits of precision

These limitations on numeric calculations don't affect most ARexx programming. Most often, the numbers you'll be using will be small integers. But they can produce unexpected effects, which a couple of examples may help you spot when they happen to you. Consider this example (remember that the '->' prompts show this to be a *Dialog* session):

```
->.a = 100 / 23
->.a = a * 2
->.b = 200 / 23
->a
8.69565218
->b
8.69565217
```

By the laws of arithmetic, *A* and *B* should be exactly equal, but in practice they are not. They're *nearly* equal, all right, but a round-off error arising from the limited precision of numeric calculations causes them to differ in the last digit. That slight difference would be enough for ARexx to report them unequal in a comparison unless special precautions were taken. Interestingly, because some 'spare' precision is maintained while a calculation is actually in progress, the expressions in this variant session do produce equal values:

```
->.a = (100 / 23) * 2
->.b = 200 / 23
->a
8.69565217
->b
8.69565217
```

String/number conversions

As we saw earlier, ARexx is willing to be fairly loose about accepting a string as a number, ignoring such irrelevancies as spaces at the beginning and end. When ARexx itself represents a number as a string, however, it uses very specific formatting rules. Converting a string to a number, then back to a string, is thus not necessarily a symmetrical operation. Try this little example, which demonstrates that adding zero is not always without effects, whatever you learned in school:

```
->'(' || '  1000  ' || ')'
(  1000  )
->'(' || '  1000  ' + 0 || ')'
(1000)
```

Scientific notation

ARexx uses a different format for numbers that can't be represented within the nine-digit limit of precision. This format, called *scientific notation,* is written as a value, called the *mantissa,* followed by the letter 'E', followed by another value called the *exponent.* The value of the number as a whole is obtained by multiplying the mantissa by ten raised to the power of the exponent. The mantissa is always adjusted to be at least one and less than ten. Here are some examples of numbers in both ordinary notation and ARexx-style scientific notation:

```
1                       1E+0
2.5                     2.5E+0
10000                   1E+4
17333                   1.7333E+4
.00034                  3.4E-4
123456123456            1.23456124E+11
```

Notice that in the final example precision has been lost and the value rounded up according to the normal rules.

Most computer languages offer scientific notation in this or a very similar form. The main reason we're telling you about scientific notation at this point, however, is so you won't be at a loss if you chance to come across it. (Read the Reference Section entry on the NUMERIC instruction to learn about a variant form, called *engineering notation,* that ARexx will use if requested and about modifying the precision of numeric operations and the 'fuzz factor' for numeric comparisons.) And as one final example of how inter-conversion between strings and numbers can cause some strings to take on values you may not have expected, try this *Dialog* expression:

```
->+1122334455 'contains five pairs of digits.'
```

Operators

Symbol	Pri	Type	B/U	Operation
~	8	L	U	Logical NOT
+	8	A	U	Prefix conversion
-	8	A	U	Prefix negation
* *	7	A	B	Exponentiation
*	6	A	B	Multiplication
/	6	A	B	Division
%	6	A	B	Integer division
//	6	A	B	Remainder
+	5	A	B	Addition
-	5	A	B	Subtraction
||	4	C	B	Concatenation
(blank)	4	C	B	Blank concatenation
==	3	R	B	Exact equality
~==	3	R	B	Exact inequality
=	3	R	B	Equality
~=	3	R	B	Inequality
>	3	R	B	Greater than
>=,~<	3	R	B	Greater than or equal to
<	3	R	B	Less than
<=,~>	3	R	B	Less than or equal to
&	2	L	B	Logical AND
|	1	L	B	Logical inclusive OR
^,&&	1	L	B	Logical exclusive OR

The 23 ARexx operators in descending order of priority (Pri). The types are Logical (L), Arithmetic (A), Concatenation (C) and Relational (R). The relational operators are also called *comparison operators*. The B/U column in the chart shows whether the operator is Binary or Unary.

We know from the previous chapter that an expression consists of a number of terms related to one another by operators. We encountered a few of the standard arithmetic operators and two operators that perform string concatenation. In the next few pages, we're going to meet all twenty-three ARexx operators.

Operator symbols

Operators have a number of properties, as you can see from the chart on the previous page. The most visible property is the *symbol* for the operator. Except for the four basic arithmetic operators and a few others, there is regrettably little standardization among computer languages about which symbols stand for which operations. For instance, the ARexx string concatenation operator symbol | | signifies the 'logical OR' operation in the C language. Logical OR is represented by a single or-bar in ARexx. C uses *that* symbol to signify a 'bit-wise OR', an operator ARexx lacks, just as C does not have an operator for string concatenation. If you're coming to ARexx from another language, be wary of making blithe assumptions about operator symbols.

Unary and binary operators

A very fundamental property of operators is the *number of operands* they take. Operators that take one operand are called *unary operators*. The most familiar example of a unary operator is the *prefix negation*, or *unary minus* operator in a number like -10. If you've guessed that operators taking two operands are called *binary operators*, congratulations. Most operators, such as those for addition and multiplication, are binary. As you can see from the chart, ARexx has three unary and twenty binary operators.

Types of operators

Another property, which we touched on in the previous chapter, is *type*. We are already familiar with the *arithmetic* operators, which expect number-valued operands and return a numeric result, and with the *concatenation operators*, which create a new string by combining string-valued operands. The other two operator types, *logical* and *relational* (or *comparison*), both involve a kind of value that is neither numeric nor string: a *boolean* value.

Boolean operators

The word *boolean* has a rather technical sound, especially if one has heard it used a few times without quite knowing what it means. Nevertheless, it is very simple: there are only two boolean values and those are *true* and *false*. For instance, supposing we make the assertion 'two is greater than one'. We can do this in ARexx with a relational operator, as in this expression:

```
2 > 1
```

If you were reading this expression aloud you would use the same words as we did above: 'two is greater than one'. Obviously the assertion is true and *true* is indeed the value of the expression. If we had said:

```
2 = 1
```

which would be read 'two equals one', the expression result would obviously be *false*.

And that's all there is to boolean values as such. We know, however, that ARexx likes to store all values as strings and this is the case with booleans too. But the strings to which booleans equate might surprise you. They are, for *false* and *true* respectively: '0' and '1'. Try this:

```
->2 > 1
1
->1 > 2
0
```

Using boolean expressions

Understand? In that case, you won't be surprised by this odd-looking example:

```
->3 + (2 > 1)
4
```

From the examples, you can see that the relational operators compare either two numbers or two strings and return a boolean value from the result of the comparison. The *logical* operators, the last type, work on boolean values and return a boolean result.

Logical operators

In English we might express one phase of a reasoning process thus: *"If* lions eat people *and* that animal over there is a lion *then ..."* and go on to draw some conclusion. The word *and* in this example is a boolean operator that takes as operands two boolean values. If both are *true*, then the value of the whole *and*-expression is also *true*. If either is *false* (if lions do not eat people, for instance), the value of the whole expression is also *false*. Our observation about lions is an example of human-style verbal reasoning—there is no direct way in ARexx to express the idea 'lions eat people'! There is, however, an operator with the same meaning as the *and* and an IF...THEN instruction to use it with. We'll look at IF...THEN later on in this chapter and at a number of other logical operators.

Operator priority

The final property of operators is called *priority*. It specifies which operations in a complex expression will be performed first. The priority rules for the arithmetic operators are well known. Consider this expression:

```
12 + 3 * 2
```

In the absence of priority rules, one could interpret this expression in two distinct ways. If the addition were performed first, the result of the expression would be 30; if the multiplication were performed first, the

result would be 18. Mathematical convention dictates that the latter is the correct result.

As you see from the table, all operators have an assigned priority. There is no need to remember the priority numbers, but these observations will help you keep them straight:

- All the unary operators have equal priority, higher than the binary operators.
- The binary operator priorities cluster by type: A,C,R,L (descending).
- AND has higher priority than the other binary logical operators.

Grouping operations

The priorities assigned to operators are such that an expression written out in the natural way will often be grouped as you intend, but exceptions are frequent. Neither ARexx nor any other language can tell, for example, that in the following expression, which converts Fahrenheit to Celsius, the subtraction should be performed before either of the other operations:

```
F - 32 * 5 / 9
```

To force a certain required grouping of operations, you must use *parentheses* ('round brackets'). Parts of an expression inside parentheses are always evaluated before the parts outside, overriding normal operator priorities. For example:

```
(F - 32) * 5 / 9
```

would give the desired result in the Fahrenheit to Celsius conversion.

Concatenation operators

We met the concatenation operators in Chapter 4. Here we'll just re-emphasize a fact that we took advantage of in chapter 4 without explicitly taking note of it: that these operators have lower priority than the arithmetic operators. As a reminder, here are some examples of the concatenation operators in action.

```
->.name = 'Felix'
->name 'the Cat'
Felix the Cat
->'They call him "' || name || '".'
They call him "Felix".
->'They call him "'name'"!'
They call him "Felix"!
```

Arithmetic operators

Nearly everyone is familiar with the four fundamental operations of arithmetic: addition, subtraction, multiplication and division. ARexx has an operator for each of these. In common with most computer languages, ARexx uses the following symbols for these operators:

+ addition

- subtraction

* multiplication

/ division

Try a few examples in *Dialog*:

```
->11+13
24
->1024-764.331
259.669
->-3*12
-36
->1000/11
90.9090909
```

Integer division A variant of the division operation is *integer division*, which uses the percent symbol **%**. Integer division is like regular division but throws away the fractional part of the result, if any. It is often useful in real-world calculations, in which fractional quantities must often be discarded. How many 75-cent apples can you buy with the $4.29 in your pocket? Conventional division gives the answer 5.72, but the shopkeeper may not approve of your attempt to leave 0.28 of the sixth apple on the shelf. Integer division gives a more practical answer:

```
->4.29/0.75
5.72
->4.29%0.75
5
```

Remainder

Another division variant is the *remainder* operation (symbolized by a pair of slashes: //) which answers the question, 'If this number were divided into that number to produce an integer result, how many would be left over?' As we just saw in our apple example, dividing 0.75 into 4.29 using integer division gives 5; the remainder operation would give the difference between 4.29 and 5 times 0.75, which is 0.54. And that, of course, means that you'll have 54 cents left after buying the 5 apples.

```
->4.29//0.75
0.54
```

Modulo arithmetic

The remainder operator is often used with integer operands in what is called 'modulo arithmetic' or, from its most familiar illustration in everyday life, 'clock arithmetic'. What is the sum of ten and seven? Seventeen? Not on a twelve-hour clock: the answer is five. You could say that adding twelve on a clock is equivalent to adding zero, since it leaves the hands in the same position. Similarly, any value of twelve or more can be simplified by 'discarding all the twelves in it', just as discarding a twelve from seventeen leaves the result of five displayed on the clock. It is easy to imagine clocks with other than twelve divisions and in the same way we can have 'modulo 5' or 'modulo 19' arithmetic as easily as the 'modulo 12' arithmetic implemented by the clock. Let's use the remainder operator to determine the time seven hours after ten o'clock:

```
->(10+7)//12
5
```

Exponentiation

ARexx provides one more binary arithmetic operator: the *exponentiation* operator, whose symbol is a pair of asterisks: **. You can translate this symbol as 'raised to the power of'. The expression:

```
side ** 3
```

thus returns the value of the variable *side* raised to the third power (giving the volume of a cube based on the length of one of its sides). It is equivalent to:

```
side * side * side
```

In ARexx's exponentiation operation, the exponent (the right hand term) must be an integer. This means you can't use it to extract roots, for example.

Exponentiation has the highest priority of the binary arithmetic operators. The first of the following expressions is equivalent to the second, but not to the third:

```
1. a + b ** 3 * 4
2. a + ((b ** 3) * 4)
3. (a + b) ** (3 * 4)
```

Next in priority come the multiplication and all three division-related operators; the addition and subtraction operators follow in last place. Remember, however, that the arithmetic operators as a group have higher priority than all the other binary operators.

Prefix conversion and negation

The two unary arithmetic operators are formally known in ARexx as *prefix conversion* and *prefix negation*, though programmers coming from other languages would probably tend to call them *unary plus* and *unary minus*. The prefix negation operator has in ARexx its expected role of negating (that is, making negative) its single operand. For example:

```
-7
```

If you're new to programming, it probably wouldn't have occurred to you that the minus sign in this example is an operator at all: you'd be more likely to think of it as just part of the number. In fact, because the high priority of the unary operators binds them very tightly to their operands, you can in most instances go on thinking of the minus sign as part of the number without ill effect.

Converting a string to a number

You might guess that the prefix conversion, or unary plus, operator would be purely cosmetic. After all, 7 and +7 are exactly the same value. Not so, however. The trick is that the plus sign *forces* its operand to be treated as a number, so the standard string-to-numeric conversion is applied. Try this:

```
->"    0012"
   0012
->+"    0012"
12
```

The effect is the same as an addition of zero to the operand (an experiment we tried earlier in this chapter, you may recall), but is more efficient and less obtrusive.

Relational (comparison) operators

As we saw earlier in this chapter, the relational operators compare two values and return a boolean result: *true* or *false*. We found that the boolean values can also be treated as numbers: one and zero respectively.

We also saw that ARexx has two modes, or standards, for deciding if a given pair of strings is equal. One mode calls for the strings to be strictly identical on a character-by-character basis. This is the mode used by the *exact equality* and *exact inequality* operators. The other mode ignores unmatched space characters at the beginnings and ends of the strings, and if the strings are numeric considers them equal provided they are equal in numeric quantity (recall that '3' and ' +3.0 ' are equal according to this standard). All the other relational operators use the second mode.

Conditional instructions

Relational operators are normally used in *conditional instructions*, of which there are several. The simplest, mentioned earlier in this chapter, is the IF instruction. This has the form:

```
if <boolean expression> then
   <do something>
<further instructions>
```

ARexx handles this by evaluating the boolean-valued expression, which might be a test for the equality of two numeric values, or some other relation. If the value arrived at is *true*, the instruction represented here by *do something* is executed. If the value of the expression is *false*, on the other hand, *do something* is skipped over and execution resumes at *further instructions*. The IF instruction will be explained in greater detail next chapter, but this quick introduction should be enough to get you through the examples in both this section and the next, which covers the logical operators.

Exact equality

The symbol for the exact equality operator is a pair of equals signs (==). Here is an example of its use:

```
/* Exact equality */
if text == ' ' then
   count = count + 1
```

Here we are testing the string variable *text* to find out if it contains exactly one space character, no more or less; if it does, a count of spaces is incremented. The exact equality operator is necessary in this example because it is not allowable for spaces to be ignored. With the regular equality operator, the comparison would succeed no matter how many spaces were stored in *text*, or even if *text* contained no characters at all.

Exact inequality

If we were counting non-space characters, we could use exactly the same snippet of code, but use the exact *in*equality operator. This is formed by preceding the exact equality operator with the *tilde* character (~). In ARexx operators, the tilde can always be read as 'not'. Our example becomes:

```
/* Exact inequality */
if text ~== ' ' then
   count = count + 1
```

These examples are not unrealistic or even exotic, but they are rather specialized. In fact, the 'exact' operators are used much less often in most ARexx programming than their less particular brethren.

Normal equality

In most cases, one really doesn't care about, or even want to know, such irrelevant facts as whether a string has spaces in front or behind, so the regular equality operators are to be preferred. When input is obtained from the user, for example, leading and trailing spaces should virtually always be ignored:

```
/* Normal equality */
parse pull input
if input = 'quit' then exit
```

In a case like this, even if the user actually happened to type ' quit', or 'quit ', or something similar, we can be reasonably assured that the spaces were not intended to be significant. The normal, unfussy, equality operator is therefore chosen.

Normal inequality

After the discussion so far, the normal inequality operator should need no special explanation. For completeness, here's an example:

```
/* Normal inequality */
if menuchoice ~= 3 then
   say "Sorry, your choice is not valid at this time."
```

Other inequalities

The remaining relational operators test for specific inequalities: less than, greater than, less than or equal to, greater than or equal to. All these operators use the normal comparison mode, rather than the exact mode. The symbols used for these operators are standard, however ARexx is unusual in providing synonyms for the *less than or equal* and *greater than or equal* operators. As you see from the chart, the synonym for the former is a combination of symbols that would be read 'not greater than', while for the latter the reading is 'not less than'. A moment's reflection will tell you that the synonyms make sense. Whether you use them is up to you; they may occasionally better express the underlying sense of a particular relation.

Comparing strings

Further examples of the relational operators can be found in the next section, on the logical operators. Our final example in this section uses the *less than or equal to* operator in a comparison of two strings:

```
if word1 <= word2 then
   say 'The words are in alphabetical order.'
```

Lexical order

The purpose of this contrived example is to show you how concepts like *less than* and *greater than* are applied to comparisons between strings. As you might expect, *less than* for strings means 'earlier in alphabetical order' and *greater than* means the opposite. But because strings can contain many characters besides the alphabetic ones, the ordering is actually more comprehensive: the term 'lexical order' is often used for this extension of the concept of alphabetical order to a complete character set. On the Amiga, a complete character set is laid out like this, starting with the characters that come earliest in the lexical ordering:

32 control characters (backspace, tab, line feed, escape, etc.)
16 punctuation characters, starting with space
10 digits (ordered 0 to 9)
 6 punctuation characters
26 uppercase letters, A-Z
 6 punctuation characters
26 lowercase letters, a-z
 5 punctuation characters
The delete character

That accounts for the first 128 out of 256 characters. These 128 characters are the standard ASCII set that most computers use. The second 128 are much less standardized. On the Amiga *only* they go like this:

32 control characters, not often used
32 miscellaneous punctuation characters and symbols
32 characters, mostly uppercase letters from foreign alphabets
32 characters, mostly the corresponding lowercase letters

A complete chart giving the Amiga's extended ASCII character set is included in Appendix C of this book. Refer to it if you need to know how various punctuation characters, for example, will behave under lexical ordering.

Logical operators

As we have seen, arithmetic operators require numeric operands and produce a numeric result; concatenation operators require string operands and produce a string result; and relational operators require operands of the same type, either string or numeric, and return a boolean result. We are now going to look at the logical operators, which all act upon boolean operands and return a boolean result. But what exactly is a boolean operand? The possibilities include:

- A comparison using one of the relational operators (remember the value of the comparison is boolean).
- A logical operation using one of the operators discussed in this section.
- A numeric value, such as a variable or a number, of either 0 or 1.

Since there are only two boolean values in the entire universe, the work done by the logical operators is stunningly simple. For instance, the sole *unary* logical operator is the *not* operator. Its job is merely to invert the boolean value of its operand. Consider the relational expression involving *less than* in this fragment:

```
if birthyear < 1951 then
   say "Howdi, old-timer!"
```

The NOT operator (~)

The *not* operator, whose symbol is a tilde, gives us one way to invert the sense of the expression:

```
if ~(birthyear < 1951) then
   say "G'morning, young'un!"
```

The high priority of the *not* operator makes the parentheses necessary. If they were left out, the *not* would be taken to apply to the variable *birthyear*, which is not only incorrect logic but would also cause the program to halt unless *birthyear* was either zero or one—rather unlikely!

The AND operator (&)

The binary logical operators are just as straightforward. For instance, the *and* operator, represented by the 'ampersand' character (shift-7 on the keyboard), combines the values of its two boolean operands into one result value in this way: if both values are *true*, make the result *true*; otherwise, make the result *false*. Here's a fragment from a (non-existent) medical self-help program. It checks to see if the patient's temperature is within one degree of the normal value:

```
if temp ~< 97.6 & temp ~> 99.6 then
   say 'Temperature is normal.'
```

As you can see from this example, testing that a value lies within a specified range requires that two tests be satisfied. Here the tests are embodied in relational expressions and the *and* operator ensures that both are *true*. No parentheses are needed in this instance, because *and* has a lower priority than any relational operator.

Inclusive OR (|)

Another binary logical operator is called *inclusive or* (often just *or*); it is symbolized by a single or-bar (shift-backslash on the keyboard). The *or* operator is much more liberal than *and*: it gives a *true* result if either of its operands is *true*, returning *false* only if both operands are *false*. This next fragment is from a computer-assisted screening procedure to identify alien life forms:

```
if color = 'green' | heads > 1 then
   say 'Further tests recommended.'
```

As in this example, *or* is used when either of two conditions would lead to the same action. The screening procedure is set up so that either green skin or multiple heads is considered significant—the presence of both characteristics simultaneously is not required for the test to succeed.

Here is another example using *or*. This time we extend the expression to include more than two terms:

```
if month=4 | month=6 | month=9 | month=11 then
   say 'This month hath 30 days.'
```

Exclusive OR (^)

The final binary logical operator is called *exclusive or*, whose symbol is the *caret* character (shift-6 on the keyboard). It is somewhat less commonly used than the previous two. It returns *true* if exactly one of its operands is *true*. If both operands are *false*, or both are *true*, the *exclusive or* operator returns *false*. Here is a fragment that depends on the fact that the product of two numbers is negative only if one of the numbers is negative and the other is positive:

```
if a < 0 ^ b < 0 then
```

```
say "The product of" a "and" b "is negative."
```

By the way, can you think of a simpler way to program this fragment and achieve the same effect?

Other String Forms

Although humans like to think of the values manipulated by computers in forms that make sense to us—numbers, strings, pictures and so on—we know that at the hardware level these different types of data are represented in a uniform way and that we depend on software to translate it appropriately. Most computer users learn at some point that data of every type can be viewed as a collection of 'bytes', each of which can assume any value between 0 and 255 inclusive. A corollary of this fact is that the numbers in a given set of bytes may mean many different things, depending on the context in which they are used.

ASCII storage of strings

In ARexx strings, for instance, every byte is normally taken to represent one character according to the standard ASCII mapping mentioned earlier in the chapter. For instance, if the character 'R' occurs in a string, the corresponding byte contains the number 82; the character 'D' is represented by 68; the character '2' by 50. Accordingly, the string 'R2D2' would be stored as four successive bytes with the values 82, 50, 68 and 50.

Control characters

In most cases, the ARexx programmer doesn't need to think about the method of representing strings: it is much easier to deal with the characters themselves than with the numbers that encode them. Some characters, however, cannot be represented in string form. These are the ASCII 'control' characters, such as the 'linefeed' character that is produced by the RETURN key on the keyboard. Others include 'backspace', 'delete', 'formfeed' (which ejects the page if sent to a printer, but clears the screen if given to the console) and the 'escape' character used to introduce special console and printer sequences. In fact, of the 256 possible values a byte can take, more than 60 cannot be entered as characters at the keyboard.

To include control characters in a string, ARexx allows two alternative representations of strings using the hexadecimal and binary numbering systems respectively. If you have programmed computers before, you may be familiar with hexadecimal and binary; for you, here are examples of both kinds of string:

```
"03 2c"x     /* $032c (decimal 812) */
'00011101'B /* %11101 (decimal 29) */
```

The 'x' (or 'X'), of course, signifies hexadecimal and the 'B' (or 'b') binary. Either single or double quotes may be used and blanks may be added at byte boundaries if desired. Zeroes are added by ARexx to the left of the string to round it out to a byte boundary; the following two binary strings are equivalent, for instance:

```
'11101'b
'00011101'b
```

Linefeed character

All readers, even if those not familiar with hexadecimal, can still use the special hexadecimal strings to specify control characters. The one you're likely to need most is the linefeed character. From the ASCII table at the back of this book, you can find out that the hexadecimal equivalent of this character is '0A'. To convert this into a string, just add the 'x', yielding usages like this:

```
LF = '0a'x

ThreeLFs = '0a0a0a'x

say "Down" || '0a0a0a'x || "here!"
```

Any other character or sequence of characters in the ASCII table, which is to say any combination of bytes at all, can be specified in this way.

More information on binary and hexadecimal strings may be found in the Reference Section. The entries for the C2X, X2C, D2X and X2D functions may be of particular interest; also those entries whose name begins with 'BIT' and a number of the functions belonging to the ARexx support library *rexxsupport.library*.

Chapter 6
Compound Variables and Built-in Functions

In the previous chapter we studied the elementary components of expressions—numbers, strings, variables and operators—in what may have seemed pedantic detail. In this chapter we turn to the more dramatic aspects of ARexx expressions: compound variables and functions.

Simple Variables

Before we get into the first main topic of this chapter—compound variables—let's take time for a quick review of the *simple variables* we have encountered in the past. We'll also learn a new instruction, DROP.

Variables provide a means of associating a name with a stored value, such as a string or a number. Legal variable names conform to two simple rules:

Variable names

- The only characters allowed in the name are the letters, the digits and a few punctuation symbols: $?!_ (dollar sign, question mark, exclamation mark and underscore).
- The name may not begin with a digit.

It doesn't matter whether you use all lower case, all upper case, or mixed case in a variable name, since ARexx always translates unquoted text to upper case before applying any other processing. This frees you to use case for readability. For instance, of these three equivalent variable names, the first is much the easiest to read:

```
CostPerAnnum
costperannum
COSTPERANNUM
```

Until it is assigned a value, the default value for any variable is its own name, translated to upper case. This very short program:

```
/* Default variable values */
```

```
say CostPerAnnum n my_name
```

produces the following output:

```
COSTPERANNUM N MY_NAME
```

'Forgetting' a variable

It is possible to make a variable forget that it has ever been assigned a value and revert to the uninitialized condition of the three variables in the last example. This is accomplished with the DROP instruction, as demonstrated in this brief script:

```
/* DROP demo */
CostPerAnnum = 150
n = 13.9
my_name = 'Athelstan'
say CostPerAnnum n my_name
drop CostPerAnnum n my_name
say CostPerAnnum n my_name
```

The output from this script would be:

```
150 13.9 Athelstan
COSTPERANNUM N MY_NAME
```

Compound Variables

The variables we have used so far associate a single name with a single value. Nearly all languages also provide one or more *composite* variable types, in which a single name may be associated with *multiple* values. This facility in ARexx is called *compound variables.* Before we look at them, let us try to get a sense of what composite variables are and how they are used.

Arrays

The type of composite known as an *array* or a *table* is provided in nearly every language. The most straightforward, a *one-dimensional array,* is like a lot of simple variables of the same type (all numbers or all strings, for example) piled one on top of the other. The variable name is applied to the whole pile rather than any one of its *cells* or *elements;* an individual element is referenced by means of a numeric *subscript* along with the group name. In many languages (though not ARexx) an array reference looks like this:

```
name(sub)
```

Here *name* is the name of the array as a whole and *sub* is a numeric expression specifying which cell within the array is wanted. The concept of arrays is usually extended to include higher dimensions. A two-dimensional array might be used to locate a point on a plane, say or to organize health data by two variables such as age and weight. A reference to such an array uses two subscripts instead of one, such as:

```
heartrate(age, weight)
```

Arrays of three and more dimensions work analogously. A central feature of all arrays, however, is their homogeneity: no matter how many dimensions they may have, all the elements they contain are of the same type.

Records

A *record*, or *structure*, is a form of composite data in which the constituent elements are *not* normally all the same type. Records are used to create new data types that combine related pieces of information in one handy package. In most languages that support records, the programmer creates a *template* that names the data type and names the *fields* (individual elements) each record will contain. For instance, a template called *book* might contain a numeric field entitled *publ_date*, a string field entitled *title* and other fields of various types. The template is not data in itself; it merely specifies a pattern in which data may be stored.

An actual record of type *book* would have its own name, such as *current_book*, with its own values for the various fields designated in the template. In many languages (including ARexx, as it happens), references to data within the structure could look like this:

```
current_book.publ_date
current_book.title
```

Other data structures

Some languages provide built-in support for other data structures such as *lists* (in which each element contains information that may be used to locate the next, and sometimes the previous, element) and many besides—*stacks*, *trees* and *sets* to name a few. More often, the more exotic constructs and the operations to support them are not built into a language but are emulated using simpler resources. Amiga programming in C, Modula-2 or assembly language, for instance, often

makes heavy use of a data organization called a *doubly-linked list*, which is not a native facility in any of those languages but can readily be emulated.

Data structures in ARexx

The compound variables of ARexx do not directly correspond to any of the typical data structures we have been discussing. They do, however, provide a very flexible means in which all these structures and more can easily be emulated, just as C and Modula-2 use records to emulate doubly-linked lists.

An array using compound variables

In fact, a use of compound variables that is easy to understand at first sight is the one that emulates an array. Let's write an ARexx script to make a somewhat abbreviated periodic table of the elements:

```
/* Create table of elements */
elements.1 = 'Hydrogen'
elements.2 = 'Helium'
elements.3 = 'Lithium'
elements.4 = 'Beryllium'
elements.6 = 'Carbon'
```

Stems* and *nodes

This script creates six compound variables sharing a common *stem symbol*, the name *elements*. In many respects, compound variables are very similar to the simple variables we've seen before. For instance, the uninitialized variable *elements.7* will return the value 'ELEMENTS.7' if you try to use it. Compound variables have a very important special property, however: when a compound variable is used in a program, ARexx replaces each segment—or *node*—of the name after the stem with its current value and only then returns a value for the name as a whole. To see what this means, suppose we continue the above script with these lines:

```
i = 3
say elements.i
```

Substitution of node names

ARexx recognizes *elements.i* as a reference to a compound variable (because of the period in the name). Before it can determine a value for the compound symbol as a whole, ARexx must—according to the rule we stated above—replace each segment after the stem *elements* with its current value. In this case there is only one segment, the symbol *i*, whose current value is '3'. ARexx makes the substitution and arrives at a final version of the compound symbol: *elements.3*. This symbol has a value—'Lithium'—and that is what eventually reaches the user via SAY.

Initializing a stem

The stem portion of a compound variable has a particularly handy special feature that lets us preset the value of the variables created using that stem. For example:

```
/* Pre-initialize an array */
elements.  = '???'
elements.1 = 'Hydrogen'
elements.2 = 'Helium'
say elements.1 elements.2 elements.3
say elements.999 elements.heating elements.euclid
```

As you will observe if you run this script, the two initialized variables have their expected values, while the other four all have the value '???'. Because we assigned that value to the stem, it is automatically conferred on *all possible variables created from that stem.* In fact, this is in general the significance of the period at the end of the variable name: it specifies all the variables whose names begin with that stem not just in assignments but also in the DROP instruction and in other contexts we haven't yet encountered.

The advantages of arrays are most apparent in connection with *loops,* which provide a concise and powerful way of processing all the compound variables formed from a particular stem. Loops are discussed in the next chapter.

Creating records

Records are easy to emulate using compound variables. Let's revert to an earlier example and create a stem called *books*. Thanks to the flexibility of compound variables, we don't have to submit a template that defines our record type to ARexx, but we should have such a template in mind. Recalling that a record has named elements, called fields, we might choose to construct each of our book records like this:

field	description
`title`	Full title of the book
`author`	Author's name
`date`	Publication date of the book
`pages`	Number of pages

The details of the data structure in real life would naturally depend on the need at hand. In some particular application dealing with books the number of pages might not be important but some other field or fields—library call number, list of topic keywords, etc.—would be required.

Initializing the records

Now let's initialize a record for the book *Desolation Island*. We'll call the record itself *island* and set it up like this:

```
books.island.title  = "Desolation Island"
books.island.author = "Patrick O'Brian"
books.island.date   = 1978
books.island.pages  = 325
```

And here's another, this time for the book *Mankind and Mother Earth*:

```
books.mankind.title  = "Mankind and Mother Earth"
books.mankind.author = "Arnold J. Toynbee"
books.mankind.date   = 1976
books.mankind.pages  = 641
```

Accessing the 'database'

We now have a database of literary information—admittedly a very small one. To 'query' the database, we need only one piece of information: the keyword, such as 'island', for the book we're interested in. Let's make the improbable assumption that the user of our database has these keywords committed to memory. Now we can let him or her access it with code like this (we assume that the *books* database has been initialized using the assignments given above):

```
say "Enter the keyword for the book you're interested in"
pull keyword
```

The *keyword* variable now presumably contains either 'ISLAND' or 'MANKIND'. If it does not, the user will learn something about the default values of ARexx compound variables. We now provide information about the book of interest:

```
say "Title :" books.keyword.title
say "Author:" books.keyword.author
say "Date  :" books.keyword.date
say "Pages :" books.keyword.pages
```

To understand what is happening here, recall that ARexx processes the symbol in each node of the compound variable after the stem, substituting in its current value. From the first of the four lines above, let's consider the compound variable reference:

```
books.keyword.title
```

The first symbol ARexx encounters after the stem is *keyword*, whose value is the input the user supplied to the PULL instruction. Let's suppose the user's input was 'ISLAND', so ARexx substitutes that in:

```
BOOKS.ISLAND.title
```

There is no variable called *title* in this program, so the value of the symbol 'title' is 'TITLE'. This gives us:

```
BOOKS.ISLAND.TITLE
```

This is the name of the compound variable to which we have given the value 'Desolation Island', so that is the ultimate value of the reference:

```
Desolation Island
```

You may wonder what would have happened if there had been a variable called *island* whose value was, say, the number 5. Would ARexx, having evaluated *keyword* to 'ISLAND', go one step further and produce:

```
BOOKS.5.TITLE
```

as the final name of the compound variable? The answer is no—only one level of substitution is performed for each segment of the name. If there had been a variable called *title*, of course, *its* value would have been substituted as well. To see how well you understand the mechanism, try to predict the output from this somewhat tricky fragment:

```
1  title = 3
   say books.island.title
2  title = "DATE"
   say books.island.title
3  field = "TITLE"
   say books.island.field
```

To understand what this code does, just put yourself in ARexx's shoes (so to speak). In each of the three SAY instructions here, the symbol *books* is translated to 'BOOKS' and island to 'ISLAND'.

Example 1

In the first instance, the final segment is *title*, whose value was set to the number '3' in the previous line. The final variable name, therefore, is:

```
BOOKS.ISLAND.3
```

Since no variable of that name has been created, the value of the compound symbol is itself and that is what the SAY instruction will output.

Example 2

In the second of the three instances, the final segment is again *title*, but this time its value is 'DATE'. Substitution yields the name:

```
BOOKS.ISLAND.DATE
```

The SAY instruction will accordingly output '1978', the value we assigned to this compound variable when we set up our database.

Example 3

In the third instance above, the final segment of the name is *field*, to which we've just assigned the value 'TITLE'. The name ARexx ends up with is:

```
BOOKS.ISLAND.TITLE
```

This variable has the value 'Desolation Island'. Notice that this result depends on the fact that the substitution process is only carried to one level, as pointed out earlier.

The number of nodes a compound variable may have is practically speaking unlimited. No matter how many there are, however, every reference to such a variable results in the substitution process we have been describing. Compound variables remain a murky area even to some quite experienced ARexx programmers, but if you have followed this section closely, you should have little difficulty in taking advantage of the unique capabilities these variables offer.

Functions

ARexx's operators, which we met last chapter, provide an efficient but deliberately elementary set of 'hard-wired' tools for manipulating values in expressions. *Functions* are similar in that they too provide a way of processing values during expression evaluation. Functions, however, trade off performance for flexibility: they do not have the same hard-wired efficiency as the operators, but they allow an unlimited range of operations to be performed.

Calling functions

Functions are known by their names, which are formed by the same rules as variable names. Each function is essentially a subordinate script, or *subprogram*, which can be *called* (or *invoked*) when required. Some functions need raw material to work on: they require that values be *passed* to them by the calling script; the number and type of these *function arguments* is a characteristic of the individual function and must be known to the programmer. Functions may also *return* values to the caller (though not all do). A value returned by a function may be used in a script just like any other value—as an argument to another function, for instance. Often, too, a function is called not because (or not only because) of the value it will return but for some other desirable behavior, such as obtaining data from or storing data to a disk. Because behavior like this is incidental to the original conception of a function, which focused strictly on the ability to obtain a returned value, it is sometimes termed a *side effect*. This is

not meant at all to convey irrelevancy, however: some functions are of interest *only* for their side effects.

Examples of functions

To give you some idea of what functions are used for and to give you something concrete to help focus the abstractions of the last paragraph, here are very brief descriptions of a few representative functions among those that are available to all ARexx scripts:

The **Abs** function takes one argument, which must be a numeric value. It returns the 'absolute value' of the number; that is, with the sign changed to positive if it is not positive already.

The **Length** function also takes one argument, in this case a string value, and returns the number of characters in the string.

The **Word** function takes two arguments. The first is a string and the second is the number of a word in that string. The returned value, another string, is the specified word. For instance, if the first argument is 'The quick brown fox' and the second argument is 3, the returned value will be the string 'brown'.

Let's now write a short script that simply demonstrates each of these functions. Notice that the arguments to the function appear in parentheses after the function name and that multiple arguments are separated by commas. It is also worth emphasizing that the function call is equivalent to any other kind of value as far as the expression in which it appears is concerned.

```
   /* First function calls */
   say "The absolute value of -3 is" abs(-3)"."
   txt = "The Hunchback of Notre Dame"
   say "There are" length(txt) "characters in '"txt"'."
   say "The fourth word in the string is
'"word(txt,4)"'."
```

Here are brief descriptions of a few more of the more than eighty functions from the set—the *built-in functions*—that contains the three discussed above:

The **Sign** function takes a single argument, a number. It returns one if the argument is greater than zero; zero if the argument is exactly equal to zero; and minus one if the argument is less than zero.

The **Reverse** function takes a single string argument and returns the same string, but with its characters in reverse order.

The **Upper** function also takes a single string. It returns the same string, but with all lower case characters converted to the equivalent upper case.

See if you can write a script, like the previous one, to try out these three functions.

Function arguments

STRIP is one of many string manipulation functions in the built-in library. Its purpose is to remove unwanted spaces (and occasionally other characters) from either the beginning of a string, the end of a string, or both. In reference-style documentation like that at the back of this book, you might see it summarized thus:

```
STRIP(string, [mode], [pad])
```

Optional arguments

The square brackets around the second and third arguments indicate that the *mode* and *pad* arguments are optional. Usually this means that if you do not supply these arguments, default values will be used to supply the lack, and that is the case here. Some functions, however, such as SHOW, may behave quite differently depending on the presence or absence of certain arguments in the function call.

Mode arguments

If you look further into the documentation for STRIP, you'll find that the *mode* argument, if given at all, must be one of the three letters B, L or T, which stand for 'Both', 'Leading' and 'Trailing' respectively. If you omit this argument, the default value (which happens to be B) is used. If you prefer, you can use the full word rather than just the initial. You can put it in quotes or not, as you like, and use either upper case, lower case or both. Giving any value other than the ones allowed for this argument will cause the script to halt with an error message, however.

Single letter option or mode arguments like this are very common in the ARexx function libraries. For a few of the many other examples, consult the reference documentation on DATE, TIME and SHOW.

The *pad* argument is typical of many of the ARexx string functions. If you omit the argument, it defaults to the space (blank) character.

By the way, it's important to remember when reading function descriptions that the arguments don't have to be literal strings or numbers. Variables and expressions, even including other function results, may be used instead: it's the *final value* of the argument after expression evaluation has been applied to it that is passed to the function.

Using the STRIP function

Now let's use the *Dialog* tool introduced in the last chapter to find out how the STRIP function handles a variety of argument combinations. First, a dry run that doesn't call STRIP:

```
->"<" || "  text  " || ">"
<  text  >
```

Now let's see what happens when STRIP is applied with default arguments:

```
->"<" || strip("  text  ") || ">"
<text>
```

We see that both leading and trailing spaces have been removed from the string. Next we experiment with the three modes:

```
->"<" || strip("  text  ", B) || ">"
<text>
->"<" || strip("  text  ", L) || ">"
<text  >
->"<" || strip("  text  ", T) || ">"
<  text>
```

As expected, the first of these three lines duplicates the default, while in the others the leading and trailing spaces are stripped respectively. What if we now specify values for the *pad* argument?

```
->"<" || strip("  text  ", L, " ") || ">"
<text  >
->"<" || strip("  text  ", L, "*") || ">"
<  text  >
->"<" || strip("**text**", L, "*") || ">"
<text**>
```

Again, the first test simply makes the default value of the final argument explicit. In the second, we are asking the function to remove leading *asterisks*, rather than spaces. Of course, there aren't any, so the call has no effect. But in the third line there are asterisks to remove and the result is as expected.

Place-holding commas

Suppose we wish to give the *pad* argument, but stay with the default for the *mode*. This is perfectly all right, but somehow we have to show that our custom pad character is still the third argument, not the second. We do this with a 'place-holding comma', as in this example:

```
->"<" || strip("### text### ", , "#") || ">"
< text### >
```

As these examples show, ARexx is very flexible in its use of function arguments. We encourage you to use *Dialog* and test scripts in further

experiments of your own with the built-in functions, based on the documentation in the Reference Section.

Where do functions come from?

We have just looked at some of ARexx's built-in functions. Every ARexx script has access to this set of functions, which gives it special importance. The functions themselves are contained, along with the other built-in facilities of the language—instructions, expression evaluator and so on—in the library *rexxsyslib.library*, which as you know must normally be in your *libs:* directory to use ARexx (the exception is if you use the script 'Start-ARexx' on the ARexx release disk).

Function libraries

Another avenue by which functions are made available is through *function libraries*, which may come from several sources. One, *rexxsupport.library*, comes as part of the ARexx system; most of the functions it provides, unlike the built-in ones, are of interest mainly to experienced ARexx programmers. Some function libraries have been created by ARexx users and made available for others to use, usually on a freely redistributable basis through no charge or low charge channels like the 'Fish disks'. An example is *rexxmathlib.library*, which provides advanced mathematical functions (trigonometric and logarithmic functions, for example) that aren't part of the core ARexx language.

Because a function library is stored as a separate file on disk, ARexx must be informed of its existence before the functions it contains are accessible to scripts. Either the *rxlib* program or the ADDLIB built-in function can be used to do this. We will demonstrate both methods later in this chapter.

Function hosts

Functions may also be provided by *function hosts*, which differ from function libraries in a somewhat abstruse way: they run as separate tasks from the function-calling script. With function libraries, the linkage with a script is more direct: no task-switching is involved when a function is called. A function host may be in the form of a program that you run or it may be provided as part of an application program. Gold Disk's *HyperBook* is an example of an application that contains a built-in ARexx function host.

External functions

A function written in ARexx may be stored as a separate file (like a script) and accessed through its file name. Try this tiny example, which you can save as *rexx:SimpleFunc.rexx*:

```
/* Simple function */
return 'Functions can return results!'
```

That's it. Now from the Shell, give this command:

```
Shell> rx "say SimpleFunc()
```

Internal functions

It makes sense to devote a separate file to a function only if that function is going to be called by a number of separate scripts (and not even then if efficiency is a critical concern, since the file must be loaded into memory each time the function is called). User-written functions most often are created to answer the needs of one particular script. ARexx, which is nothing if not versatile, also lets you include functions within the same file as the script that calls them. These are known as *internal functions*. Apart from the built-in functions, in fact, this is the most common form in which ARexx functions appear.

How ARexx locates functions

As we have seen, there are five sources of functions:

- *Built-in functions* are part of the ARexx kernel, *rexxsyslib.library* in the system *libs:* directory.
- *Function libraries,* such as the vendor-supplied *rexxsupport.library* and such freely-redistributable libraries as *rexxmathlib.library,* provide sets of functions that are accessible to ARexx scripts via a direct link mechanism.
- *Function hosts,* such as the host built into Gold Disk Inc.'s hypermedia *HyperBook* program, are somewhat similar to function libraries but use a less direct link mechanism and are normally distributed as stand-alone programs rather than as modules in the *libs:* directory.
- *External programs,* like our very simple example *SimpleFunc.rexx,* are ordinary scripts that are invoked as functions using the file name as the function name.
- *Internal functions* are part of the same source file as the currently executing script.

The Library List

Despite some fundamental differences, function libraries and function hosts are enough alike that ARexx keeps track of them as a single group, on what is known as the *Library List*. An important feature of the Library List is that the ordering of the list determines the order in which the currently open libraries and hosts are scanned when ARexx is trying to match a particular function name. (How to specify the ordering, or *priority*, of a function library or host, is something we'll cover later on.) If more than one library/host has functions of the same name, the one that will be executed is the one belonging to the library/host with the higher priority.

Library search order

Of course, there are other possibilities for 'name collisions' as well. With so many different places it can look to locate functions, ARexx needs a well-defined general mechanism for adjudicating between like-named functions. The core of this mechanism is the *search order*. Every time ARexx encounters a function call in a script, it tries to match the name by looking at each of the possible locations in this order:

1) Internal functions (within the script)

2) Built-in functions

3) Function libraries and hosts (ordered by priority on the Library List)

4) External programs

As soon as a match for the function name is found the search is called off. This means that a function named REVERSE in your script takes precedence over the built-in REVERSE function, which in turn takes precedence over a function of that name from any other source.

Skipping internal functions

Even this is not quite the whole story. If you call the REVERSE function, but put the function name in quotes—that is, make it a string rather than an ordinary symbol—the internal functions will be dropped from the search order. The REVERSE function in your script will be ignored and the one in the built-in library used instead. In Chapter 9 we'll see one way of exploiting this initially curious-sounding feature.

Search order for external functions

For completeness, we should point out that there is also an ordering within the last search location—the external programs. Let's suppose you call a function by the name of FITZROY and that ARexx has been unable to locate a function of that name in your script, among the built-in functions or in any active function library or function host. Before it abandons the search, it will look for a script it can execute, trying each of these file names in turn:

1) `fitzroy.rexx`

2) `fitzroy`

3) `rexx:fitzroy.rexx`

4) `rexx:fitzroy`

If at this point the FITZROY function has still not been located, ARexx at last gives up and reports *Function not found.*

Loading a function library

When a script needs access to the functions contained within a particular library, other than the built-in library, it must load the library from disk if it is not already in the Library List. In this section we'll load in the ARexx support library, *rexxsupport.library*, and call the SHOWDIR function it contains. We can do it all from *Dialog*.

Checking if a library is in the List

To begin, we should find out if the library is already in the Library List. For this we use the SHOW function:

```
->show(L,'rexxsupport.library')
0
```

SHOW is a multifaceted function whose purpose is to provide information about various resources, including the Library List. We are here using but one of its many capabilities, which are fully described under its entry in the Reference Section. The first argument, 'L', announces that the Library List is the resource we wish to inquire about; the second argument specifies the library of interest. In this case, because the library hasn't yet been loaded, the function returns *false* or zero. If the call to SHOW had returned *true* we could skip the next step, which is:

Adding a library with ADDLIB

```
->addlib('rexxsupport.library', 0, -30, 0)
1
```

If you watch closely for disk activity when this function executes... you won't see any. All ADDLIB does is make an entry in the Library List. The library will actually be loaded later on, the first time a function is called that ARexx can't locate by the time the new Library List entry is encountered during the function name search.

Of the three numeric arguments to ADDLIB, the first is of the most practical interest, since it sets the priority of the Library List entry and hence the position of the library in the function search order compared to other libraries and function hosts. The larger this number is within its allowed range of -100 through 100, the higher is the library's priority. The other two arguments are always the same for any given library and should be specified in the library documentation. For an explanation of what they mean and for more details about ADDLIB, consult the Reference Section at the back of the book.

Adding a library with *rxlib*

Another way to get a library's name onto the Library List is to use the *rxlib* program from the AmigaDOS Shell. You have to supply exactly the same information as you would when calling ADDLIB from a script, however the format is naturally a little different. For example:

```
Shell>rxlib rexxsupport.library 0 -30 0
```

The *rxlib* program is handy in some circumstances, but normally it is best to add function libraries explicitly in the scripts that you write.

Using SHOWDIR

We're now ready to use SHOWDIR, one of the functions in the support library. SHOWDIR takes three arguments:

```
SHOWDIR(directory, [mode], [pad])
```

As you see, the argument list is very similar to that for the STRIP built-in function discussed earlier. The action of SHOWDIR is to gather the names of all the files (mode F), or the directories (mode D), or both (mode A, for 'all', which is the default mode) in the given directory and return a string consisting of these names separated by the *pad* character, for which the default is a space. Try it on your *sys:* directory:

```
->showdir("sys:")
```

What that *should* do is display a long list of file and directory names, the contents of *sys:*, separated by spaces. The information, though not the formatting, would be the same if you gave the Shell command:

```
list sys: quick nohead
```

If anything went wrong, it was likely to be either this:

```
->showdir("sys:")
*** Error: Function not found
```

or this:

```
->showdir("sys:")
*** Error: Requested library not found
```

The first case may mean that you have neglected to call ADDLIB for *rexxsupport.library* since the last time you rebooted. The other possibility is that you have removed it from the Library List, or that some script has done so, like this:

```
->remlib('rexxsupport.library')
1
```

Either way, the remedy is simply to invoke ADDLIB now, as described above, then retry SHOWDIR.

Locating the function

To understand the second case (*Requested library not found*), let's reconstruct what ARexx did to process your SHOWDIR call. To begin with, following the search order, it looked (in the *Dialog.rexx* script) for a function called SHOWDIR; not finding one, it then looked in the built-in library, again without success. Now it turns to the function libraries and function hosts named on the Library List, coming eventually to the name you supplied via ADDLIB. Seeing that the library for that entry has not yet been loaded, ARexx goes looking for it in your *libs:* directory, where it *must* find a matching file name. But even that is not enough. Within the library itself the name is given again and that too must match; unlike the file name match, this one must also match in case—*REXXsupport.library* will not match *rexxsupport.library* (the correct name), for instance.

To cope with the error, then, we should first call REMLIB, as shown above, to remove the incorrect entry from the Library List, then call ADDLIB again, this time using the correct name for the library.

Assuming now that everything is working, let's try SHOWDIR again. This time we'll just ask for directories (using the D mode instead of the default A) and separate the names not with a space but with a linefeed character. As you saw at the end of Chapter 5, we ask for a linefeed with the special string **'0a'x**:

```
->showdir('sys:','Dir','0a'x)
```

Summary

We've now looked at the basic components of expressions and met a few of the many functions in the built-in library. In the remaining chapters of this section we'll encounter more of the built-in functions. We'll also, starting in Chapter 8, begin writing functions of our own.

Meanwhile, continue to experiment. Although we will not be using the *Dialog* tool in future chapters, it can continue to be useful when you want to check out some ideas about expressions, or see for yourself exactly how a function described in the Reference Section behaves before trying it out in a script.

Chapter 7
Compound Statements and Loops

So far we have looked at only a few ARexx instructions: the SAY instruction for output to the screen, PULL for input from the keyboard, the assignment instruction for giving a value to a variable and the DROP instruction for 'uninitializing' a variable. We have also briefly covered the more complex IF instruction, which permits the result of a boolean expression to govern whether some subordinate instruction will be executed. We begin this chapter by examining IF more closely.

IF, ELSE and Compound Statements

IF

Here is an example of the simplest form of IF:

```
if partners ~= 0 then
   say "Each person's share is $" || income/partners
```

This much we have already seen. A first complexity arises when we want to take some special action when the value of the *partners* variable *is* zero. Here's an approach that works:

```
if partners ~= 0 then
   say "Each person's share is $" || income/partners
if partners = 0 then
   say "Invalid value for 'partners'."
```

ELSE

This is unnecessarily inefficient, however. Although it looks as though two different tests are being applied to *partners*, really there is only one, but a different action is being taken according to whether that test evaluates to *true* or *false*. We can avoid duplicating the test if we use the *else* keyword, like this:

```
if partners ~= 0 then
   say "Each person's share is $" || income/partners
else
   say "Invalid value for 'partners'."
```

Now let's consider a similar complication. This time, we want to perform not just one action when *partners* is non-zero, but two (or maybe more). A first approach again is to use multiple IFs:

```
if partners ~= 0 then
  say "Each person's share is $" || income/partners

if partners ~= 0 then
  say "This is 1/"partners "of the total $" || income
```

Grouping with DO and END

Here the redundancy of the second test is even more obvious than in the previous example. The solution lies in the concept of a *compound instruction*: a sequence of instructions that are somehow bound together to appear from the 'outside' as a *single* instruction. You can tell ARexx that a group of consecutive instructions is to be regarded as a single unit by bracketing them with the special keywords DO and END. They are as easy to use as bookends:

```
if partners ~= 0 then do
  say "Each person's share is $" || income/partners
  say "This is 1/"partners "of the total $" || income
  end
```

Naturally, DO and END can be used to extend the range of ELSE as well:

```
if partners ~= 0 then
  say "Each person's share is $" || income/partners
else do
  say "Invalid value for 'partners'."
  say "The number you supply MUST be non-zero!"
  end
```

Nested IF statements

By the way, the instruction that depends on an IF or ELSE can be of any type: even another conditional like IF/THEN itself. One important result of this is that you can chain IF/THEN/ELSE instructions together to deal with several different alternatives:

```
if input = 604 then
  say "British Columbia"
else
  if input = 208 then
    say "Idaho"
  else
    if input = 601 then
      say "Mississippi"
    else do
      say "Sorry, I don't recognize that area code."
      say "Frankly, I'm skeptical that it is valid."
      end
```

ELSE IF

A sequence of instructions like this illustrates that IF/THEN/ELSE forms a single unit: everything after the first ELSE is subsidiary to it and can be regarded as a single instruction. The indentation reflects this in the standard way. However, this kind of construction is common enough to have evolved its own convention for indentation, one that reflects the logic from the point of view of the human programmer rather than the language:

```
if input = 604 then
   say "British Columbia"
else if input = 208 then
   say "Idaho"
else if input = 601 then
   say "Mississippi"
else do
   say "Sorry, I don't recognize that area code."
   say "Frankly, I'm skeptical that it is valid."
   end
```

SELECT... WHEN

Code fragments like this are so common, in fact, that a special compound instruction called SELECT is provided to accommodate it. The following example of its use is essentially equivalent to the previous example using IF/ELSE. The only real difference is that the SELECT construction makes the purpose of the code even more explicit, especially when a large number of alternative cases is being considered.

```
select
   when input = 604 then say "British Columbia"
   when input = 208 then say "Idaho"
   when input = 601 then say "Mississippi"
   otherwise
      say "Sorry, I don't recognize that area code."
      say "Frankly, I'm skeptical that it is valid."
   end
```

As with the IF/ELSE chain, SELECT considers a number of candidate expressions, executing the associated instruction when one evaluates as *true*. The expressions are introduced with WHEN rather than IF, but the instruction is introduced with THEN as before. As usual, the dependent instruction can be compound, using DO and END, if desired. The place of the final ELSE, whose associated code is executed when all tests have failed, is taken by OTHERWISE. Observe that OTHERWISE may have multiple dependent instructions, even without an explicit DO/END.

OTHERWISE errors

One difference between the OTHERWISE in SELECT and the final ELSE of an IF/ELSE chain is that the latter is entirely optional: ARexx does not care if you put in code to cover this clean-up contingency or not. It is a potential error, however, to omit the OTHERWISE case in a SELECT instruction. The error becomes actual when, during a program run, all the WHEN tests fail. ARexx then insists on the presence of an OTHERWISE and will halt the program if it doesn't find one.

A note on the formatting of conditionals and compound instructions

Different programmers have different ideas about the best way to format ARexx scripts. All experienced programmers agree, however, that *some* sort of orderly formatting is necessary if scripts are to be easily readable. The primary means of formatting is *indentation*. The underlying idea of all formatting schemes is that the indentation of each line of code should reflect its place within the logical structure of the script. A block of lines whose execution hinges on a conditional instruction, for instance, should be indented with respect to that instruction. Schematically:

```
if <condition1> then do
   <block of lines dependent on condition1>
   end
else if <condition2> then do
   <block of lines dependent on condition2>
   end
else do
   <block of lines for default case>
   end
```

There are other ways of formatting an IF-ELSE in ARexx; this common method happens to be the one we have adopted. Other kinds of constructs such as functions, DO-loops and SELECT conditionals present similar opportunities for bringing out the meaning of a script in this graphic way. Observe the formatting in this book and use it as a basis for a standard approach of your own, even if it doesn't happen to be quite the same as we prefer. The important thing is to be logical and above all consistent.

Loops

DO loops

The *conditional instructions*, IF and SELECT, give ARexx one kind of power, that of making decisions based on inputs that are unknown at the time the program is written. Another kind of power involves computing brawn rather than brain: it is the power to perform the same operation over and over again in exactly the same way. If computers could get bored we would be in serious trouble, so heavily do we depend on their capacity for tireless repetition. In ARexx, we tap that capacity with DO, an instruction we met in the previous section. For a first look at the several variations on this theme, try this little script:

```
/* First DO-loop - print 2**1 through 2**5 */
n = 1
do 5
   n = n * 2
   say n
   end
```

As you can see by running the script, the two instructions:

```
n = n * 2
say n
```

are executed not once but five times, exactly as though we had executed this simple-minded script:

```
/* Emulating a loop */
n = 1
n = n * 2
say n
n = n * 2
say n
n = n * 2
say n
n = n * 2
say n
n = n * 2
say n
```

This script is conceptually simpler than the previous one and in a way more obvious, but few would argue that it is easier to read. The limitations of this 'in-line' approach become even clearer when you think about the in-line script that would equate to:

```
/* A loop with more repetitions */
n = 1
do 100
   n = n * 2
   say n
   end
```

And just in case you're not yet convinced that there's no future in in-line coding, think about how you'd handle this without loops:

```
/* A loop with variable repetitions */
say "Enter number of repetitions
parse pull reps
n = 1
do reps
   n = n * 2
   say n
   end
```

Now let's try another script that is an obvious candidate for a loop. This one prints a table of the squares and cubes of the numbers 1 through 10 (recall from Chapter 5 that ** is the ARexx exponentiation operator, so the **n**3** in this script means 'n cubed'):

```
/* Table of squares and cubes */
n = 1
do 10
   say n n*n n**3
   n = n + 1
   end
```

DO FOREVER

If you try this script you'll find that the output is rather ragged, since we have made no attempt to make the three numbers output on each line appear in columns. This problem is easily solved, as you'll find when we look at string functions in a later chapter, but we'll just put up with it for the present (though feel free to experiment with solutions of your own). Now let's consider some interesting variations on the theme of loops. For starters, following DO with the special word FOREVER has the result you would expect: the loop will execute an unlimited number of times, terminating only when it is interrupted for some other reason. Such as? There are a number of possibilities, but in normal cases the loop will terminate when one of these instructions is executed:

```
leave
break
exit
return
signal
```

LEAVE

For now we'll go with LEAVE, which does exactly what we want and no more: breaks out of the loop and continues with any subsequent instructions. Of course, if we simply add LEAVE to our existing loop, it will terminate on the very first repetition, which would be rather pointless. We must make the LEAVE depend on some condition that will evaluate as *true* only when it suits our purposes. For instance:

```
/* Table of squares and cubes */
n = 1
do forever
   say n n*n n**3
   n = n + 1
   if n > 10 then leave /* escape the loop */
   end
```

Loop indexes

This script behaves exactly like the last one, but it has one important difference: in the present script, the variable *n* has a special role, that of an *index* or *loop counter* variable. It has a starting value (1) and an amount by which it is augmented (also 1) in each loop repetition. Its value is used within the loop, but—apart from the systematic increment—never modified. This kind of usage is so common that ARexx provides a special DO variant to take care of the bookkeeping automatically. The result is much more compact code:

```
/* Table of squares and cubes */
do n=1 to 10 by 1
   say n n*n n**3
   end
```

The DO line in this script has four parts:

do	instruction keyword
n=1	initializer
to 10	limit
by 1	increment

Limit and increment for index variables

Neither the *limit* nor the *increment* is compulsory. If *limit* is missing, the loop variable will be incremented indefinitely and one would normally expect other arrangements (like LEAVE) to have been made to escape the loop. If *increment* is missing, it is taken to be 1, the value given explicitly in the example.

The index variable, here *n*, is special in that ARexx automatically updates it on each pass through the loop and tests it against the value computed from the *limit* expression if one has been provided. In other respects it is an ordinary variable, not segregated or protected in

any way from the common herd. In particular, you can modify the index variable yourself within the loop if you wish, thereby possibly affecting the number of loop repetitions:

```
/* Modifying the index variable */
do counter=1 to counter+9 by counter+counter
   say counter
   if counter = 3 then counter = 6
   end
```

Though ARexx allows you to interfere with the normal operation of a loop index variable in this way, the practice is universally discouraged: it is untidy and makes your code more difficult to understand; moreover, it is hardly ever necessary.

Another observation we can make about this last script is that the *limit* and *increment* expressions are evaluated only once, before the loop is first entered, and the values obtained are used from then on (a good thing in this case, since the script would otherwise not terminate).

We have now seen three ways of controlling the repetition count of a DO loop; we can represent them with these examples:

- `do 5`
- `do forever`
- `do i=1 to 20 by 2`

We have also seen that in the third of these types, the TO and BY parts of the instruction are optional. Besides these forms of DO—the *iterative* forms—we recall from the beginning of the chapter that DO used all by itself introduces an ordinary compound instruction with no looping.

Conditional loops: WHILE

Now let's turn to the two *conditional* forms of DO loop, forms that resemble IF/THEN in that the decision to execute the instructions in the loop depends on the value of a boolean expression. The use of the conditional forms is just about self-explanatory, as you'll see from this example using the commoner of the two, WHILE:

```
/* List the powers of two less than 1000 */
i = 0
n = 1
do while n < 1000
   say "2 to the" i "=" n
   n = n * 2
   i = i + 1
   end
```

Before the DO-loop is entered for the first time, the variable *n* is compared to 1000. Since its initial value is less than 1000, the boolean expression returns *true*—the condition is satisfied—and we enter the loop. If the initial value of *n* were 1000 or more, the instructions in the loop would never be executed at all. At the end of the first pass through the loop, *n* has been doubled, but its value still satisfies the WHILE condition, so the loop is entered again. And so on after the second pass and the third and right up to the tenth. At the end of the tenth pass, however, *n* has a value of 1024, the controlling expression returns false at last and ARexx skips over the loop to resume execution at whatever instruction follows the END, if any.

UNTIL

In a WHILE loop, the controlling condition must be *true* at the beginning of each iteration of the loop (including the first) if the loop is to continue executing. An UNTIL loop differs in a very symmetrical way: it requires the controlling condition to be false at the end of each iteration if the loop is to continue. A consequence of this is that an UNTIL loop always executes at least once. Here is an example of an UNTIL loop:

```
/* Wait for palindromic input */
do until in = reverse(in)
   say "Please enter a palindrome:"
   parse pull in
   end
```

Now that we have the simple form of DO, three iterative forms and two conditional forms, you may wonder if any further variations are possible. The answer is yes. First, any of these forms may take an extra phrase consisting of the word FOR followed by a numeric expression. Applied to the simple form of DO, this merely creates the same kind of loop that would result if the FOR were absent. That is:

```
do for 12
```

is effectively identical to:

```
do 12
```

You may wonder what happens in this case:

```
do m for n
```

The answer is that the FOR takes precedence and the loop will execute *n* times. In all other cases, the effect of FOR is to set a limit on the number of iterations that will be performed, regardless of other means of controlling the loop. All the following loops will run for exactly six iterations:

```
/* A batch of 6-iteration loops */
do for 6
   say "#1"
   end
do 3 for 6
   say "#2"
   end
do 12 for 6
   say "#3"
   end
do forever for 6
   say "#4"
   end
do i=1 to 1000 by 3 for 6
   say "#5 ("i")"
   end
low = 1
hi  = 1000
do for 6 while low < hi
   say "#6 ("low"/"hi")
   low = low + 3
   end
low = 1
hi  = 1000
do for 6 until low >= hi
   say "#7 ("low"/"hi")"
   low = low * 2.3
   end
```

DO combinations

And now, if you're ready for DO's grand finale, we have just one more major point to cover: any of the iterative formats of DO can be combined with either of the conditional formats and/or with FOR, to produce loops with multiple restrictions. Take a look at this:

```
/* The guessing game */
answer = random(1,1000,time('s'))
say "I am thinking of a number between 1 and 1000..."
do turn = 1 to 10 until guess = answer
   say "Your guess?"
   parse pull guess
   if answer > guess then
      say "Too low."
   else if answer < guess then
      say "Too high."
   end
if guess ~= answer then
   say "You ran out of guesses! The answer was" answer
else
   say "You got it in" turn "turns!"
```

In this guessing game an iterative loop is used for the turn counter, with the maximum set for ten turns (which is the most turns you should require if you play perfectly). But that maximum is only one of the conditions for exiting the loop; the other is if the player makes a correct guess, which we check with the UNTIL conditional. This

particular script requires a *turn* variable so that it can report how many turns the user needed to win (see the last line). If we decide to discard this feature, changing the last line to:

```
say "You got it!"
```

the DO instruction could be simplified to:

```
do 10 until guess = answer
```

Restrictions on combinations

The only restrictions on combining loop types in this way are:

- No more than one iterative type or one conditional type may be used together in the same loop. That is, you can't say something like:

```
do while n1 < 100 until s1 = 'quit' /* illegal */
```

- When an iterative and a conditional type are used together, the iterative type must be given first. FOR is counted as a component of the iterative type for this purpose. This loop will not work:

```
do while i < 5 for 10 /* won't work! */
```

Using Loops with Compound Variables

The examples of looping that we have already seen hint at a few of the possible uses of this fundamental programming concept. Loops really shine, however, when used with a *composite data type*—which in ARexx means compound variables.

Suppose we have a compound variable called *scores*, with instances from *scores.1* through *scores.100*. Let's find the total of all these scores:

```
/* Total scores - version 1 */
total = 0
total = total + scores.1
total = total + scores.2
total = total + scores.3
   .
   .
total = total + scores.99
total = total + scores.100
```

Well, space doesn't permit the printing of the whole script, but you can probably get the idea from this excerpt. You may also have visualized a much more concise and elegant way of tackling the problem, something like this:

```
/* Total scores - version 2 */
total = 0
do i=1 to 100
   total = total + scores.i
   end
```

Apart from the fact that this is a very much smaller script, which is a good thing in itself, it is obviously also much more flexible. If we suddenly had to deal with two hundred scores rather than one hundred, the change would involve editing a single number—the loop count—rather than typing in another hundred lines of code. In fact, it would be a simple matter to change the loop count from a hard-coded number to a variable, which would be initialized somewhere else in a larger script containing this loop. The same code would then work no matter how big a table of scores was involved.

Converting characters to numbers

With a little extra work, you can use a loop to step through the elements of a compound variable in which the element names are letters rather than numbers. The secret lies in the functions C2D and D2C, which inter-convert between letters and the numbers that are used to represent them in the ASCII character set. For instance, the letter 'A' is represented by the number 65, 'B' by 66 and so on up to 'Z', which is 90. The lower case letters are similarly represented by the numbers from 97 to 122. The non-alphabetic characters, including 'control' characters like TAB and RETURN, also have equivalents. This little script demonstrates how to convert characters to their numeric codes:

```
/* Convert first letter of input string to ASCII */
do until c==''
   say "Enter a character (RETURN to quit)"
   parse pull c
   if c ~== '' then
      say "The numeric code is" c2d(left(c,1))"."
   end
```

The LEFT function, one of the many string-handling functions in the built-in library, is the only new feature of this script apart from C2D itself. We are using it here to isolate the leftmost character of the input string *c*.

Some of the library string functions are used so commonly that you will see them again and again. Another example is SUBSTR, which we use in the next script to extract all the letters of a string one at a time:

Example: counting letter frequencies

```
/* Count letter frequencies in input string */
say "Enter a line of text."
pull line
counts. = 0     /* Set all counts to 0 */
 /* First loop builds table of counts */
do i=1 to length(line)
   c=substr(line,i,1)
   if c >= 'A' & c <= 'Z' then
      counts.c = counts.c + 1
   end
/* Second loop reports counts */
do i=c2d('A') to c2d('Z')
   c=d2c(i)
   say c':' counts.c
   end
```

This script counts the occurrences of each letter in an input string and reports the frequencies to the user in the form of a table. The program has two loops. In the first, which iterates from 1 through to the number of characters in the string (as determined by the LENGTH function), we add one to element *c* of the compound variable *counts* for each alphabetic character *c* in the input string. The second loop iterates from 'A' to 'Z', or rather from the ASCII equivalent of 'A', 65, to the equivalent of 'Z', 90. The equivalents are determined in the program using C2D. The index variable is then changed back to a character, with D2C, and used to access the stored count for that character. The point of this second loop would be more neatly expressed if we could code this:

```
do c='A' to 'Z'
   say c':' counts.c
   end
```

Reasonably enough, however, ARexx insists that the loop index should be numeric, so the extra conversions with C2D and D2C are necessary.

Chapter 8
User-Written Functions

We learned in a previous chapter that ARexx includes an extensive library of *built-in functions* and that other function libraries are available for programs that need their more specialized services. No matter how many existing functions you may have at your disposal, however, if you do much programming you will often require functions that are not in any library. Some may be functions that meet the unique requirements of a particular script and will be needed only once. Others may answer some more general need in several or many of the scripts you write and you may—perhaps with minor variations for particular purposes—use them again and again.

The Anatomy of a Function

Defining a simple function

Here is a script that defines and uses an internal function named PI. Although we haven't formally covered the syntax of internal functions, you should have no difficulty understanding what is going on:

```
/* A PI function */
say pi()
exit
pi:
   return 3.141592654
```

The essential ingredients of any ARexx function are a *label clause* and a RETURN instruction. We have both here. A label clause is defined simply as a valid ARexx name (formed by the same rules as variable names) followed by a colon. Its purpose is to mark a position within a script. The label determines both the name and the starting line of an internal function.

Transfer of control

Calling any function, whether internal or external, involves a temporary *transfer of control.* For the most part, ARexx processes each clause of a script in sequence, from beginning to end, just as someone filling out a form might methodically process each line from top to bottom. When a function is invoked, however, this sequential processing is disrupted. ARexx must look for its next instruction not in the next clause of the script, but in some other location, either

elsewhere in the same script or external to it. In the analogy of filling out a form, this would correspond to carrying out an intermediate calculation elsewhere on the same form, or on a different form, then bringing the value of that calculation back to complete the current line.

The RETURN instruction

One duty of the RETURN instruction is to restore control to the instruction from which a function was called. RETURN's other duty, which is not always required, is to bring back a value for use by the calling instruction. In the example above, the calling instruction is **say pi()**; in this case, a return value from the function is needed.

EXIT versus RETURN

The EXIT instruction in our example causes a different kind of transfer of control. Whereas RETURN always causes a transfer of control back to the immediate caller of the presently executing function, EXIT always terminates the presently executing *script*. You may recall from Chapter 6 that an external script can actually be executed as a function. In that case, RETURN and EXIT mean exactly the same thing and either can be used to return a value. For example:

```
exit 100
```

and:

```
return 100
```

are exactly equivalent ways of terminating an external function.

A script that is executed as a command—with *rx*, for instance—also returns a value. With *rx*, any returned value must be an integer and it is passed back as the *return code* from the command.

Without the EXIT instruction in the PI script, the lines below EXIT would be executed in their due sequence. Since the PI function was not written to be part of the *mainline* of the script, this would be undesirable and in this particular case would actually constitute an error. The error would not come with the execution of the label *pi:*—executing a label in this way has no effect at all—but with the attempt by the RETURN instruction to return a non-integer value to *rx*.

Function side effects

As mentioned in Chapter 6, functions may be called for their so-called *side effects* as much as for the value they return. Some functions may return no value whatsoever—side effects are the only effects they have! Here is a script containing such a function:

```
/* Starline.rexx */
call starline()
exit
starline:
   say "********************"
   return
```

Functions without values—using CALL

There are three points to be noted here. The first is the CALL instruction, which provides a means of calling a function that does not return a value. If you tried to use such a function in an expression in the usual way, ARexx would report an error, since it requires a value to substitute for the function call when the expression is evaluated. You can also use CALL when only the function's side effects, and not any result it may return, are of interest. Another interesting point about using CALL to invoke a function is that the parentheses are not required around the function arguments when you do so. There's no special reason to leave the parentheses out—we recommend you use them for consistency—but it's worth remembering this fact in case you encounter the syntax in a script written by someone else.

The second point about the STARLINE script is that though the RETURN instruction appears all by itself, with no associated value, it is still needed to pass control back to the caller.

Function Arguments

Trying for generality

And the third point is that STARLINE is a pretty feeble function. Though it does its job efficiently and reliably—it almost certainly does not contain a 'bug'!—it's far from versatile. Such a function is said to lack 'generality', meaning that it is useless except for whatever special purpose prompted its creation. There are a number of obvious ways to make it more general:

- Let the caller specify the number of asterisks to be printed.
- Let the caller specify the character to be printed.
- Return the string of characters to the caller as the function value, rather than printing them as a side effect. That way the caller can do further processing on the string, if desired, before outputting it.

From STARLINE to CHARLINE

These all sound like good ideas, so let's implement them. We want to end up with a function (we'll call it CHARLINE now instead of STARLINE) that we can use like this:

```
say charline("+", 25)
```

It's pretty clear that the CHARLINE function is going to be based on a loop and that the number of loop iterations will be controlled by a variable. In fact, if we just wanted to write a piece of code to create a string consisting of *length* repetitions of the character *char*, we should be able to do so quite simply based on what we already know of loops:

```
line = ''    /* Initialize the result */
do length
   line = line || char
   end
```

And since we'll want to pass the result back to the caller, we should add:

```
return line
```

There's just one snag: we don't really have any variables called *length* and *char*. What we have instead are the function arguments "+" and 25. But how can we refer to them inside the function? They don't have names!

Reading arguments with PARSE ARG

There are two ways to get at the arguments. The more convenient uses the PARSE ARG instruction we have already met. As you recall, a line like:

```
parse arg height
```

will cause the command line argument to an ARexx script to be stored in the *height* variable. Unlike scripts invoked as commands with *rx*, functions can take multiple arguments. Using PARSE ARG, we can read them into a series of comma-separated variables like this:

```
parse arg height, width, depth
```

For the CHARLINE example, we could read the arguments into the *char* and *length* variables like this:

```
parse arg char, length
```

PARSE ARG is very convenient and often the best way to go. Sometimes, however, it is preferable to use a more powerful tool: the built-in function ARG. We'll be using ARG's special capabilities in some of the examples in this chapter, so we'll defer further

exploration of PARSE until Chapter 10, but you should become familiar with both.

Reading arguments with the ARG function

To start, here's a script whose sole purpose is to demonstrate ARG's powers:

```
/* A demonstration of arg() */
call test_func(1,,"banana")
exit
test_func:                              /* Output */
   say "Arg()      :" arg()             /* Arg()      : 3  */
   say "Arg(1)     :" arg(1)            /* Arg(1)     : 1  */
   say "Arg(2)     :" arg(2)            /* Arg(2)     :    */
   say "Arg(3)     :" arg(3)            /* Arg(3): banana */
   say "Arg(4)     :" arg(4)            /* Arg(4)     :    */
   say "Arg(1,'e'):" arg(1,'e')         /* Arg(1,'e'): 1  */
   say "Arg(2,'e'):" arg(2,'e')         /* Arg(2,'e'): 0  */
   say "Arg(1,'o'):" arg(1,'o')         /* Arg(1,'o'): 0  */
   say "Arg(2,'o'):" arg(2,'o')         /* Arg(2,'o'): 1  */
   return
```

As you have probably surmised, ARG can give you a variety of information about the arguments supplied to the currently executing function. ARG with no arguments, as in the first SAY instruction above, returns the number of arguments available—the length of the argument list—here three. If a number is given, as in the succeeding four instructions, ARG returns the value of the corresponding argument. If the argument was not supplied, as is the case for arguments 2 and 4, a null string is returned. Finally, if a number is given along with either 'e' (for 'exists') or 'o' (for 'omitted'), a boolean value is returned according to whether a value was given for the corresponding argument.

Taken together, the capabilities of ARG let you create functions that can handle any number of arguments and optionally provide default values for arguments that aren't supplied by the caller.

Incorporating ARG into CHARLINE

Now we can finish coding the CHARLINE function. Try this yourself:

```
/* The charline function */
say charline("_", 25)
say charline("|", 25)
say charline("+", 25)
exit
charline:
   char   = arg(1)
   length = arg(2)
   line = ""   /* Initialize the result */
   do length
      line = line || char
      end
   return line
```

This produces a stunning graphic effect... well, a graphic effect:

```
_________________________
|||||||||||||||||||||||||
+++++++++++++++++++++++++
```

A happy consequence of changing over to variables from the hard-coded string of asterisks is that we are no longer restricted to repeating only a single character. Try changing the 'mainline' of the above script to something like this:

```
/* The charline function */
say charline("/\", 12)
say charline("\/", 12)
exit
```

This generates:

```
/\/\/\/\/\/\/\/\/\/\/\/\
\/\/\/\/\/\/\/\/\/\/\/\/
```

Rectangles with CHARLINE?

The CHARLINE function has now acquired the prized trait of generality that we had in mind. Or has it? Suppose we found that in a particular script we frequently required multiple lines formed from the same repeating sequence of characters. This might be used in creating rectangular patterns like:

```
::::::::::::::
::::::::::::::
::::::::::::::
```

Obviously this could be achieved by calling CHARLINE as many times as necessary in succession, probably in a loop and concatenating the results (adding a line feed character so that the next string will appear on a different line), as in this fragment:

```
rectangle = ''
width     = 14
height    = 3
CR        = '0a'x    /* Linefeed */

do height
   rectangle = rectangle || charline(':',width) || CR
   end
```

Or we could add another argument to the CHARLINE function specifying the number of lines we want it to generate. The argument could even be optional, defaulting to one. In the default case we'd omit the linefeed, producing the same behavior as the original function. That way, we could be sure that any programs we'd written

incorporating the original function would continue to work. The new CHARLINE would look like this:

```
charline:
   char   = arg(1)
   length = arg(2)
   line = ''   /* Initialize the result */
   do length
      line = line || char
      end
   if arg(3,'e') then do
      rectangle = ''
      CR        = '0a'x
      height    = arg(3)
      do height
         rectangle = rectangle || line || CR
         end
      end
   else
      rectangle = line
   return rectangle
```

Variable margins with CHARLINE?

Our function is obviously getting quite a bit more complicated. This is partly because it now handles two rather different cases, a fact which ought perhaps to start a small warning bell ringing away somewhere in the back of our minds. But let's carry on gamely, because after using this new, more powerful version of CHARLINE, we quickly discover that it has a very serious shortcoming: it is not possible to create a rectangle anywhere except at the very left edge of the output device, which for us at present is the console window. We can create, for example, this:

```
****
****
```

but not this:

```
                                    ****
                                    ****
```

Well, in some situations that might be a serious limitation. To get around it we could write a new, even more powerful version of CHARLINE that takes a fourth argument specifying the number of spaces to put on each line as a left margin. This argument will have a default value of 0 and again it will only apply at all if the number of lines (the third argument) is given. The new version of the function will look like...

The limits of generality

But wait a minute. If we just stop and think where this is leading, that warning bell we heard a minute ago is going to get deafening loud. Because it's becoming obvious that this process of adding arguments and adding features to the once-elegant CHARLINE function could be carried on virtually without limit.

How about if we add an argument to specify a character to use as an outline for the rectangle? With a single function call we could then get effects like this:

```
=======
=*****=
=*****=
=======
```

Yet another argument might be an optional character string to be overlaid on the center of the rectangle, for a 'framed title' effect:

```
$$$$$$$$$$$$$$$$$$$$$
$$$$ MONEY MATTERS $$$$
$$$$$$$$$$$$$$$$$$$$$
```

And so on and so on and so on. But we've come a long way down the garden path and now it's time to retrace our steps, all the way back to this:

```
do height
   rectangle = rectangle || charline(':',width) || CR
   end
```

Nesting function calls

This fragment, on which we based the first major elaboration of the CHARLINE function, works by calling the original, elegant CHARLINE as often as needed to produce a rectangle. That's well and good. But instead of expanding CHARLINE to incorporate this code, we would have done much better to create a new function, DRAWRECT, that calls the existing CHARLINE without trying to improve on it. This is a simple instance of a 'nested' function call. In fact, CHARLINE itself contains a nested call to the built-in ARG function. It is not uncommon for the execution of an ARexx script to entail many layers of nested functions. It is even possible for a function to call itself, either directly or indirectly via some other function; this technique, known as recursion, is almost indispensable for some kinds of programming tasks.

```
drawrect:
   parse arg char, width, height
   CR        = '0a'x
   rectangle = ''
   do height
      rectangle = rectangle || charline(char, width) || CR
```

```
   end
return rectangle
```

Modularity: divide and conquer

Observe that this function no longer attempts to handle the original task of generating a single line of characters with no terminating line feed. And although it is thereby less powerful to some degree, we have really lost nothing, for the CHARLINE function is still available to any caller. We retain generality and have gained a new degree of *modularity*, which is the desirable characteristic of breaking up large complex jobs into small simple ones. The simpler organization is also reflected in the fact that we can now dispense with the special abilities of the ARG function and can return to the simplicity of PARSE ARG in both DRAWRECT and CHARLINE.

Deciding how to subdivide large tasks into smaller functions is not always easy, nor is there always a single best approach. It is one of the many aspects of programming in which art complements science, in which judgement and taste must work alongside rational analysis.

Now let's turn back to the CHARLINE function itself to consider an entirely different type of problem. See if you can find the serious flaw in this script:

```
/* Display input line between two double lines */
say "Enter a line of text:"
parse pull line       /* get line from user        */
say charline("=", 40) /* display first double line */
say line              /* display input text        */
say charline("=", 40) /* display second double line*/
exit
charline:
   parse arg str, reps
   line = ''    /* Initialize the result */
   do reps
      line = line || str
      end
   return line
```

When variable names collide

The problem, of course, is that both the CHARLINE function and its caller are using the variable named *line*—but for entirely different purposes. The mainline's plan is simple: read the user input into *line*, display a row of equals signs, display *line* and finally display another row of equals signs. CHARLINE's use of the variable is also straightforward: set *line* to an empty string, then add as many characters to it as necessary to fulfil the caller's requirement and finally return the value of *line* to the caller. Unfortunately, in the process, CHARLINE will destroy the user input value that the mainline was carefully saving.

This is a classic difficulty in computer programming, one that crops up time and time again in various forms. On this occasion the fault in the program was easy to spot and hence easy to correct—one obvious way would be to change the name of the *line* variable in either part of the program. In a larger program, duplicate variable names can cause problems that are much harder to trace. The fact that some variable names are very common, such as *i* for a loop index variable, aggravates the problem. In very large programs, with many functions that may be invoked directly or by calls nested within other functions, the probability of variable name 'collisions' increases exponentially. The chaos that can result may be all but unmanageable.

Local Variables: PROCEDURE and EXPOSE

Protecting variables with PROCEDURE

Fortunately, there is a solution, which like many good solutions is pretty obvious—especially when you know what it is—though different languages may implement it in very different ways. The solution is to limit the *scope*—the amount of a script or program—in which a variable may be known. In ARexx, it is possible to make any function completely ignorant of all variables declared outside its own borders. This is done with the special instruction PROCEDURE:

```
/* PROCEDURE demo */
say "1." val
val = 1
call setval()
say "4." val
exit
setval: procedure
   say "2." val
   val = 3
   say "3." val
   return
```

Thanks to PROCEDURE, the variable *val* used in the SETVAL function here is not the same as the *val* used by the mainline, as the output demonstrates:

```
1. VAL
2. VAL
3. 3
4. 1
```

PROCEDURE insulates SETVAL from the rest of the script. Variables used outside of SETVAL are not known within it, as line 2 of the output demonstrates. Variables used within SETVAL have no effect

outside it, as line 4 demonstrates. The two sets of variables are completely segregated, so name collisions cannot occur.

Overriding PROCEDURE

The ability to create *local variables* using PROCEDURE gives us the answer, as we have seen, to a serious problem. Unfortunately, it creates a new problem of its own. Sometimes we want to have our cake and eat it too: we want some of the variables used in a function to be private, but we also want to share certain variables with the caller. Fortunately, this too is possible. We follow PROCEDURE with the subkeyword EXPOSE and a list of the caller's variables that we need to access:

```
/* EXPOSE demo */
dish  = "duck"
sauce = "orange"
say "We recommend the" dish "with" sauce "sauce."
call changemenu()
say "You might also try the" dish "with" sauce "sauce."
exit
changemenu: procedure expose sauce
   dish  = "ice cream"
   sauce = "chocolate"
   say "And for dessert, the" dish "with" sauce "sauce."
   return
```

The purpose of the CHANGEMENU function is to revise the menu entirely—both the dish and the sauce. There is a bug, however—only one of the two relevant variables is exposed. As a result, the output from the program is clearly defective:

```
We recommend the duck with orange sauce.
And for dessert, the ice cream with chocolate sauce.
You might also try the duck with chocolate sauce.
```

Some uses of EXPOSE

In general, it is a good idea to use EXPOSE rather sparingly. Sound modular programming demands that data should not be shared unnecessarily by a program's subunits—the functions of an ARexx script. EXPOSE is appropriate in some situations, however, for example:

- To obtain multiple results from a function.
- To allow access to a global information base.
- To allow access to named constants.

Let's look at each of these in a bit more detail. In the process we'll have a chance to cover not only the uses of EXPOSE but some practical aspects of programming functions in general.

Obtaining multiple results from a function

In the 'dish and sauce' example above, the point of the CHANGEMENU function is to provide new values for two variables: *dish* and *sauce*. Because only *sauce* was exposed, *dish* remained unmodified, but that could easily be corrected by modifying the PROCEDURE instruction:

```
changemenu: procedure expose dish sauce
```

Returning multiple results

It often happens that one would like to obtain two values (such as a co-ordinate pair) or even more from a single function call. Though a function is limited to returning a single value, the additional information can be transferred through a variable the function exposes. In the following example, the INTERSECTION function computes the intersection point of two lines in Cartesian coordinate space. The two lines are in the form **y = mx + b**, and the arguments to the function are *m1, b1, m2* and *b2*. The function returns the y-coordinate of the intersection as the function result, and the x-coordinate in the exposed variable *x_coord*. Notice that this variable had no explicit existence for the caller before the function initialized it:

```
/* Compute line intersection */
y_coord = intersection(2,3,3,-1)
say "The lines meet at ("x_coord","y_coord")."
exit
intersection: procedure expose x_coord
   parse arg m1, b1, m2, b2
   x_coord = (b2 - b1) / (m1 - m2)
   return m1 * x_coord + b1
```

Extracting results with WORD

This is not the only way to obtain multiple results from a function. Another way is for the function to return all the results concatenated into a single string, from with they can be extracted in various ways. Using the WORD function from the built-in library, for example, the previous example could be rewritten like this:

```
/* Compute line intersection */
coords = intersection(2,3,3,-1)
say "The lines meet at ("word(coords,1)","word(coords,2)")."
exit
intersection: procedure
   parse arg m1, b1, m2, b2
   x_coord = (b2 - b1) / (m1 - m2)
   return x_coord m1 * x_coord + b1
```

This approach requires a little more work on the part of the caller, but reduces or eliminates undesirable data sharing between the caller and the function.

Accessing a global information base

We have seen how compound variables may be used to set up an information base, like the database of books in the examples of the previous chapter. A script written to work with such a database would normally have a variety of functions for tasks like adding an information record, and deleting, updating, displaying, sorting and storing records. Most of these functions will need access to the database as a whole. Rather than forcing all the functions to share all their variables, it is better to expose the stem of a compound variable in each function that needs it. Notice in the following example that exposing the stem has the effect of exposing every compound variable created from that stem:

```
/* Exposing global information */
planets.mercury.diameter = 4876
planets.mercury.velocity = 47.88
planets.venus.diameter   = 12102
planets.venus.velocity   = 35.02
call planet_stats(mercury)
call planet_stats(venus)
exit
planet_stats: procedure expose planets.
   parse arg p
   say "Planet  :" p
   say "Diameter:" planets.p.diameter "km"
   say "Velocity:" planets.p.velocity "km/sec"
   return
```

Accessing named constants

Some constant values are so important or fundamental that they have commonplace names of their own, the classic example being *pi*. Script readability may often be enhanced by using the name rather than the numeric value in such cases: it surely makes for easier reading and clearer logic if a script refers to *pi* rather than 3.141592654. This is readily possible, as you know, by creating and initializing a variable of that name:

```
pi = 3.141592654
```

Using variables as 'constants'

Some languages provide a special facility for *named constants*, but interpreted languages like ARexx generally do not. In ARexx, a named constant is just a variable that you don't modify. Constants must therefore follow the usual rules of scope for variables, and will not be available inside functions that use the PROCEDURE instruction unless they are specifically exposed.

Provided you take care to use distinctive names for constants, thus minimizing the risk of collision, the gain in readability from exposing them freely throughout a script is usually worth the cost in modularity.

Constants need not be so literally a 'constant' value as *pi*. Some other examples:

```
CR       = '0a'x          /* a linefeed character (0a=10) */

WHITE    = '1b'x"[32m"    /* console sequence for color 2 */

DATAFILE = "s:phonedata" /* file for a phonebook program */
```

System-dependent constants

One thing the above constants all have in common is that they are not constant now and for all time like *pi*, but only in a particular frame of reference. The CR constant defined above is the line-ending character used on the Amiga. It is the ASCII character called 'linefeed', and has the value 10. On some computer systems, the 'carriage return' character, whose value is 13, is used instead. In a script that might have to be converted to run on such a system, one source of incompatibility could be eliminated at a stroke by changing the definition of CR. This is much easier and less error-prone than changing many references to *'0a'x*. Taken together with the fact that a script using the name CR is much more self-explanatory than one that uses the literal numeric string, the case for the named constant is compelling.

The WHITE constant is similar: it too defines a value that is constant within the script but will have to change if the script is used on another system. In this case, however, the system could be another Amiga—one running version 1.3 instead of version 2.0 of the operating system. In the default palette of the Workbench screen under 1.3, the white pen is color 1. Again, it is much easier and safer to change a single variable definition when a script moves to a new environment than it is to change a scattered set of literal values.

Context-dependent constants

The third example, DATAFILE, is a constant *only* within the context of the script in which it is defined. Even if that script were used only by one person on one machine, there could be many reasons for changing the definition—to move the file out of an increasingly crowded **s:** directory, for example. The use of a named constant lets the programmer make the change by editing a single line of the script.

Exposing variables in nested functions

EXPOSE, as we learned above, gives a PROCEDURE-isolated function access to specific variables known to the function's caller. It cannot make variables accessible that are known to the caller's caller, but not the caller itself. The caller must expose any variables belonging to its own caller that the function in question (or any function that *it* in turn may call) needs to use. For example:

```
/* EXPOSE, two levels deep */
red   = "anger"
green = "jealousy"
say "Red means" red "and green means" green"."
call func1()
exit
func1: procedure expose red
say "Red is" red "and green is" green"."
call func2()
return
func2: procedure expose green
say "Red =" red "and green =" green"."
return
```

The output is:

```
Red means anger and green means jealousy.
Red is anger and green is GREEN.
Red = RED and green = GREEN.
```

Simulating constants with functions

Since ARexx does not provide global variables that can override the insulating effect of the PROCEDURE instruction, some variables—those that define an information base, for example—might need to be exposed in virtually every function of a script. With named constants, it is possible to get around this by using, instead of a variable, a tiny function that does nothing more than return the constant value. An example is the PI function with which we began this chapter. This method works because function names *are* globally known within a script. It is not an efficient approach, but it may be worth considering in some cases. With most variables, however, this isn't even an

option. You must either specifically expose the variable in whatever functions it is needed, or allow the function and its caller to share *all* variables, not just the ones in which they have a common interest.

It is easy to sometimes forget to expose a variable or two in the course of developing a large ARexx script, and the resulting bugs can be annoyingly hard to locate. Just remember that this is one of the common problems to look for when things start to go mysteriously wrong.

A Last Look at CHARLINE

Having come to the end of our long excursion into the realm of local variables, we are at last ready to return to the CHARLINE function that occupied so much of our attention earlier in the chapter. Here, incorporating all we so far know—including local variables with PROCEDURE—is the final version of CHARLINE:

```
/* charline(str, reps)
 returns reps concatenated copies of the str.*/
charline: procedure
   parse arg str, reps
   line = ''    /* Initialize the result */
   do reps
      line = line || str
      end
   return line
```

CHARLINE's fatal flaw

Nearly all of this chapter has consisted of creating new versions of CHARLINE, then finding something wrong with the latest version that required further refinement. It is appropriate, then, to end this chapter by pointing out CHARLINE's most fundamental flaw: it is a duplicate of a function that already exists in the built-in library. The library function, called COPIES, does exactly the same thing and takes exactly the same arguments. The only difference is that, being a library function, it is more efficient.

The final lesson of this chapter is something like this: The ability to create your own functions in ARexx is so vital that you will use it over and over again when writing scripts. But many functions are already available, in the built-in library, the support library, or some other library you may have access to. Become familiar with the contents of those libraries, and you may save yourself unnecessary work. Practise writing your own functions, and you won't be at a loss when the libraries can't help you.

Chapter 9
File Input and Output

Very few programs or scripts are sufficient unto themselves. Almost all expect information at run time that did not exist, or was not known, when the programming was done. Sometimes the supplementary information is supplied interactively by a user at the keyboard; sometimes it comes from a file of data stored on disk. Either way, from the script's point of view, the information is *input*.

Similarly, most programs generate new information—*output*. This may be in the form of words and numbers displayed on the screen or a printer, or as text or 'binary' data written to a disk file.

As two faces of the same coin, input and output are usually considered together as *I/O*. In this chapter we will tour ARexx's facilities for both interactive and file-oriented I/O.

We have been leaning heavily for several chapters on the familiar I/O operations of reading a line from the keyboard and displaying a line on the screen. Now we turn again to the same basic operations, but this time using disk files.

The Nature of Disk Files

The pre-eminent form of data storage for microcomputers is magnetic disks, or devices like the Amiga's 'RAM:' that emulate magnetic disks. The exact methods by which information is magnetically encoded on the disk surface, and the ways in which it is organized, are critical to fast, accurate retrieval, and very interesting in themselves. However, operating systems like AmigaDOS are carefully designed to shield ordinary programmers from having to deal directly with the crude physical realities of sectors, cylinders and surfaces of which disks are constructed. They allow us to view the information on suitably formatted disks as being organized into higher-level entities such as files and directories.

Although in reality the information in a file may be stored in many widely-scattered locations on a particular disk, in normal programming the file may be regarded as a mass of data whose

internal organization—into lines, database records, executable program segments, or what have you—is entirely abstract. It need not, and does not, conform to the physical organization in any way.

Because we are not required to know the physical locations on disk at which the information in a file of interest is stored, we need another means of referring to the file. That means is the file name, and it is AmigaDOS's business, not ours, to locate on disk the information to which the file name refers.

Opening and closing files

Compared with transferring the information once it has been located, however, tracking down its location based on the file name is rather slow. If the process had to be repeated every time a program (or script) wanted to exchange information between the computer and a file, the overall pace of I/O operations would slow down dramatically. For this reason, among others, operating systems provide a way of establishing communications with a file, known as *opening* the file. Subsequent I/O is expedited because the operating system, having determined and noted the file's physical location, can now access it directly with no time-consuming search. The opened file must also be *closed* when the program has finished with it, allowing the operating system a chance to sever the communication in an orderly fashion.

Identifying open files

As far as AmigaDOS is concerned, a file after it has been opened by a program is identified by a number. (The number 'happens' to correspond to the memory address of an information structure describing the file, but from the program's point of view its only purpose is to identify the file uniquely while the file is open.) High-level languages usually provide some means of allowing programs to give a more meaningful name to the open file. In ARexx, the script provides a character string for this purpose at the time of opening the file.

Old and new files

Creating a new file and opening an existing one are in some ways very different operations. For instance, creating a new file deletes the existing file of the same name, if one exists. However, the method of opening the file is virtually the same in either case, the only difference being the 'mode' specified as an argument to the OPEN function in the built-in library. The normal reason for creating a file is that one wishes to write data into it, whereas an existing file is most often opened to read the data it already contains. The mode arguments 'w' and 'r' (or 'write' and 'read', if you prefer) determine whether the file name you provide will be treated as a new file to be created or an existing file to be located.

Creating a file

Here is a script to create a file in the 'RAM:' device:

```
/* Create a file */
if open('myfile', 'ram:testfile', 'w') then
    call close('myfile')
else
    say 'Unable to open file.'
```

ARexx's OPEN function

Observe that the OPEN function takes three arguments. The first is the name by which the file will be referred to in the script; the second is the actual file name; and the third is the mode. The last is optional, defaulting to 'read' mode. Of course, you can use string variables or expressions as required instead of the literal strings in the example above.

As you can see from its use in the IF instruction, OPEN returns a boolean result. OPEN can fail for a variety of reasons—invalid file name, disk volume not available, disk write protected, and so on. The result should therefore always be checked and an appropriate action—such as exiting the script—taken to deal with failure.

The CLOSE function

CLOSE can also fail, and so also returns a boolean result. About the only reason for CLOSE to fail, however, is that the file was not open. If this happens it may indicate a logic flaw in your script, but otherwise won't do any harm: after all, you did want the file closed, and it is. In some circumstances, one would expect CLOSE to fail but it actually does not. This will happen, for instance, if you try to close a file on a floppy disk that the user has removed from the drive. AmigaDOS will normally put up a requester asking for the disk to be replaced, but if the user cancels the requester, CLOSE returns True even though the file remains open. This is AmigaDOS's fault, not ARexx's, and in fact it is rarely a problem in practice. Many programmers never even consider the result from CLOSE.

An important duty of high-level languages is protecting programmers from themselves. One way ARexx protects you is by automatically trying to close any files that may still be open when a script exits. This is a handy back-up feature, but most programmers feel that it's sloppy to use it routinely; remembering to close the files you open is one of the sound practices that contribute to reliable, well-structured scripts.

Output: Writing to a File

Now that we know how to create an empty file, it's time to try creating one that actually contains some data. The following script creates a small portion of the information base for a simple telephone area code utility:

Example using WRITELN

```
/* Initialize area code file */
datafile = 'datafile'
filename = 'ram:AreaCode.dat'
if open(datafile, filename, 'w') then do
    call writeln(datafile, '205 Alabama')
    call writeln(datafile, '907 Alaska')
    call writeln(datafile, '602 Arizona')
    call close(datafile)
   end
else
    say 'Unable to open "'filename'"'
```

Standard input and output files

Like the SAY instruction we have used so often, the WRITELN function outputs a character string, and tacks on a linefeed character. SAY always sends text to the 'standard output' file, however, which in most cases is a CLI window. WRITELN is versatile enough to write to *any* open file.

Incidentally, although the standard output file is special in that it is normally already open when your script starts to run, it is in most respects a file like any other. It has its own name, 'STDOUT', which you can use if you wish in instructions like:

```
call writeln('STDOUT', 'Hello world!')
```

The corresponding 'standard input' file, named 'STDIN', is also available for your use. Calling READLN on 'STDIN' is very much like using PARSE PULL, but without the special capabilities of the PARSE instruction (which we'll be covering in full next chapter).

Since the file we just created is an ordinary text file, we can check its contents with the AmigaDOS *type* command from the CLI:

```
Shell> type ram:AreaCode.dat
205 Alabama
907 Alaska
602 Arizona
```

Everything has gone exactly as planned. Yet someone with a skeptical turn of mind might still be troubled by such questions as:

- What if something goes wrong? What if, for instance, available RAM was exhausted by a call to WRITELN? Or, if we decided to use a file on floppy disk, what if it filled up, or was write protected, or had a 'read-write error'? Should the script be able to handle these eventualities? Can it be made to do so?

- In any case, what point can there be in writing a script merely to generate a file whose contents are already known? Is that not clumsily indirect, when the same data file could easily be created in an ordinary text editor with less typing?

In answer to the second question, it has to be admitted that the script as it stands is not the most efficient means of creating the data file. Rather than immediately abandoning the idea of using a script in favor of an ordinary text editor, however, we might consider a third alternative: writing a script that serves as a specialized, bullet-proof text editor that not only lets us enter the text for the data file, but also verifies that it is in the correct format. We'll investigate the practicability of writing such a script towards the end of this chapter.

Dealing with I/O errors

Dealing with disk full errors and the like is a thornier issue, as we have already seen in discussing CLOSE. The WRITELN function does return a value: the length of the string actually written to the file. If (and only if) this value is one greater than the length of the string we *intended* to write (the extra character being the appended linefeed), no error occurred. In a 'quick and dirty' script, we may be so confident that WRITELN will not fail that we neglect to check its return value. The script above, which is meant to be run only once, is a case in point.

In the general case, however, this attitude is too cavalier. An error that is not properly handled, or at least reported to the user, is an error whose undetected consequences may involve loss of data—a cardinal sin that all programs should strive to avoid. As a rule, AmigaDOS will itself notify the user through a requester when something has gone wrong with writing to a file, but it's unwise to depend on it doing so. A script designed for future use should report and gracefully deal with any errors that may arise. With WRITELN, one method of doing so is to provide a customized version of the function—a version that deals with errors in a way appropriate for your script. Here's a simple example of how this could be done:

```
/* Customized WRITELN */
writeln: procedure
   len = 'writeln'(arg(1), arg(2))
   if len ~= length(arg(2)) + 1 then do
      say "Error on write to file. Aborting script..."
      call close(arg(1))
      exit
      end
    return len
```

Replacing built-in functions

This version of WRITELN takes advantage of a somewhat obscure feature of ARexx mentioned in the discussion of the search order for matching function names back in Chapter 6: a function name in quotes is never taken to be the name of an *internal* function (a function defined in the script). When your script calls WRITELN (with no quotes around the name), the first place ARexx looks for a function of that name is within the script itself. Normally there won't be one, so it will have to go on to search in the built-in library, where it will find a match. But in the present case, we *do* have a function called WRITELN in the script, and it's this customized version that will be called. And now comes the trick... The customized WRITELN is only an error-handling shell for the 'real' WRITELN, which it must somehow call—without accidentally invoking itself! It does so by calling 'WRITELN' (in quotes), forcing ARexx to bypass the script and move straight on to the built-in library.

As you can see, the error handling done by our version of WRITELN is anything but fancy: it does no more than exit the script with an error message at the first sign of trouble. Often that's enough, since it at least prevents the error being compounded by further processing of what is probably corrupt data. More sophisticated versions could be written to fill particular needs. An advantage of this approach is that it can be added 'transparently' to an existing program. Simply add the error-checking version of WRITELN to our original script for writing the area-code file, and it will be automatically assimilated, without changing a single line of that script.

WRITECH

A very similar built-in library function is WRITECH, whose only difference from WRITELN is that it does not added a linefeed to the output string. This is useful when you are building a line piece by piece, or dealing with non-textual data in the form of hexadecimal strings, for example. This little demo of WRITECH sends text to the standard output file:

```
/* WRITECH demo */
line = "hope is merely disappointment deferred"
do w=1 to words(line)
    call writech('STDOUT', reverse(word(line,w))' ')
    end
```

```
call writech('STDOUT','0a'x)
```

which produces the output (in your CLI window):

```
epoh si ylerem tnemtnioppasid derrefed
```

Input: Reading from a File

Perhaps you have noticed that our work with files has so far been notably straightforward, despite the excursions into error checking and function name matching. We have covered opening and closing files, with OPEN and CLOSE, and output to files with WRITELN and WRITECH. We are about to discover that input is just as easy...

Simple parsing of input

The area-code file consists of a number of lines in the format:

```
nnn <place>
```

Here *nnn* represents a number with exactly three digits, and *<place>* indicates a string giving the state or province using the area code of that number. After reading such a text line from a file into the variable LINE, we could parse the information in it with:

```
area  = word(line, 1)
place = subword(line, 2)
```

One useful form in which to store the information in the area-code file would be a set of compound variables. One plan would be to use the numeric code as a way of accessing the place information:

```
areacode.205 = "Alabama"
```

or generally:

```
areacode.area = place
```

Reading from a file with READLN

Almost the only obstacle to implementing this system immediately is that we don't yet know how to read lines from a file. As you might expect, the relevant function is called READLN:

```
/* Reading from area code file */
datafile  = 'datafile'
filename  = 'ram:AreaCode.dat'
areacode. = 'Unknown'
if open(datafile, filename, 'r') then do
   do 3
      line  = readln(datafile)
      area  = word(line, 1)
```

```
            place = subword(line, 2)
            areacode.area = place
            end
        call close(datafile)
    else
        say 'Unable to open "'filename'"'
```

In most respects, this script is a close analogue of the one used to create the data file in first place. The most important difference is the mode 'r' (for 'read', as we learned earlier) used to open the file. We read the three lines in the file with READLN, which returns the next line of the file, minus its terminal linefeed, as an ordinary string value. We split the string into an area code and a place name, and use these to initialize a compound variable as planned. We have already set up the compound variable with the value 'Unknown' so that area codes not in the database will be handled in a reasonable way if the script is called upon to try and access them.

Determining End-Of-File

The script is, however, somewhat feeble-minded in its rigid expectation that the area code file will contain exactly three lines. That expectation will be true if the file was created with the script given earlier in this chapter, but it would be very inflexible to insist on that. Our present script would be much improved if it could read exactly as many lines as were available in the input file, then stop without making a fuss. If there were three lines, that would be fine. If there were thirty, or three hundred, or zero lines, it should also be fine. To program in the extra intelligence for this feature, we need a function for determining if we have yet read to the end of the input file.

The EOF function

The function exists in the built-in library; its name is EOF (for 'end of file'). Given the name of an open file, EOF returns *true* if the end of the file has been reached, and *false* if it has not. The end-of-file condition is detected the first time an attempt to read (with READLN, for example) fails because not enough data, or no data at all, is available.

Suppose we have a file called *datafile,* that contains exactly two lines of text, each terminated as usual with a linefeed character. The first call to READLN on this file will read in the first line, and set the 'current position' for the file to just after the linefeed that marks the end of that line. The second call to READLN will bring the second line in, and again set the current position to just after the linefeed character. There are no characters left to read, but because both reads to date have been successful, the end-of-file condition has not been

detected: if we call EOF now it will return *false*. We call READLN one last time, and fail—the length of the returned string will be zero. A call to EOF will at last return *true*.

Reading an entire file

To show EOF in action, here is one version of a script that displays the text contained in a given file, like the AmigaDOS *type* command:

```
/* type.rexx */
tfile = 'typefile'
if open(tfile,arg(1),'r') then do
   do while ~eof(tfile)
      say readln(tfile)
      end
   call close(tfile)
   end
else
   say "Unable to open file" arg(1)"."
```

If you experiment with this script, you'll discover that it has a small bug: most files are displayed with an extra blank line at the end. We already know the reason for this: the end of file is not detected until after READLN fails, but we are displaying the results of calling READLN regardless of its possible failure. How can we fix this? The standard approach, used with I/O operations in many programming languages, is to perform the read once *before* the while, and again just before the end of the while:

```
/* type.rexx */
tfile = 'typefile'
if open(tfile,arg(1),'r') then do
   line = readln(tfile) /* get first line of file */
   do while ~eof(tfile) /* read all lines in file */
      say line
      line = readln(tfile) /* read next line */
      end
   call close(tfile)
   end
else
   say "Unable to open file" arg(1)"."
```

This handles the case where the file is empty (contains no lines of text), and stops properly at the last line of the file.

Logic errors in even a simple file-reading program are easy to commit if you're not watching for them. Line-by-line processing of text files is a common application for ARexx scripts, so it's a good idea to understand the potential problems from the start.

READCH

Just as READLN corresponds to WRITELN, there is a READCH function that corresponds to WRITECH. It is most useful for reading portions of 'binary' rather than ASCII files—files that may include any kind of non-textual data rather than being organized into lines of text characters.

Example: checking an IFF file

For instance, simple IFF files containing sounds, pictures, animations, and so on, begin with the four characters 'FORM', followed by four non-text bytes giving a byte count (generally the file size less the eight we have read), followed by yet four more giving the type of IFF 'form', such as '8SVX' for sounds or 'ILBM' for pictures.

The following little script reads in these three fields from a given IFF file in turn, and displays information about the file to the user:

```
/* Check IFF file with READCH */
iff_file = 'iff_file'
if open(iff_file, arg(1), 'r') then do
 type = readch(iff_file, 4)
 if type ~= 'FORM' then
    say '"'arg(1)'" is not a simple IFF file.'
 else do
   size = readch(iff_file, 4)
   form = readch(iff_file, 4)
   if length(form)=4 then
    say 'File "'arg(1)'": size' c2d(size)+8', type "'form'".'
   else
      say 'Unable to read 12 bytes from "'arg(1)'".'
   end
 call close(iff_file)
 end
else
 say 'Unable to open "'arg(1)'".'
```

Checking for read errors

To test for error conditions after a call to READCH, you can check that the length of the returned string contains the requested number of bytes. If it does not, the cause is either that the end of file has been reached or there was a read error. The script above makes this test twice: once implicitly, when it compares the *type* variable with the literal string 'FORM'; and later explicitly by determining the length of the *form* string. If the second test succeeds, we may feel assured that the *size* variable was also read successfully.

SEEK: Positioning within a File

We have referred to the concept of a 'current position' within an open file—the position from which the next data will be read or to which the next data will be written. This position, which normally is zero when a file is opened, and increases as the file is subsequently accessed, is tracked by AmigaDOS. Sometimes it is useful to be able to change the file position directly, without actually reading or writing data. This is called 'seeking' within the file, and ARexx provides for the purpose a built-in function called SEEK.

To demonstrate SEEK we will look at another type of binary file. It is called *system-configuration*, and it is found in your *devs:* directory. The file contains your system settings, as set in Preferences. One of these, your choice of printers, is stored as a name beginning 128 bytes from the start of the file and occupying up to 30 bytes (the end of the name is signified by a byte whose value is zero; bytes beyond that should be ignored).

SEEK lets you specify a desired file position relative to any of three 'anchor positions': the beginning of the file, the current position, or the end of the file. These are represented by the modes 'b', 'c' and 'e' respectively, the default being 'c'. You give as an offset from the anchor position a number that may be either positive or negative. A positive offset will move towards the end of the file from the anchor position; a negative number will move towards the beginning. For example, if you wished to seek backwards 100 bytes from your current position, the appropriate call would be:

```
newpos = seek(file, -100, 'c')
```

The return value from SEEK is the new position relative to the beginning of the file. In this next script we use the anchor 'b' for the call to SEEK, positioning relative to the start of the file:

```
/* What printer is chosen in Preferences?
   (Works with AmigaDOS 1.3 and earlier only)
   The ☞ character indicates that a line
   should be entered as a single line
*/
cfg_file = "cfg_file"
if open(cfg_file, "devs:system-configuration", "r") ☞
 then do
   if seek(cfg_file, 128, 'b') = 128 then do
      printer = readch(cfg_file, 30)
      if length(printer) = 30 then
```

```
            say "Current printer:" left(printer, ☞
                        index(printer,'00'x) - 1)
        else
            say "Error on read"
        end
    else
        say "Error on seek"
    call close(cfg_file)
    end
else
    say "Error on open"
```

One risk you run by poking around in binary files like this is that file formats are not always written in stone. Unless you are quite sure that files of the type you are examining will always adhere exactly to their present format, you must be prepared for the possibility that the format will at some time change, invalidating your script. The above script, for example, will not work with Version 2 of AmigaDOS, since the Preferences printer name is no longer stored in *system-configuration*. You can still read those bytes as we did above, and in all likelihood you will even find a printer name there, but it probably won't be the one you have set in Preferences. The moral is: try *not* to meddle with binary files whose formats you can't control; but if you must, proceed with full knowledge of the risk that the format will change.

Modifying existing files

In our discussion of OPEN, near the beginning of this section on file operations, we stated that the mode arguments 'r' (for 'read') and 'w' (for 'write') control whether or not a new file is created by the OPEN call. The mode does not, therefore, necessarily declare your intentions about the opened file in quite the way that the words 'read' and 'write' would imply. Though reading from an old file and writing to a new one are certainly typical operations, there is actually no restriction. You can open an existing file with 'r' and write into it with WRITELN or—perhaps more usually—WRITECH. The data you write will replace the same number of characters from the current file position, or be appended to the file if you are positioned at the end. Similarly, you can read data from a file that you have opened (and hence created) with 'w', though of course you will have to write something into it before there is anything to read, then call SEEK to move the file position back over what you have written.

There is a third mode for OPEN: 'a' for 'append'. It is really a variant of 'r'. An existing file is opened as usual, but the file position is immediately set to the end of the file in preparation for adding new

material. It is equivalent to opening the file with 'r' then immediately seeking to the end.

That completes our grand tour of ARexx's built-in file functions. Working with files is usually quite straightforward if you have a clear-cut plan. Certainly, the behavior of files is comfortably predictable compared to what may be expected in dealing interactively with a human. Which is an important aspect of the next topic...

Interactive Input and Output

To end this chapter, we return to the idea of creating a special-purpose 'text editor' that will input and validate records for the area-code database. To begin, let's open the database file in append mode, in preparation for adding new records:

```
/* Area-code database 'text editor' */
db_file = 'datafile'
db_name = 'ram:AreaCode.dat'
if open(db_file, db_name, 'a') then do
   /* Code to input and append records goes here */
   call close(db_file)
   end
else
   say "Unable to open database file."
```

The file must already exist for an open in append mode to work. If it does not, you can initialize it from the Shell with:

```
Shell> echo >ram:AreaCode.dat noline
```

Alternatively, you could have the script check for the existence of the file (using the EXISTS function from the built-in library) and create it if necessary.

The input phase of this script will be a loop. Each iteration of the loop will get one line of input from the user, validate it, then append it to the file. (Instead of using READLN from STDIN, user input is performed with the PARSE PULL instruction. The PARSE instruction is discussed in detail in the next chapter.) We'll break out of the loop when we see an empty line:

```
/* Code to input and append records */
/* The ☞ character indicates that a line
   should be entered as a single line */
options prompt ">"
do forever
   parse pull area place
```

```
        if area = '' then
           break
        else if ~datatype(area,'n') then
            say "*** Area code must be numeric."
        else if length(area) ~= 3 then
            say "*** Area code must be three digits."
        else do
            record = area place
            if writeln(db_file, record) ~= length(record)
☞
           + 1 then
               say "*** Warning: write failed! ***"
           end
        end
```

Putting its two parts together, this script is slightly longer than most of our other examples, but for a real-world program with error-checking, albeit a crude one, it's no heavyweight.

Once you've had a little bit of practice with ARexx programming, you'll be able to whip together scripts like this in no time flat, and in the process you'll find that doing so is not only useful but fun.

Chapter 10
Parsing and String Handling

Introduction to Parsing

In the study of grammar, to *parse* a sentence is to analyze its grammatical structure in terms of parts of speech (nouns, verbs and so on), phrases and clauses. Programming-language compilers and interpreters (including ARexx) parse instructions in a similar sense, except that the grammars of computer languages are much simpler than those of natural languages, and much more rigorously defined.

Parsing in everyday programming is humbler still. Usually, the strings to be analyzed have only one allowed 'grammatical' structure, or at most a few. The main goal of parsing is to split up a string into meaningful subunits based on its structure.

Parsing with string functions

We encountered a simple example of string parsing in Chapter 9 when we used the WORD and SUBWORD functions to split lines read from the area code file into their two parts:

WORD and SUBWORD

```
area  = word(line, 1)
place = subword(line, 2)
```

The structure of the area code lines is simple enough that we can parse by looking at the first word boundary only. The first word is the area code; the balance of the line, which may have more than one word, is the corresponding state or province. If it had turned out that *both* fields could contain a variable number of words, we would have needed another method of locating the boundary between the two fields. A special punctuation character, such as the comma, might be used for this. Suppose the input lines each contained a name and address:

```
Ryan Ginger,1120 Black Forest Lane
Viola da Gamba,Apt. 4, 211 Chancery Street
```

INDEX, LEFT and SUBSTR

In these lines, the first comma marks the end of the first field and the beginning of the second. Extracting the two fields is simple using the INDEX function:

```
com_pos = index(line, ",")
if com_pos = 0 then
   say "Comma missing in input line."
else do
   name = left(line, com_pos - 1)
   addr = substr(line, com_pos + 1)
   end
```

Parsing with the built-in string functions offers the ultimate in flexibility and power: there is essentially no limit to the complexity of the parsing operations you can program in this way. Part of ARexx's guiding philosophy, though, is that the commonest operations should be as simple as possible to program, so a second method of parsing is provided—the PARSE instruction.

Other string functions

ARexx contains about 30 string functions altogether. The reference section (section IV) explains each function in detail with examples; these detailed descriptions are in alphabetical order. To browse through the available string functions and find the one you need for a particular task, see the *Reference Guide* at the beginning of the section.

The PARSE Instruction

The capabilities of PARSE are not as general as those of parsing 'by hand' with the string functions. In the many situations PARSE *can* handle, however, it provides a simpler alternative approach. For instance, here's one way we can parse out the area code and state name from the *line* variable:

```
parse var line area " " place
```

Format of a PARSE instruction

A PARSE instruction consists of three parts, the first being the instruction keyword, PARSE, itself. Next comes the *parse string*, or *source*, which tells PARSE where to find the string it is to analyze. The source for this particular instruction, specified by *var line*, is the *line* variable. The third part, called the *template*, tells PARSE how to split up the source string. In our example, the template causes the contents of the string to be split into the two variables *area* and *place*, using the first space in the string as the split point. The space itself is

consumed—it does not appear in either of the two output strings. As for *line*, it is unchanged. The parsing operation does not affect the original source string, only the variables that appear in the template, and sometimes the working copy of the source string used by PARSE.

In exactly the same way, PARSE can easily handle the name and address parsing example in which we used INDEX to locate a comma dividing the name and address fields, then other string functions to extract the fields themselves. With PARSE, it's much simpler:

```
parse var line name "," addr
```

Parsing user input

PARSE (as we have seen in earlier chapters) can work from sources other than variables, giving the instruction special capabilities. Giving the source *pull*, for example, causes a string to be read from the keyboard (or more precisely, the 'standard input'). The template for this source is quite often a single variable name, meaning that the entire input string is transferred to the variable, but this need not be so. For example:

```
/* using parse pull */
say "Enter name,age, e.g.: Fred Foremost,44"
parse pull name "," age
say "You claim to be" name"," age "years old."
```

One would often prefer the cursor displayed for keyboard input to appear on the same line as the prompt itself. This can arrange with a variant of the OPTIONS instruction, OPTIONS PROMPT. Here's another version of the previous script:

```
/* using parse pull */
options prompt "Enter name,age, e.g.: Fred Foremost,44: "
parse pull name "," age
say "You claim to be" name"," age "years old."
```

Initially the prompt is set to the null string, which is why we have not encountered it in examples up till now. With OPTIONS PROMPT, though, you can change it as often as you like.

Parsing arguments

Another source we have already encountered is *arg*, in which the source consists of the argument string (or strings) to the current script or function. When parsing with *arg*, we often use multiple templates in the same PARSE instruction, one template for each argument we expect to be available. The comma-separated templates are applied in turn to each available argument string. The following instruction, for instance, might be used to initialize local variables in a function that expects three arguments:

```
parse arg name, age, weight
```

We will have more to say later on about the types of source PARSE can use. First, though, let us explore the construction of templates in more detail.

PARSE Templates

The purpose of parsing is to extract meaningful substrings from the source string and store them in variables. Those variables are called the *targets* of a PARSE operation. In this PARSE instruction, the targets are *area* and *place*:

```
parse var line area " " place
```

How does PARSE determine which parts of the parse string are assigned to which target variables? The answer is in the other component of templates: *markers*. There are several types of marker; each specifies in its own way a *position* in the parse string. In the example just above, the quoted space character is a marker specifying the position of the first space in the parse string. The first target, *area*, is assigned everything in the parse string up to that position; the second target, *place*, is assigned everything after that position.

Parsing by tokenization

In very much the same way as blanks within a string expression serve as an implied concatenation operator, so blanks separating two targets within a template serve as an implied marker that will result in the word to the left of the blanks being assigned to the left-hand target. An

example will make this feature, called parsing by tokenization, easy to grasp:

Parsing words

```
/* Parsing by tokenization */
line = "The Pied Piper of Hamelin"
parse var line first second third rest
say "<"first">"
say "<"second">"
say "<"third">"
say "<"rest">"
```

Notice in the output that it is *only* the left-hand target of each pair that is assigned its substring through tokenization. The result is that the final target, *rest*, is assigned the remainder of the string including the space character:

```
<The>
<Pied>
<Piper>
< of Hamelin>
```

In general, if the last item in the template is a target, it is assigned whatever is left over in the parse string. This is the same as saying that there is an implied marker at the end of the template specifying the position *end of string*.

What happens if the parse string is exhausted before all the targets have been assigned? For instance, suppose we modify the third line of the previous example to read:

```
parse var line first second third fourth fifth rest
```

and add lines to output the new variables *fourth* and *fifth*. Since there are only five words in the parse string, what will happen to the variable *rest*? As you might logically expect, it is assigned the empty string. The output from the modified example is:

```
<The>
<Pied>
<Piper>
<of>
<Hamelin>
<>
```

Observe that *rest*, unimportant in itself, has nonetheless performed a useful service in forcing *fifth* to be parsed by tokenization.

Placeholders

There is one special target that is not a variable. It is called the *placeholder*, and is represented by a period in the template. The placeholder behaves exactly like any other target in that it 'consumes' a portion of the parse string. However, the substring matched by the placeholder is thrown away; it is designed specifically for the job performed by *rest* in our last examples. We could (and should) modify the PARSE instruction of those examples once more to use the placeholder rather than a dummy variable, giving simply:

```
parse var line first second third fourth fifth .
```

Pattern markers

Parsing with the implied markers of tokenization is very easy and intuitive. Often it's all you need. Another easy method that you can use either alone or in combination with tokenization involves *pattern markers*—literal strings that specify a position in the parse string containing the matching characters. We have seen a pattern marker in one of our earlier examples. It is the quoted space character in:

```
parse var line area " " place
```

For another example, suppose we need to extract the day, month and year from a date formatted by AmigaDOS, such as '27-Jun-91'. This could be done with simply:

```
date = "27-Jun-91"
parse var date day "-" month "-" year
```

The dashes matched by the pattern markers are not assigned to any target, implying that the markers do more than merely indicate a position in the parse string—they actually *remove* the matched text from the parse string in the course of matching it. (This doesn't affect the original string, only the working copy used by PARSE.) Pattern markers are the only type that does modify the parse string. Normally this passes unnoticed, but it does produce some side effects when pattern markers appear in the same template with the various types of numeric marker covered below.

Multi-character markers

Pattern markers can contain multiple characters. Suppose you need to extract a list of players from a file containing a list of tournament match-ups in the form of two names separated by 'vs.', like this:

```
John O'Reilly vs. Melissa Jane Freeman
```

The names can be extracted using exactly the same technique as in other examples:

```
parse var line name1 " vs. " name2
```

The list for a doubles tournament would be more complicated:

```
John O'Reilly and Tina Smith vs. Melissa Jane ☛
     Freeman and Bob Yuen
```

Using the string functions to parse this kind of line would be significantly more difficult than with the simpler lines we've tried before. With PARSE, and pattern markers, the difference is trivial:

```
parse var line name1a " and " name1b " vs. ☛
     " name2a " and " name2b
```

Using variables as pattern markers

By the way, although pattern markers are most often given as literal strings, the name of a variable containing the pattern string can be used instead. In order that the variable won't be taken as a target in the template, it is enclosed in parentheses:

```
/* Variable pattern markers */
vocab_line = "/my house/\chez moi\"
delim='/'
parse var vocab_line (delim) text (delim)
say text
delim='\'
parse var vocab_line (delim) text (delim)
say text
```

As expected, this produces the output:

```
my house
chez moi
```

Parsing fixed-length fields

Database records are often stored as a set of fixed-length fields. Parsing records of this kind depends purely on quantitative information: the position of each field, expressed as a count of characters from the left of the field, must be known in advance. Let's say you're using your Amiga to keep track of the exploits of your favorite baseball team. Each record in your database of game results is constructed on this pattern:

Name of opposing team	20 characters (padded on right with spaces)
Runs scored by your team	2 digits
Runs scored by opponents	2 digits

Your team's runs by inning	2 digits by 9 innings - 18 characters
Opponent's runs by inning	2 digits by 9 innings - 18 characters

Here is an example of how a record might look:

```
    Montreal Expos
070301010200000003000000000010001000100
```

From the pattern you can determine that the fields within a record begin at characters 1, 21, 23, 25 and 43, and that the width of a whole record is 60 characters. Records can be read from the database with an instruction like:

```
record = readch('baseball_file', 60)
```

Absolute markers

To parse the record, we use *absolute* markers, which numerically specify a position in the parse string (the following should be entered on one line):

```
parse var record opp_name 21 our_runs 23 opp_runs ☛
     25 our_inn 43 opp_inn
```

As usual, the starting position in the parse string is 1, and the ending position is the end of the string. The other positions, given as literal numbers, show the positions at which the string should be split. Unlike pattern markers, absolute markers do not consume any characters in the parse string: the split point is between the specified character and the preceding one.

Relative markers

Another type of numeric marker is the *relative* marker, which specifies a position relative to the position determined from the previous marker, whatever it may have been. Both *area1* and *area2* in this example have the same value:

```
/* Relative markers with + and - */
rec = "Bill Smith, (604) 555-9378"
parse var rec '(' area1 +3
parse var rec ')' -3 area2 +3
```

The number -3 in the second PARSE line in the example depends on the fact that the text matching a pattern marker is removed from the parse string, as mentioned above. If this were not so, the value would have to be -4 for the instruction to work correctly.

Like pattern markers, numeric markers can be variables as well as literal values. But if that's the case, you may wonder, how can ARexx distinguish between the variables used for markers and those used for targets? In the case of relative markers, there is no difficulty: the minus or plus sign eliminates any possible ambiguity. With absolute

markers, the problem is solved by preceding the variable name with an equals sign:

```
/* Numeric markers using variables */
nm1 = 3
nm2 = 6
nm3 = 9
digits = '1234567890'
parse var digits left =nm2 -nm1 mid +nm1 right =nm3
say left '-' mid '-' right
```

This produces the output:

```
12345 - 345 - 678
```

Extracting words from a string

Since strings in ARexx can be of any length, they provide a convenient way to store single-word data elements such as lists of names. The individual words can then be extracted from the string by a number of means, including the SUBWORD and WORD string functions. When each word in the string needs to be processed within a loop, however, a convenient technique is to use PARSE to "pull out" one word at a time from the string. This is done by using the same variable as the source of the PARSE template, and as the final target.

Consider this use of the PARSE instruction:

```
shoplist = "milk eggs bread cheese disks coffee"
parse var shoplist item shoplist
```

Clearly, after the PARSE instruction *item* will contain the first word in the *shoplist* variable, 'milk'. Since *shoplist* is also the last target in the PARSE template, it will now contain everything else in the list; in other words, the first word has been stripped from the string.

Using this technique in a loop makes it easy to work with each item in the list, one by one, regardless of how many there are. The loop terminates when the string is empty, meaning there are no more words to parse. The following script is a simple example, using the shopping list string again:

```
/* Parsing in a loop */
shoplist = "milk eggs bread cheese disks coffee"
do while shoplist ~= ""
   parse var shoplist item shoplist
   say "<"item">"
   end
```

The output from this script will be:

```
<milk>
<eggs>
<bread>
<cheese>
<disks>
<coffee>
```

The same technique can be used to parse command-line arguments to a script. This is especially well-suited to a script that accepts several command-line arguments, each to be processed in the same way. For example, a script that counts the words in a file would best serve the user if it could accept several file names, count the words in each of the files, and report a total word count as well as the individual counts in each of the files. As you saw back in Chapter 4, scripts get the entire command-line—all the file names, in this case—as a single argument. Using PARSE ARG with a list of variable names works well when the number of arguments is known, but the word-counting script should be able to accept an arbitrary number of file names. Applying the extraction technique used above, you could set up the script like this:

```
/* count words in any number of files */
parse arg files
/* put all file names in "files" string */
total=0
do while files ~= ""
   parse var files name files
   word_count=words_in_file(name)
   total=total+word_count
   say "FILE: " name "WORDS:" word_count
   end
say "TOTAL WORDS:" total
```

The *words_in_file* function is not listed here, but writing it would make an excellent exercise at this point, combining file I/O operations (Chapter 9) with string operations.

In the above script, the single argument is initially placed into the *files* variable by a simple PARSE instruction. Another method that would work equally well would be to use the ARG function, as in:

```
files=arg(1)
```

PARSE String Sources

Throughout this chapter, we have been dealing with a single source for parse strings: *var*, which means that the string is the value of the named variable. There are several other commonly-used sources for the parse string, along with a few others that are used less frequently. We'll discuss only the commoner ones here; you'll find the rest covered under PARSE in the Reference Section at the end of the book.

PARSE ARG

We encountered this source informally in an earlier chapter. The parse string it produces is the argument string to the current command, or the first argument to the current function, depending on where it is used. Commands have only one argument, but external functions can have up to fifteen and other types can have any number. In order to process multiple arguments, you can supply the PARSE instruction with multiple templates, separated by commas, to which successive arguments will be applied in turn. Most often, each argument will consist of a single value, and the template for it will simply be a variable name:

```
parse arg height, width, breadth
/* Three templates */
```

PARSE PULL

We've met this source before as well. It reads the parse string from the 'standard input' device (normally the keyboard). If you provide multiple templates with PARSE PULL, multiple lines will be input. Again, it's common with this source for the template to consist of just a variable name:

```
parse pull line
```

PARSE VALUE

The *value* source lets you use any ARexx expression to generate the parse string. The keyword WITH marks the end of the source expression; the template comes immediately thereafter. This instruction:

```
parse value substr(line,10) with name ", " address
```

is identical in effect to:

```
str = substr(line,10)
parse var str name ", " address
```

Chapter 11
Debugging, Tracing and Error Trapping

Debugging

Up to now we have learned new ARexx programming concepts piece by piece, working with short scripts to test each concept as it was introduced. If you have been experimenting along the way by writing your own scripts, you have probably come up with unexpected results (a kind way of saying 'bugs') from time to time because of a logic or programming error in your ARexx code.

With programs just a few lines long, any problems can usually be discovered and corrected by 'thinking through' each line of code, perhaps trying out some experiments, and then modifying and re-testing the program.

In 'real life,' however, even the smallest scripts are often refined until they grow far beyond their humble origins. When you are working with complicated scripts of hundreds or even thousands of lines of code, fixing the problems can be quite a bit more troublesome.

Learning to debug

Whenever a program you're testing doesn't do what it's supposed to, you've found a bug. A blatant example of a bug—and one of the easiest to fix—is a syntactical error that results in ARexx halting the script and reporting an error message. Subtler bugs are those that simply result in incorrect output from your programs. Even harder to locate are problems that only seem to happen once in a while.

No matter what the case, you'll have to find the cause of the bug before you can fix it, and this can be the most difficult and time-consuming part of programming. The process of correcting bugs in a program is called debugging, and it's unfortunately something you'll end up doing with just about every program you write. While your goal should always be to 'get it right the first time,' it is rare to complete a program without discovering any bugs along the way.

The pervasiveness of bugs can cause great problems for the beginning programmer, or even an experienced programmer using a new language for the first time. While it is reasonably straightforward to teach programming concepts in a book, there is no obvious step-by-step way to teach debugging. Most programmers think of debugging as more of an art than a science, and debugging proficiency seems to benefit more from experience than from any amount of reference material or written examples.

On the other hand, there are a number of techniques you can adopt that will help you in your attempts at tracking down difficult bugs. Before getting into these, it's worth reviewing some of the bugs you may have encountered so far.

Common errors Even in the short ARexx scripts that you might have created while taking your first steps at learning the language, it is likely that you have had to fix your programs after committing one or more of the following errors:

- Typing mistakes resulting in the script being halted and an error message of some kind being printed out.
- Leaving off a DO or END instruction for an IF..THEN clause, resulting in the program doing something other than expected before the script was halted with an error.
- Using the wrong name for an instruction, resulting in the instruction seemingly being ignored by ARexx.
- Failing to initialize a variable before using it in a numeric expression, resulting in an 'Arithmetic conversion error'.
- Passing a function the wrong arguments or passing arguments in the wrong order.
- Logic errors: the program is doing exactly what you told it to, but there's something wrong with your algorithm—the method you've devised for solving the problem.

The above examples encompass a large number of specific errors, but don't come close to covering all of the categories of errors that can be made. They do illustrate a progression, however, from the kinds of bugs that are relatively easy to locate and fix to ones that can be trickier.

Syntactical 'fatal' errors

The first category of bugs is generally the most trivial to discover and fix. If you make an error that causes ARexx to stop the program and report an error of some kind, consider yourself lucky: you know where the problem is, and the error message will give you some clue where to look. It doesn't tell the whole story, however: hitting a wrong key or two in typing a clause or expression could result in "Extraneous characters," "Invalid keyword," "Unbalanced parentheses," "Function not found," or a number of other messages.

Fortunately, if the mistake was due to carelessness and not because of a misunderstanding you have about the way an expression or clause should be constructed, you'll usually be able to identify the problem as soon as you see the offending line of code.

The second type of error—leaving off a DO or END around a THEN block—can sometimes produce unexpected results, but ARexx will usually catch these errors (sometimes only at the end of a script) by reporting "Missing or unexpected THEN." Still pretty simple.

Invalid instructions

The third type of error—using the wrong name for an instruction—can be a bit more difficult to trace, since something you expect your program to do just doesn't happen. If you're a BASIC programmer and accidentally use PRINT instead of SAY for example, you won't get an error message; the instruction will simply be ignored.

This is due to the way external commands are handled: anything that isn't a valid ARexx clause or instruction is passed as a command to the current host. The default host is 'REXX', which will run a script by the given name ('print.rexx' in this case) if it is present. If no such script exists, nothing happens at all.

If you're not using a command host in your program, you can catch these errors by specifying a nonexistent host address at the start of the program, for example ADDRESS XXX. All non-ARexx instructions will then result in the script halting with a "Host environment not found" error.

Uninitialized variables

The fourth type of error, not initializing a variable before using it, can sometimes have confusing consequences in ARexx since a variable takes on its own name as a value by default. This is why using an unitialized variable in a numeric expression yields an "Arithmetic conversion error." In most cases, however, these bugs can still be fairly quick to locate.

Errors in function parameters

Errorsresulting from using functions incorrectly can be difficult to diagnose if you don't know what you're doing wrong. If you think you understand the way a function is supposed to work but you have made a false assumption, it will be very hard to see the problem in your program no matter how carefully you look.

Logic errors

Logic errors, probably the most common cause of bugs, can also be the most difficult to uncover. It is impossible to find a problem with the program itself because there isn't one: you just told it to do the wrong thing. You won't be able to find a missing variable initialization or an incorrect value in the program, and if you don't realize that the method itself is at fault, you could spend hours looking for program errors that aren't there.

Program diagnostics

When you find yourself trying to pinpoint one of the trickier bugs, effective debugging methods can save a great deal of time by pointing you to the heart of the problem. Knowing that "Everything is working fine up to this point" can save a lot of floundering around looking at irrelevant parts of a program.

One of the simplest and most effective bug-killing tools at your disposal is the simple SAY instruction. By using SAY to display the contents of relevant variables at strategic points in the execution of a program, you can get an 'inside look' at the program as it runs. There are a few different ways this can help you:

Tracing program flow

When you look at your program in the general area of the bug, it is useful to know what branches are being taken, what DO loops are being executed, if a specific function is being called, etc. This can often immediately point you to a specific IF or DO WHILE statement that is branching unexpectedly.

For this simple form of tracing, consider the following kinds of messages in your debugging SAY instructions:

```
say 'Reached function "GetData"'
 /* just after function label */
say 'Exiting function "GetData"'
 /* just before function RETURN */
say 'Inside of name-comparison IF'
say 'Taking OTHERWISE in SELECT'
say 'Inside record-count DO loop'
```

Conditional debug instructions

Instructions like the last one that occur inside a DO loop can result in lots of messages being printed to the console window. For loops with a great many repetitions, you might want to display the message only every tenth loop iteration or so:

```
LoopCnt = 0
DO ...
    /* body of DO loop */
    if LoopCnt // 10 == 0 then
        say 'Inside DO loop, iteration #' LoopCnt
    LoopCnt = LoopCnt + 1
    END
```

The expression 'LoopCnt // 10 == 0' will only be TRUE every tenth time through the loop, due to the nature of modulo arithmetic.

It is also a good idea to put a condition on all your debugging SAY instructions:

```
if debug then say ...
```

By putting the assignment 'debug=1' at the start of your program, all of the debugging code will be activated. When your program works properly, you don't have to remove all of the debugging instructions, just change the debug assignment to 'debug=0'. You can then reactivate your debugging code at any time when new bugs appear.

Preventive diagnostics

You needn't limit your use of diagnostic SAY instructions to after a bug has been discovered. By using such checks right from the beginning as your program is being developed, you can find potential problems before they develop into hard-to-trace bugs.

Taking a little extra time to put in diagnostic code and extra checks for bad data while developing your program almost always repays itself in time saved in debugging. Debugging a program can often take longer than writing it in the first place, so spending programming time to save possible debugging time is usually a good trade-off.

Using TRACE

The tracing techniques just discussed are so commonly used by programmers and so useful to the debugging process that ARexx has extensive tracing facilities built right in.

Basic tracing

The TRACE instruction can be put in your program to activate one of several trace modes. In its simplest form, the TRACE instruction is used like this:

```
trace <mode>
```

Where 'mode' is one of the following symbols (in either upper case or lower case): ALL, COMMANDS, ERRORS, INTERMEDIATES, LABELS, RESULTS, SCAN. Any of these may be abbreviated to the first character only, such as TRACE I for TRACE INTERMEDIATES.

Any trace option can be selected at any point in a script. The trace mode selects which clauses in the script are traced and what information is displayed. TRACE ALL, for example, traces all clauses and displays each one as it is executed.

No matter which trace option is being used, one vital piece of information appears to the left of each clause: the line number in the script. This lets you locate the exact position of any traced clause in the script by going to that line in your text editor. The line number is your index to the code in your program.

A Sample TRACE ALL

Try running this simple (and quite useless) script:

```
/* TRACE ALL example */
trace all
count=0
do i=1 to 3
   count=count+1
   end
say 'Counted to:' count
```

The output to your Shell console window should look like this:

```
3 *-* count=0;
4 *-* do i=1 to 3;
  5 *-* count=count+1;
  6 *-* end;
```

```
     4 *-* do i=1 to 3;
     5 *-*  count=count+1;
     6 *-* end;
     4 *-* do i=1 to 3;
     5 *-*  count=count+1;
     6 *-* end;
     4 *-* do i=1 to 3;
     8 *-* say 'Counted to:' count;
Counted to: 3
```

Every time ARexx encounters a new clause, it is displayed along with its line number. If multiple clauses are found on the same line, each is shown separately. The 'nesting' level of each clause is also indicated by indentation. This has nothing to do with how the listing is formatted in your text editor, but shows how many control structures deep a clause is: in the example above, the clauses within the DO loop are shown indented one level deeper than the DO instruction itself.

TRACE ALL is used when you wish to see what statements in your program are being executed. It is analogous to the SAY program diagnostics shown earlier that merely report where they are.

Other TRACE options

Depending on the kind of bug you're trying to find, other TRACE options may be more appropriate. RESULTS, for example, will show you the result of every expression that is evaluated as well as the TRACE ALL information. This can help you track the values of variables and possibly see where things are going wrong.

INTERMEDIATES gives so much detailed information about the execution of every clause that even the simple script above generates over forty lines of debugging information.

Details about all of the TRACE instruction's options can be found in the entry for TRACE in the Reference Section.

Default TRACE mode

You might not have been aware of it, but if you've run any ARexx scripts at all you have already been using a TRACE mode: the default TRACE NORMAL. In this mode, whenever an external command returns a value greater than or equal to the current FAILAT option (default 10), the offending statement is displayed along with its line number in the script. (Command return codes and FAILAT are covered in the next chapter.)

This is a form of trace because the program line itself is actually displayed, not just an error message. If you disable tracing altogether with the TRACE OFF instruction, these line numbers will no longer be shown when such an error occurs.

Controlling tracing from your program

The fact that TRACE can be used as an instruction from within an ARexx script can cut out a lot of unwanted output. You can insert the appropriate trace instruction in your program just before a section of code you wish to check out, then disable it with TRACE OFF or TRACE NORMAL again at the end.

This will let your program operate normally while executing the parts of code you're not interested in analyzing, and let you see the debugging information only for the suspicious program lines. After seeing the output and determining that parts are operating properly, you can home in on the bug by tracing smaller and smaller sections of code.

The Global Tracing console

In our simple example above, the only output from the program was displayed by the final SAY instruction, coming after all of the tracing was over. If you are debugging a program that is printing information and maybe even getting user input from the console, a trace can be very confusing. With the program's output intermixed with the trace information, it can be hard to tell what's what.

For this reason, ARexx provides a "Global tracing console," which is a separate console window that displays the output from the trace of any ARexx program. (Even this can get confusing if you decide to trace more than one ARexx program at the same time, but that should be easy to avoid.)

The Global tracing console is opened by using the TCO ("Tracing Console Open") program supplied on the ARexx disk or in the 'Rexxc' directory under AmigaDOS 2.0. If you've installed ARexx properly, this program should be available and in your current command path. Simply type 'tco' at a Shell or CLI prompt to open the console window:

```
Shell> tco
```

You can move and resize the tracing console window as usual to get it out of the way or display as much data as you need.

You can take down the tracing console again using the TCC ("Tracing Console Close") program. The tracing console can be opened or closed in the middle of a trace, and it will respond almost immediately.

```
Shell> tcc
```

Keep in mind that the tracing console is just a facility that can be used by any trace modes that may be in effect. Opening the tracing console does not itself begin trace mode, and closing it does not stop the trace—it merely diverts the trace output back to the Shell window.

Interactive Tracing

For thorough analysis of a program's execution, you need to step through each clause one at a time, investigating variables and the results of expressions at will. This form of tracing, often called *single stepping*, is available in ARexx as *interactive tracing mode*.

Interactive tracing mode is activated with a question mark, and can be used in conjunction with any of the trace options. For example:

```
trace ?a /* interactive trace mode, ALL option     */
trace ?  /* interactive trace mode, same option    */
trace ?results
         /* interactive trace mode, RESULTS option */
```

Once interactive trace mode has been entered, TRACE instructions in the program will no longer have any effect, but you can still control the trace options interactively as explained below.

Interactive tracing is generally best used in conjunction with the global tracing console. All of your debugging is done in the console window, while the output from the program is displayed in the original Shell console window.

Using interactive tracing

Whenever a traced clause is displayed in interactive trace mode, the program stops and waits for your input before continuing. If you simply press RETURN, the program continues to the next traced clause and stops again. All of the standard debugging information is displayed in interactive mode, and what you see will depend as usual on the trace option currently in effect.

If you enter '=' before pressing RETURN, the currently displayed clause is executed again. Not all instructions can be executed twice, and interactive mode will not wait for input after a CALL, DO, ELSE, IF, THEN or OTHERWISE. These instructions are traced as usual, but ARexx won't stop and wait for your input until something else is executed.

The greatest benefit of interactive tracing mode is that you can enter any valid ARexx statements before continuing with program execution. This lets you examine the contents of any variables using

SAY, try out various expressions, call functions, and even perform DO loops and other compound statements using semicolons to separate the clauses.

Changing trace modes

When you encounter a TRACE instruction in your program while in interactive trace mode, it will be ignored by ARexx. This lets you continue interactive tracing without losing control. You can, however, change trace options by typing in the TRACE directly as an interactive-mode statement.

When entering a TRACE instruction in this way, it is important to understand the way the '?' option is handled. The question mark option toggles interactive mode, meaning that if you use it while in that mode already, the mode will be turned off. After the next statement is executed, the program will revert back to automatic tracing, and you'll lose control. To change a trace option while in interactive mode and stay in interactive mode, change the option without using the question-mark again.

Skipping over multiple clauses

When tracing DO loops or other repetitious program segments interactively, you can find yourself pressing RETURN over and over again, waiting for the loop to near its completion point. To help out in this respect, special TRACE instruction options allow you to skip over an arbitrary number of clauses. In other words, you can temporarily override interactive mode, but regain control again before the program goes too far and bypasses the code you're interested in tracing.

The number of clauses to skip is controlled using numeric arguments to the TRACE instruction. Remember, you can type the TRACE instruction—or any other statement—while in interactive mode.

Passing TRACE a positive number is like pressing RETURN that many times during an interactive trace. For example:

```
trace 5
```

will skip over the next five 'break-points,' places where ARexx would normally wait for your input in interactive trace mode. The output generated by the current trace option is displayed as usual, resulting in more debugging lines being displayed as higher numbers are used.

Using a negative number instead skips a specified number of clauses while suppressing any trace output along the way. When the final break-point is reached, the trace information for that clause (and other non-interactive clauses like DO, etc.) is shown as usual. For example:

```
trace -20
```

will skip over 20 clauses (usually 20 lines, unless you've used multiple statements on a line) without showing trace output until the last clause.

The TRACE function

A close cousin of the TRACE instruction is the TRACE() built-in function. This function takes as an argument a string containing the same option symbols as the TRACE instruction, and it returns a string representing the options in effect at the time the function was invoked. This lets you store the current trace options before setting a new one, then restore the original options again later:

```
trace_opt = trace('?All')
 /* interactive TRACE ALL in effect */
...
call trace(trace_opt)
 /* restore original trace mode */
```

There are other important differences between the trace function and the trace instruction besides the return value. For one, the trace instruction will change the trace options even during interactive tracing, where TRACE instructions are ignored. This is why the second call to the trace function works in the above script: if the TRACE instruction had been used instead, it would have no effect, since the program would be in interactive mode at that point.

Another difference is that the trace function takes a string or string expression as an argument, while the TRACE instruction takes a fixed symbol. The above example uses the contents of a string variable as trace options when restoring the trace mode to its original state. To do this using the TRACE instruction would require the use of the VALUE keyword.

When using the '?' option with the trace function, keep in mind that interactive mode will be toggled: if interactive mode is already on and you specify the question mark in another call to TRACE(), interactive mode will be turned off. The same behavior is not exhibited by the TRACE instruction because it is ignored in interactive mode.

Command inhibition

In the same way as the question mark turns interactive trace mode on or off, the exclamation point can be used in a trace mode option to turn 'command inhibition' mode on or off.

Command inhibition mode prevents the execution of external host commands. Normally any statement not understood by ARexx is sent as a command to the current host. It is up to the host what to do with the command, but while you're debugging a program, you may want to avoid sending any commands that are potentially dangerous.

For example, if you are testing a script that deletes files using the special COMMAND host to access AmigaDOS's *Delete* command, a bug in the script could result in incorrect files being deleted.

By turning on command inhibition mode, you can confirm that the logic of the program is correct, all variables are set properly, and the host commands will have their intended effect. Only once you're satisfied that the correct data will be sent to the command host do you turn off command inhibition mode and allow the ARexx script to access the host.

The Global Trace flag

With the various trace options, interactive trace mode, command suppression and the tracing console at your disposal, you might think that there are no secrets left for an errant program to hide. You would be wrong.

Even the best tracing facilities can't do you any good if you're not using them at the time a program exhibits a bug. When a program that isn't being traced gets caught in a loop, you need a way to 'break into' it and start an interactive trace. As you might expect, ARexx provides a way to do this.

Global trace: TS The TS ("Trace Start") program sets a global trace flag that all ARexx programs obey. Any ARexx program that is running when the TS program is executed will immediately break into interactive trace mode. RESULTS mode will be used (?R) unless the program is already in INTERMEDIATE or SCAN trace mode, in which case the mode will remain unchanged.

TS is used in a straightforward manner:

```
Shell> ts
```

Once in interactive trace mode, you can debug the program in the usual way. The global tracing console will be used if it is open. You can change trace modes or turn trace off while in interactive mode by using the TRACE instruction as usual.

End global trace: TE

The global trace flag remains set after running the TS program, which means that all future ARexx programs will break into interactive mode when they are run. To clear the global trace flag, use the TE ("Trace End") program. This will turn trace to OFF for any programs that switched to interactive tracing due to the trace flag being set. It will also restore the default trace mode to NORMAL rather than ?RESULTS for any programs run in the future.

Halting scripts: HI

Another way to get a program out of loop is to halt it using the HI ("Halt Immediate") program. This will immediately halt all currently executing ARexx scripts without entering trace mode, and can be useful to get out of a script that's 'stuck' or caught up in a long operation.

Error Trapping

A topic related to debugging and tracing is error handling: what control do you have over errors that cause a script to fail? ARexx's SIGNAL instruction provides an elegant way to trap errors and other conditions such as an external halt or Ctrl-C. The ARexx manual calls these traps *Interrupts*, but they have nothing to do with the interrupts used by the Amiga's hardware and operating system. To avoid confusion, we will simply call them *error traps* because of they way they catch an error condition in mid-program.

The SIGNAL instruction is used to specify what error conditions are to be trapped. When a specified condition occurs, control is transferred to a specially-named label in the program. This lets your script perform any 'clean-up' operations that might be required before exiting, like taking down windows or other resources that may have been obtained through external libraries or hosts.

SYNTAX error-trapping

As an example, the SYNTAX condition occurs whenever there is an execution error that would normally halt a script. If you would like the script to take some action before it halts (instead of displaying ARexx's error message), you can take care of that in your error handler. To trap syntax errors in this way, you would use the following statement near the beginning of the script:

```
signal on syntax
```

With the syntax signal enabled, the script's syntax error handling code will be executed whenever an error occurs anywhere in the script. The error handling code could be as simple as this:

```
syntax:
   say 'Trapped Error:' errortext(rc)
   exit 20
```

All this does is display a customized error message before exiting with a return code of 20. Try using the above code in a script that contains the line '=3' or something similar, which is total nonsense as an ARexx statement. The script will report "Trapped Error: Invalid expression."

The error message comes from invoking the ERRORTEXT() function with RC as an argument. After a syntax error, the variable RC contains the error number. The message corresponding to the error can be determined with ERRORTEXT(). (RC in this case has a different meaning than it does after a host command is executed, in which case RC is the error severity code from the command.)

SIGNAL Conditions

Other conditions that can be trapped with the SIGNAL instruction:

SIGNAL ON ERROR: Host command return codes (RC) greater than zero cause this condition, usually indicating an error or warning of some kind.

SIGNAL ON FAILURE: Host command return codes (RC) greater than the current FAILAT level (default 10) cause this condition. Unless trapped, this produces a "Command returned..." error message in normal (default) trace mode.

SIGNAL ON HALT: External halt via the HI program, normally resulting in "Execution halted" message.

SIGNAL ON IOERR: Errors that are detected by ARexx's input/output routines such as READCH() and WRITECH().

SIGNAL ON NOVALUE: Uninitialized variables can normally be used as strings containing the name of the variable itself in upper case. With the NOVALUE signal turned on, such usages cause an error trap.

BREAK_C, BREAK_D, BREAK_E, BREAK_F: Breaks from AmigaDOS

See the SIGNAL entry in the reference section for detailed information.

Trapping multiple conditions

Several different conditions can be trapped simply by specifying more than one SIGNAL instruction. A script might very well wish to trap SYNTAX, ERROR, and HALT conditions to ensure that the script does not exit without special clean-up code being invoked.

The labels for each condition could indicate separate clean-up routines, or all point to the same code like this:

```
syntax:
error:
halt:
   call cleanup()
   exit 10
```

Special variables

You've already seen the use of the RC variable in SYNTAX error traps. RC can also be used in an ERROR or FAILURE trap to see the value returned by the command that caused the error.

Another useful variable that works in all types of error trap is SIGL. SIGL contains the line number of the statement in the script that caused the error, and can be used in error messages. A complete SYNTAX error report might look like this:

```
syntax:
   say 'ERROR TRAP:' errortext(rc) 'in line' sigl
```

This would result in a message of the form:

```
ERROR TRAP: Invalid expression in line 8
```

Turning signals off

Individual error traps can be disabled using the OFF keyword:

```
signal off syntax
signal off error
```

Program-generated error traps

SIGNAL can be used in an alternate form which causes a 'jump,' or transfer of control, to any label in the script. To perform the current FAILURE trap, for example, you could use the statement:

```
signal failure
```

This can be used to transfer control from inside a loop or function to any point in a script. An application for this is general error handling: you could create a label called 'BOMBOUT:' and then perform a 'SIGNAL BOMBOUT' if the program detected some disastrous error that couldn't be handled by normal means.

This use of SIGNAL is similar to a GOTO in BASIC and other languages. In ARexx, other structures like IF...THEN and DO...WHILE are provided to handle ordinary flow control requirements. Using SIGNAL as a GOTO can be appropriate in some circumstances, but it should only be applied when necessary, since it it can make your program's logic hard to follow.

Another variant of this is the 'computed goto', using an expression to form the label name in the SIGNAL instruction. This is done using the VALUE keyword, as in:

```
signal value 'label'i
```

The above statement could cause an immediate transfer of control to 'label1:', 'label2:', or 'labelXYZ:', depending on the value of the variable I. The use of SIGNAL in this way should be applied carefully and documented clearly inside the script, since it can make a script difficult to debug.

Uses for error traps

We have already mentioned 'cleaning up' chores as a good use for error traps. Another common technique is to use a FAILURE trap to exit with the current value of RC:

```
signal on failure
...
failure:
   exit rc
```

This is useful when scripts are being called as commands from other scripts (possible when using the default host address REXX). If any script fails by exiting with a return code greater than the current FAILAT level (normally 10), the script calling that script will fail as well. If all the scripts in use have the failure trap in place, the error will eventually get back to the original caller, and no additional code will be executed after the error has taken place.

This is particularly useful in systems that use scripts to add new commands and whose host commands return error codes to indicate a user-abort. An example is *IntroCAD Plus* (Progressive Peripherals and Software), discussed further in Chapter 13.

SECTION III
CONTROLLING APPLICATIONS

Chapter 12
External Control

ARexx Communication

In Chapter 3, you saw how ARexx can be used to send commands to host programs that are currently running in the system. It is the Amiga's multitasking operating system that allows this process to take place.

Multitasking

You are probably familiar with multitasking on the Amiga; multitasking allows you to use the Workbench while your Word Processor is running, type Shell commands while downloading a file with a telecommunications program, or edit text while a 3D solid model is being ray-traced, just to name a few possibilities. Besides the applications that you've run yourself, there are also a number of 'handlers' and 'devices' working quietly in the background, also multitasking. All this is possible because a part of the Amiga's operating system called *Exec* keeps track of all tasks and switches control of the computer from one to another many times a second. This makes it appear that all the tasks are running 'at the same time'. When a program is just awaiting user input and not doing anything—like the time a word processor spends waiting for you to press a new key—it doesn't use up any computing time at all. This means that running several programs at once will often result in very little slowdown in the general operation of the computer. Remember, the delay between two typing keystrokes may seem like a short period of time to you, but a lot of computing can be accomplished in that fraction of a second.

Multitasking is what enables ARexx to blend in so seamlessly with other programs as they work, even allowing an ARexx script to work as part of an application program. The ARexx task simply runs at the same time as the other tasks, only using CPU time when instructions are actually being executed. But multitasking alone isn't enough: ARexx has to communicate with the host application somehow.

Inter-process communication

Communication between programs in the system—called *inter-task* or *inter-process* communication—is made possible by another *Exec* software mechanism called *Messages* and *Ports*. A Message is a specially organized group of data, and a Port is a collection area for messages. Ports can have names, which is how an ARexx host is addressed: the ARexx ADDRESS instruction specifies the name of the message port for ARexx command messages to be sent to. Chapter 3 contains some simple experiments that illustrate the use of this command; it is covered in more detail later in this chapter.

Communication between ARexx and a host application goes back and forth in a number of ways. When you run a script using the RX program from the Shell, special commands are sent to the current host (specified by ADDRESS), and results from those commands are sent back from the host to ARexx. (Programs that accept ARexx commands externally like this are sometimes called ARexx 'servers'.) When you run an ARexx program as a macro from within an application, the application sends the macro to ARexx to be executed, where it behaves in the same way as if it were run using RX.

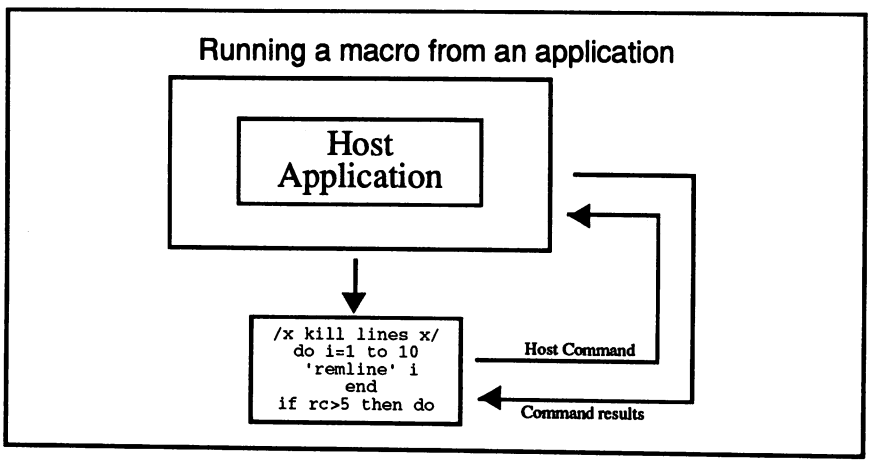


Figure 12-1

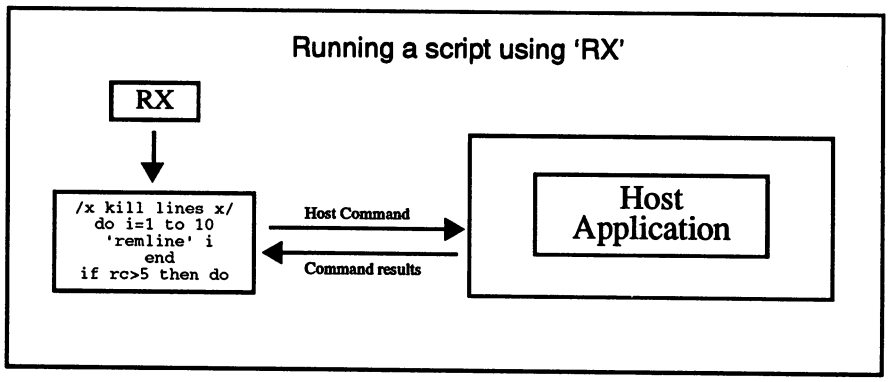


Figure 12-2

Using ARexx Commands

In Section II of this book, you learned about ARexx programming. The language that you learned was the core of ARexx, but ARexx can be extended by using commands supported by a host application. These commands are specific to each host and are explained in the relevant documentation for the application, but there are some general rules about how host commands are used in your program.

Addressing the host

You have already sent simple commands to host applications in Chapter 3, where you were introduced to the ADDRESS instruction for establishing the current host address. The *host address* (the ARexx message port name) for an application must be known before you can control that application with an ARexx program. The ADDRESS instruction is used to tell ARexx the host address to send host commands to. A host address is just a simple name unique to each host application; the application's documentation will tell you its host address. The host address, or name, is case-sensitive, so you must use the name in the ADDRESS instruction exactly as indicated in the documentation.

The ADDRESS instruction

The ADDRESS instruction can be used in a number of ways. The simplest form is the one already explained: giving the name of the host to which all future host commands are to be sent. For example, to address *Art Department Professional*, whose host address is 'ADPro', you would use the following ARexx instruction (note the use of quotes around the host name in order to preserve the case of the letters):

```
address 'ADPro'
```

This sets the name of the *current host*, but ARexx also remembers the previous host address. You can switch back to the previous host simply by giving the ADDRESS instruction with no host name. This feature can be used when a script is communicating with one host, but needs to send a few commands to an alternate host. The ARexx script would go like this:

```
/* commands to current host */
...
address 'Host 2'
/* commands to 'Host 2' */
...
address
/* communicating with original host again */
...
```

If you wish to send a single command to an alternate host, you don't even need to switch back and forth using two ADDRESS instructions. Another form of the instruction lets you send a command to any host without changing the current host address. The host command is given immediately after the ADDRESS instruction, like this:

```
/* commands to current host */
...
address 'Host 2' 'QUIT' /* send QUIT to 'Host 2' */
/* commands to original host */
...
```

Host commands

The simplest form of host command is just a string that appears where an ARexx instruction or assignment clause would normally go. So, the following line in an ARexx program:

```
'quit'
```

...would result in the string 'quit' being sent as a command to the current command host.

If the string was not in quotes, ARexx would first attempt to interpret it as an internal instruction. If it was not a valid ARexx instruction, the symbol would be taken as the name of an ARexx variable, and the contents of the variable substituted for the command. If no variable with that name was defined, the instruction would simply be translated to upper case; this may or may not make a difference, depending on the host. In the case of using 'quit' without quotes, the command 'QUIT' (in upper case) would be sent to the current host providing there is no variable called 'QUIT' currently defined. Although host commands will usually work properly when used in a script without quotes and many ARexx scripts use commands in that way, it is good practice to use quotes around all host commands to avoid possible ambiguities.

Most hosts ignore the case of the letters used in commands, but some are fussy about the capitalization. You'll have to consult the host software documentation—or experiment—to find out.

Using variables as commands

We have recommended putting host commands in quotes to ensure that there is no confusion with ARexx instructions or variables. In some rare cases, however, you may wish to perform variable substitution in giving a host command. Consider the following script:

```
/* extract an archived file */
address command   /* use AmigaDOS commands */
say 'extract file using ARC or ZOO?'
pull arcprg
say 'name of file to extract?'
pull filename
arcprg 'x' filename  /* ARCPRG defines the command */
```

In this script, we use the contents of the variable 'arcprg' as a host command; the host in this case is AmigaDOS, so we execute the program name provided by the user of the script. The user can type ARC or ZOO to extract an archived file (whose name must also be provided) using the appropriate utility. (In this simple example, there is nothing to stop the user from typing something like 'DPaint' and running that program instead!)

This example is mainly to show how ARexx handles command names that are used without surrounding quotes. In reality, the above technique is of limited usefulness, and can make scripts harder to understand.

Forcing command recognition

When using variable names as commands, you can distinguish the command from an internal ARexx instruction by putting an empty string at the beginning of the command, like this:

```
''arcprg 'x' filename
```

This would have the effect of forcing the line to be interpreted as a command to an external host rather than as an ARexx instruction. You can use this technique whenever you are putting together a string to form a host command that may be confused as an ARexx instruction, label, or assignment.

Command arguments

Some host commands like the QUIT example used above will achieve their desired effect by simply issuing the command alone. Other simple host commands might select a menu option or other function for a specific application. The paint program *DigiPaint 3*, for example, uses simple commands (called *action codes* in the *DigiPaint 3* manual) like 'Undo', 'Scis', and 'Clrs' to emulate buttons that can be clicked on the program's control panels.

Many host commands, however, will require arguments that provide additional data to the command. The 'extract archive' script above, for example, provided arguments to the command (the string 'x' and a file name). Similarly, the *DigiPaint 3* 'Move' command requires X and Y coordinates as arguments.

Arguments are simply specified as a single string after the command, which is passed on to the host. The host then extracts the individual arguments from the string, which are usually separated by one or more spaces. Since you are simply giving a regular ARexx string in the command, you can use a string expression in the usual way, combining constants, variables, functions, and any other elements of an expression. In most cases, you'll simply put each argument side by side, separated by a space—ARexx will take care of putting these together to form a string.

For example, the following ARexx script could be used to send a valid *Move* command to *DigiPaint 3*:

```
/* center mouse pointer in DigiPaint 3 */
address 'DigiPaint'
x = 160
'Move' x 100
```

The above code selects *DigiPaint* as the host, and sends a *Move* command that results in the mouse pointer being placed in the center of the screen. You will notice that the X coordinate was given using the variable 'x', while the Y coordinate was specified as a constant. ARexx turns the 'x 100' part of the command into '160 100', following the usual rules for expression evaluation. Along with the 'Move'

command, *DigiPaint 3* is sent the argument string '160 100', from which it extracts the two values X and Y.

Each of the following commands would result in exactly the same argument string being sent to the host:

```
'Move' 160 100
'Move 160 100'
x=50; 'Move' x*2+60 left(x*20,3)
```

The particular arguments required for each host command, and whether the arguments are numbers or strings, is documented in the ARexx command reference portion of the application software's manual. We will cover several popular applications in the next chapter.

Other forms of commands

The form that a command takes is totally determined by the host application; not all hosts will necessarily use the standard format of a command name followed by arguments. *IntroCAD Plus* from Progressive Peripherals and Software, for example, uses concise symbols for some commands. A menu is selected by the menu name preceded by a slash, for example:

```
'/draw/box'
```

Less conventional is *IntroCAD Plus*'s command to place the mouse pointer (point and click) at a specific screen coordinate. The '!' symbol is used, but it is placed *after* the X and Y coordinates. Such a command might look like this:

```
xcoord ycoord'!'
```

Since the variable names are not ARexx commands and the variables contain numbers, ARexx does not recognize the line and sends it off as a host command. To be sure of avoiding errors when using such unusual command formats, you may wish to use the empty-string technique to force host command recognition. The above command would become:

```
''xcoord ycoord'!'
```

Using scripts as commands

Using generic ARexx programming and using scripts without the ADDRESS instruction, you were able to execute an ARexx script just by giving the name of the script as if it were an instruction; any instruction not understood by ARexx was taken as the name of a script in the current directory or 'REXX:' directory. When you switch the host address with the ADDRESS instruction, commands are sent to the new current host instead of to ARexx's internal message port, so you lose this ability.

In order to execute scripts as commands while communicating with an external host, you can send the commands directly to ARexx's port like this:

```
address 'REXX' mycommand
```

This will execute a script called 'mycommand' or 'mycommand.rexx' in the current directory or 'REXX:' directory, and pass along any arguments that may be provided. This allows you to add your own commands by writing scripts, while still using the commands provided by the current function host.

To switch back to the ARexx port permanently and stop sending commands to the current host, just use ADDRESS in its alternate form:

```
address 'REXX'
```

Results from commands

After giving a command to a host, you often need to determine what happened as a result. In some cases, your ARexx program simply needs to determine whether an operation was successful. In other cases, the command may have been given solely to determine some information from the host, like the contents of a database field or the current colors used in the display. Getting information from a host can be done in a number of ways, which will depend on the particular host's implementation.

Return codes

Return codes are a built-in feature of the ARexx language. When trapping syntax errors with the SIGNAL ON SYNTAX instruction, the variable called 'RC' can be used to find the specific error that occurred. The same variable can also be used to determine the error status from host commands.

Most hosts will use this as an indication of the success or failure of a command, but any value at all can be returned. When used as an error indicator, RC will contain zero if the command worked without a

problem, or a nonzero value indicating the severity of the error if there was a problem. When using commands that invoke major operations like loading files or creating new data, your ARexx scripts should check the RC variable before assuming that the operation was performed; disk errors or low memory conditions could have prevented the command from working. The actual error codes returned from specific commands will depend on the host application, and will be documented in the ARexx section of the user's manual. Generally, lower values (usually less than ten) are just warnings, and higher values mean failures or errors of some kind.

As an example of using error return codes, consider the following script, used to load a given picture name into *Art Department Professional*:

```
/* load given IFF file into
   Art Department Professional */
parse arg picname
   /* get picture name from command line */
address 'ADPro'
   /* talk to art department professional*/
'lformat IFF'      /* set load format to iff        */
'load' picname 'NOPAD' /* load picture into adpro  */
if rc>0 then do    /* can't continue if load failed */
    say 'Sorry, couldn't load picture!'
    exit
    end
```

This script can be used from a Shell command, using the *RX* program to load a file into *Art Department Professional* (ADPro) if it is currently running. Additional instructions could perform various image processing operations on the loaded picture, but only if the picture loaded successfully. If the script were saved in your 'REXX:' directory and called "LoadPic.adpro," you could load a picture called "SamplePic.IFF" into *ADPro* as follows:

```
Shell> rx loadpic.adpro samplepic.iff
```

You can see in the script listing that two commands are given to the host: 'LFORMAT IFF' sets the load format to an IFF picture file, and 'LOAD' loads the specified picture file, using the 'NOPAD' option, also given as an argument. After the LOAD command, the 'RC' variable is examined for a non-zero value to see if the load was successful; if not (rc > 0), the script displays a message to the user and exits before continuing with any further operations.

Whether return codes are set, and how they are set for individual commands, depends on the host. Some host programs may use RC for purposes other than error codes: the *Baud Bandit* telecommunications program from Progressive Peripherals and Software, for example,

returns information like baud rate, screen depth, etc. from some commands in RC. Generally, such usage seems to be the exception rather than the rule, as RC is intended as an error indicator.

Failure levels

If you have *Art Department Professional* and you tried the above script to load in a picture, you may have noticed what happened if you gave it the name of a file that does not exist. You get the following messages displayed in the Shell window:

```
    5 *-* 'LOAD' picname 'NOPAD';
+++ Command returned 10
Sorry, couldn't load picture!
```

The first two lines of this message are an error report from ARexx: since the host command returned a nonzero 'RC' value, ARexx took this to be an error. The script is not halted as it would be with a syntax error, but a message is displayed showing the command that caused the error along with the return code—10 in this case. In an ARexx script like this that checks the error code and deals with it itself, you may not want ARexx interfering by displaying error messages. You can control the level at which ARexx considers a return value an error using the OPTIONS FAILAT instruction. For example:

```
options failat 20
```

If you put this instruction near the beginning of a script, only return values from host commands of 20 or greater would be considered an error by ARexx. If you put this instruction into the *ADPro* script above, only the script's "Couldn't load picture!" message would be displayed, without the extra ARexx messages; a return code of 10 would no longer be severe enough for ARexx to be concerned.

You can also trap host command error returns in the same way as other interrupts, using the SIGNAL ON ERROR instruction. See the the Reference Section on the SIGNAL instruction for details.

The RESULT variable

Some hosts may need to supply more information than simply whether a command succeeded or failed. The standard way for results to be returned is in the special RESULT ARexx variable. In order to be able to read this variable after giving a host command, you must use the OPTIONS RESULTS instruction in the ARexx program before the host command is given. The following line should be placed near the beginning of any ARexx programs that communicate with hosts returning results:

```
options results
```

This instruction causes ARexx to request results (if any) whenever a host command is sent. After giving a host command, any desired

result would be in the RESULT variable; what results you can expect to find there will depend on the command itself. RESULT might contain the text for an error message if a command failed, or it might contain information that you requested with a previous command. Therefore, it is important to check the value of RC before interpreting the results of a command that may fail.

As an example of using the RESULT variable, consider the following simple script for use with *Bars & Pipes Professional*:

```
/* simple bars and pipes example */
address 'Bars&Pipes ARexx'
  /* talk to Bars&Pipes program */
options results /* use result variable              */
'frame'         /* ask b&p for current frame type  */
say result      /* show result from frame command  */
'tempo'         /* ask b&p for current tempo       */
say result      /* show result from tempo command  */
```

The FRAME and TEMPO commands as they are used here have no other function than to return a result. The only way you can use this result is by first using the OPTIONS RESULTS instruction, then examining the RESULT variable after giving the command. If you were to remove the OPTIONS RESULTS instruction in the above script, ARexx would not ask *Bars&Pipes* for results from the commands, so none would be returned—the output from both SAY instructions would simply be 'RESULT', since the variable is not initialized. With the script as listed, the values '24' and '120' are displayed, the default values for frame type and tempo, respectively.

Multiple results Some host commands may need to return more than one number or string. This can be done by simply putting all of the values together in a string, separated by spaces; the string is read from the RESULT variable as usual. Your ARexx program can easily read these individual values from the RESULT variable by using ARexx string functions or by using the PARSE instruction.

As an example, the *TurboText* text editor by Oxxi Inc. provides over 140 ARexx commands to control many aspects of the text editor. Some of these commands are used to get information, and they return results in the RESULT variable as usual. The 'GetCursorPos' command, for example, returns three values in RESULT: the current cursor line, cursor column, and an ON or OFF value indicating whether the cursor is on a "fold" or not. We could put this information into separate variables as follows:

```
/* ARexx macro for TurboText */
address 'TurboText0'
'GetCursorPos'
```

```
parse var result line column fold .
```

After the PARSE instruction, the LINE and COLUMN variables will contain the cursor's line and column position respectively, and the FOLD variable will contain the string 'ON' or 'OFF'. The script can continue using these variables, and forget about the RESULT variable until it's needed for another command.

Results in other variables

Some host applications return results in other variables, which will be documented in the ARexx section of the application's manual. Like the special RESULT variable, these variables can be set to any string, giving any information required. As an example, *Art Department Professional* uses a variable called ADPRO_RESULT to return information after many commands. The use of this variable is also dependent on the presence of the OPTIONS RESULTS instruction. The information supplied in ADPRO_RESULT depends on the command, but it is often used to give an explicit error message explaining why a command failed (whether the command failed or not is indicated by the RC variable).

An even better example of using variables to get information from a host can be found in Software Visions' *Microfiche Filer Plus* (MFFP) database program. Some ARexx commands in *MFFP* accept the stem name of a compound variable, and set the value of nodes in the compound variable. For example, using *MFFP*'s 'get' command, you can examine fields in a record like this (this will only work from within *MFFP*, not as an external RX script):

```
/* show first field of selected record */
'first'      /* move to first selected record */
'get' rec
   /* put contents in 'rec' compound variable */
'display' rec.1.value
   /* display value first field in rec */
```

This method of returning (and sending) information is a special feature of *MFFP*, and does not require the RESULTS option.

Function Hosts

Function hosts are different from command hosts in that they work by adding new functions to ARexx rather than by receiving host commands as messages. Function hosts do not need to be addressed with the ADDRESS instruction; whenever a function host is running, its extra functions are available to any ARexx program. Results can be obtained by calling functions provided by the host. Functions, rather than commands, are used to perform any operation that might be required.

An example of an application that operates as a function host is Gold Disk's *HyperBook* hypermedia program. Some of its functions, like

NEXTPAGE() and PREVPAGE(), are simply used as commands (with the ARexx CALL instruction) and require no arguments. Others, like GETWIDTH() and SEARCHTEXT() are used for the results that they return. The advantage of a function host over a command host is that the functions can be used like ordinary ARexx functions in expressions, making it easier to use information from the host in your ARexx programs. The only possible problem with the use of a function host arises when more than one host uses the same name for a function; function hosts can get around this problem by using a special prefix in the function names when the functions are used by external scripts.

The use of functions instead of commands can make your scripts more concise. For example, a script to increase the value of a page color by one might look like this using a conventional command host:

```
/* change page color - command host */
address 'host address'
options results
'getpagecolor'
'setpagecolor' result + 1
```

Whereas using a function host, the script would be simplified:

```
/* change page color - function host */
call FH_setpagecolor(FH_getpagecolor() + 1)
```

ARexx in Applications

How ARexx is supported can vary considerably from one application to another. This is because ARexx can be useful in different ways with different programs, or perhaps the vision of the software designers may have just been different. Whatever the cause, it is helpful to be aware of the number of general ways ARexx can be used.

Terminology

When we talk about macros and scripts, we are talking about ARexx programs. Some applications may use the same terms differently in their documentation and user interface, however. A program that has its own script language that can be used independently of ARexx (e.g. telecommunications software) will often refer to these programs as 'scripts'. Similarly, a program that has the capability to record a sequence of operations and play them back (e.g. text editors) may refer to these as 'macros'. When ARexx is supported as well, ARexx scripts and macros will usually be referred to explicitly, as in a menu item that says 'Execute ARexx' or something similar. You should be aware that the mention of 'script' or 'macro' in a program does not necessarily indicate ARexx-compatible features.

External scripts vs. built-in macros

Some applications provide an ARexx communications port, but do not have the facility to run ARexx macros from within the application itself. With these applications, you must save your macros as separate scripts, and execute them using the RX command from the Shell. This involves switching from the application to the Workbench screen and typing a command every time you want to use a macro. An example of a program that supports ARexx in this way is NewTek's *DigiPaint 3*. This approach makes more sense in applications that use ARexx as a means of being controlled from other programs, rather than as a macro language. An example of such an application is *Bars & Pipes Professional* by The Blue Ribbon Soundworks Ltd. *Bars&Pipes Professional* has ARexx commands to control the playing of songs so that you can coordinate your MIDI instruments with a multimedia presentation; ARexx is not used as a macro language in controlling the user interface of the program itself.

A more common approach is for an application to let you run ARexx programs as macros, but the programs themselves are stored externally to your project, usually in the 'REXX:' directory. This makes accessing macros from within the application more convenient, and is a good solution for applications where the macros used are not expected to vary from project to project. It can result in a great many macros to look through in your 'REXX:' directory, however.

A more all-encompassing approach is for an application to have built-in macro storage capabilities, and let you run these macros from a menu, control panel, or the keyboard. A complete implementation also allows the creation and editing of macros, either with a special text editor built into the application, or by invoking your standard system text editor (*Ed* or the editor of your choice). Any macros you create will be automatically saved with your current project. This is good for applications like authoring and presentation systems that often use macros specific to a project.

Many host applications do not require the use of the ADDRESS instruction in macros; the application launching the script is the current host by default. You may wish to put the ADDRESS instructions in your macros anyway, in case you wish to launch the script from the Shell or from another application.

Macro names

When macros for a specific application are stored externally, it is useful to be able to distinguish these macros from those intended for another application. Since host commands are unique to each application, looking at the file names in your 'REXX:' directory could be confusing if there were macros for a number of different applications: you wouldn't know which macros were intended for use with which program.

Fortunately, ARexx has a way of dealing with this problem: it is done through file name extensions that are unique to an application. You have already learned that generic ARexx scripts (those without host-specific commands) often use the '.rexx' extension in the file name to distinguish the file as an ARexx program. Other extensions can be used for macros for a specific host: ARexx scripts designed for *Art Department Professional*, for example, can use the extension '.adpro'. When the scripts are used as macros from within the application for which they are intended, the special extension is used as an alternative to the standard '.rexx': the scripts can be called by their basic

names, without adding the extension. Applications that support this feature will give the extension to use in their documentation.

Even if this extension feature is not explicitly supported by an application (meaning you'll have to use the extension as part of the macro name), it is a good idea to use extensions in the file names of ARexx scripts in your 'REXX:' directory that are designed for a specific application. This will prevent confusion when looking through your 'REXX:' directory, and let you distinguish application-specific macros from generic ARexx programs that don't require a host.

Executing macros

The ARexx macros that you use from an application can be selected and executed in a number of different ways, depending on the application. Some programs, like ASDG's *Art Department Professional,* let you run ARexx programs by pressing function keys; the ARexx programs must be saved as specially-named scripts in the 'REXX:' directory. Other programs let you select a macro by typing its name or selecting the macro from a file requester. For greater convenience, some applications allow a combination of methods; ASDG's *CygnusEd Professional* text editor, for example, allows file-requester selection of ARexx scripts, but also allows you to 'install' a number of macros or ARexx commands that can be selected from a menu or by function keys.

Another approach is the provision of an ARexx 'console window' for executing macros. The console window is like a Shell window, but you can execute macros for the host application just by typing their names, as you would with AmigaDOS commands in a Shell. The advantage of an ARexx console window over function-key or menu selected macros is that you can supply arguments to the scripts on the command line. Examples of applications that provide an ARexx console are *IntroCAD Plus* and *TurboText.*

Other programs let you blend ARexx macros into your current project more seamlessly: *Nag Plus,* the appointment schedule program from Gramma Software, will execute an ARexx script when you click the right mouse button over an appointment cell, passing it appointment information. Gold Disk's *Home Office Advantage* spreadsheet program lets you assign an ARexx macro to cells in a spreadsheet and run the macro by double-clicking the cell. Hypermedia programs like *AmigaVision, CanDo, HyperBook* and others let you execute specific macros by clicking on preset 'hot spots' on the screen. Presentation

programs like Gold Disk's *ShowMaker* can execute ARexx macros at any point in a presentation.

Since there is no hard rule about how ARexx macros are used, you should consult the ARexx section of the application's documentation to learn the specific implementation.

Where do macros go?

Many applications simply assume that macros will be stored as separate script files in the 'REXX:' directory, along with all other ARexx scripts. The script file is loaded from disk every time the macro is invoked. Other applications may let you 'install' commonly-used macros in memory so that they can be executed instantly. This can sometimes also be done by defining short macros as 'string files', which are just miniature ARexx programs stored in memory.

When programs are stored as external files, you may want to put macros for some applications in separate directories (or disks) and re-assign 'REXX:' when using that application; this prevents your standard 'REXX:' directory from becoming cluttered and filling your main storage device with macros from different applications.

Even if the macro files you wish to access are not stored in the 'REXX:' directory, you can usually use them by giving the entire path name of the macro from within an application. For example, if you keep all of your *Microfiche Filer Plus* macros in a special 'MFF' directory on your 'DH0:' hard drive, you could call up a macro called 'reorder.mffm' (for example) by using the name 'dh0:mff/reorder' from the application ('mffm' is the standard extension for *Microfiche Filer* macros).

Output from macros

When developing macros, it is often convenient to use the ARexx SAY command to display values that show you what's going on in the program. When you launch a macro from an application, where does this information go? As with most of these issues, this too depends on the particular host being used.

In most cases, if the host was launched from the Shell (by typing its name or using the RUN command), any standard output from ARexx macros will go to the original Shell window. If the application was launched from the Workbench, there may be a special output window on the Workbench screen where the ARexx output (and all other

stdout output from the application) will be displayed. If not, you may not be able to see your ARexx macro output at all. At least one application recommends launching it from the Shell specifically for the purpose of getting ARexx macro output for debugging purposes; after the macros are developed and the output is no longer required, you can run the program from the Workbench.

Another approach is for a program to have a built-in facility for displaying the output from ARexx macros. *HyperBook* shows ARexx output in a requester after the macro has finished; output too large to fit in a requester is displayed using a text-reader utility. Similar facilities for displaying messages to the user are provided in a number of other applications.

In many cases, you won't ever need ARexx output in macros. If you do, and your application does not show it to you, try launching the application from the Shell and look at the Shell window for the output. You should also be able to use the Shell window for ARexx console input (using the PULL instruction in the macro).

What ARexx controls

Whether you control an application through external scripts or internal macros, what really counts is what you can actually do with the application's supported ARexx commands. Different applications let you control their functions with ARexx in different ways:

• **User interface access:** programs that are extremely interactive like graphics software often provide ARexx commands to perform the equivalent of menu and mouse controls that the user could do directly. This often includes pushing the application's screen behind or in front of all other screens to 'pop up' and 'put away' the application from an ARexx program.

• **Specialized commands:** some applications like spreadsheets and database programs add ARexx commands that let you work directly with the data, rather than simply mimicking the user interface. This is often more convenient when writing ARexx programs than thinking through the manual operations that would be necessary to achieve a particular operation. Some programs may take this one step further and add capabilities that are only accessible through ARexx commands; an example is user input facilities in a hypermedia program that allow it to be used as an 'authoring' system.

• **Script control:** programs with their own script language, like some telecommunications software, may provide all or some of the script-language commands as ARexx commands, giving you the choice of implementing a script in ARexx instead of using only the native script language facilities. This kind of ARexx support is ideal, since you get the benefit of a wide range of application-specific commands coupled with the flexibility of the ARexx language.

Controlling one application from another

Besides its use as a standardized macro language, ARexx is important on the Amiga because it gives you the ability to make two or more separate programs work together. For example, you might create a multimedia production using a presentation tool like Gold Disk's *ShowMaker* or Commodore's *AmigaVision*. Using ARexx scripts that address other host applications, you could control external MIDI musical instruments by communicating with *Bars&Pipes*, or even control NewTek's *Video Toaster* running on another Amiga for spectacular video effects.

Chapter 13. Specific Applications

You've seen how ARexx is more than just an ordinary computer language because of the way it can control other programs. What this means to you as a computer user is that ARexx programming is something you do all the time, not just when you are creating your own programs from scratch.

Because ARexx is so well supported by major software applications, you can use it to make just about everything you do with your computer more automatic, faster, easier, and—not least of all factors—enjoyable. Whether you are word processing, telecommunicating, animating, composing, using a database or a spreadsheet, there are ARexx-compatible programs that let you tap their power directly by using custom-written macros.

Since each application uses its own set of ARexx commands and uses ARexx in a slightly different way, it's worth taking a look at each application individually.

This chapter examines a few ARexx-compatible programs in each of several application categories. This sample of available software titles makes no claim as to completeness, but is a good representation of what is available. Our hope is that by collecting information about many Arexx-supporting products in one place, readers will begin dreaming up possibilities for Arexx-orchestrated multi-program applications. This is one area where the Amiga platform beats all others, and is especially important for the kinds of multimedia productions that are being developed today at an accelerated pace.

Word Processing

In word processors, ARexx is usually applied as a macro language for defining common operations. By combining the program's basic text editing and formatting operations in ARexx scripts, you can design macros to automate common sequences of operations.

Another way that ARexx can be useful in a word processor is to connect the program with another text application using inter-process

communication. An example is "hooking up" a thesaurus or grammar checker.

ProWrite 3.1
New Horizons

Host Address:	"ProWrite"
Added commands:	111
Macro support:	Run macros by file name or by function keys
ARexx applications:	text editing macros; hook-up to external programs

ProWrite uses ARexx as its standard macro language, and includes ARexx commands to control just about every feature of the program. ARexx commands for menu selections, cursor movement, and text extraction and insertion are all provided. Macros are stored in separate ARexx scripts in the same directory as the ProWrite program. Note that this differs from the usual practice of putting scripts in the REXX: directory. The macros are run by selecting *Other* from ProWrite's *Macro* menu (right-Amiga M) and typing the script's file name. Also, up to ten macros can be run just by pressing SHIFT and a function key: these macros must be named 'Macro_1', 'Macro_2', etc. in the ProWrite directory.

When running macros from within ProWrite, it is not necessary to address the 'ProWrite' host. Scripts being run externally (using RX) must, of course, use the ADDRESS "ProWrite" instruction before issuing any ProWrite ARexx commands.

The easiest way to create macros in ProWrite is to open a new document window, type the ARexx script in as a regular document, and save it (to the ProWrite directory/drawer) using the 'text only' option.

There are ARexx commands to access most of ProWrite's features and to control cursor movement. Slightly different are the EXTRACT and TYPE commands, which deal with the actual text in the document. The EXTRACT command will put selected text from the document into an ARexx string, where you can parse and process it, then re-insert it into the document with the TYPE command. Unfortunately EXTRACT stops at the end of a paragraph, limiting the amount of text you can get from a document. EXTRACT can still be useful for working with small amounts of text, however. An example is the sample script included in the ProWrite package called *Math*.

Using a macro in ProWrite

The *Math* macro is a script found on the ProWrite release disk. Make sure this script is in the same directory as the ProWrite program, then try the following example. This macro evaluates the selected text as a mathematical expression, and inserts the result of the expression in the document. Since the highlighted text cannot span multiple paragraphs, the *Math* macro can't be used to add columns of numbers. It is useful, however, as a quick calculator. Try the following in ProWrite:

- Enter an expression into a document window, for example '15692.27/12'.
- Select the entire expression by clicking and dragging with the mouse.
- Execute the *Math* macro: select 'Other...' from the 'Macro' menu (or press right-Amiga M), then type 'Math' and press return.
- The text in your document will be modified to read: '15692.27/12=1307.68917'

Shortcut macros

If you found the *Math* macro to be useful enough that you used it all the time in your work, you might want to use a function key to instantly run it instead of selecting a menu option and typing 'Math' every time. To do this, simply rename the 'Math' file in the ProWrite drawer as 'Macro_1' (this can be done from Workbench if you haven't deleted the macro's '.info' icon file). Now, pressing the F1 key while holding down SHIFT will invoke the math macro.

How *Math* works

Adding mathematical calculation abilities to a program would normally be a fairly major job for the programmer, and might not be worth the effort if the feature wasn't demanded by a large number of users. By using ARexx macros, *ProWrite* users can take advantage of ARexx's mathematical prowess even though *ProWrite* itself doesn't do math. If you look at the *Math* script, you'll find a surprisingly short program:

```
/* Evaluate math expression and paste */
Address 'ProWrite'
Options Results
Extract
If RC ~= 0 Then Exit RC
result=Insert('a=',result)
Interpret result
CursorRight
Type '='a
```

This script simply extracts the selected text using ProWrite's EXTRACT command, then evaluates it as an expression using ARexx's powerful INTERPRET instruction. The result of the calculation, stored in the A variable, is then re-inserted into the document with ProWrite's TYPE command.

Italicize a word

Here's another example: this macro will italicize a word if the cursor is positioned within the word or at the end of the word; if the cursor is at the beginning of the word, the previous word will be italicized. Save this macro as 'Macro_2' in the same directory as the *ProWrite* program, and you will then be able to change a word in your document by simply clicking on it and pressing Shift-F2.

```
/* ProWrite: Italicize word */

/* select the word */
'CtrlUp'
'ShiftDown'
'CursorLeft'
'CtrlDown'
'CursorRight'

'StyleItalic' /* italicize */

/* move cursor past word */
'CtrlUp'
'ShiftUp'
'CursorRight'
```

You will notice the use of *ProWrite*'s key commands for holding down the SHIFT and CTRL keys while moving the cursor. This lets us select the word without using the mouse. You could easily adapt the above script to perform other operations on the selected word by changing 'StyleItalic' to 'StyleBold', 'StyleUnderline', 'ColorYellow', etc. All of these commands are listed in the *ProWrite* manual under "Macros and ARexx" in the reference section.

Note in the above script that we did not include the ADDRESS "ProWrite" instruction, since this macro will always be invoked from within *ProWrite*, which provides the appropriate default host address. This may not be true for older versions of *ProWrite*. If the script does not work, try adding the ADDRESS "ProWrite" instruction to the macro, right after the comment in the first line.

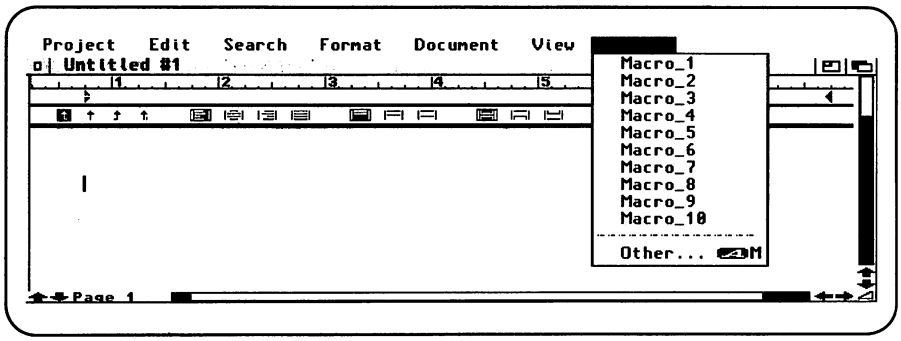


Figure 13-1:ProWrite's Macro menu

Electric Thesaurus
Softwood, Inc.

Host address:	"EThes_1" for first window, then "EThes_2", etc.
Added commands:	17
Macro support:	Run macros from menus (macros are of limited use)
ARexx applications:	As a host for external programs requiring Thesaurus capabilities

Electric Thesaurus (ET) supports a small set of ARexx commands, just enough to look up a word, get information about it, and pop ET's screen to the front or back. As for macro capability, ET supports running named macros (stored as script files) from a menu, but this capability has little application due to the limited ARexx support. It is much better to use ET's ARexx port by running scripts from the CLI using the RX program, or—better still—using ET from another ARexx-compatible program. This lets you add an on-line thesaurus to any text-oriented program that doesn't have one: text editors, terminal programs, word processors, etc.

Using *ET* from *ProWrite*

As an example, take a look at this short *ProWrite* macro that looks up the currently selected word and pops ET's display to the front to show the definitions and synonyms for the word. For this to work, both ET and *ProWrite* must be running, and this script must be saved in your *ProWrite* directory. From *ProWrite*, double-click the word in your document to look up, then run this macro:

```
/* ProWrite-->Electric Thesaurus */
options results
'Extract'
address 'ETHES_1' 'LOOKUPND' result
if RC=0 then address 'ETHES_1' 'SCRTOFRONT'
```

Once you've found the definition you're looking for, you'll have to switch back to the *ProWrite* screen yourself (use the screen gadgets or left-Amiga M).

A more flexible way to use ET from ARexx is by getting the information for the word you're looking up directly. There are several commands that will put information into the ARexx RESULT variable for the script to use. You can then display information with the ARexx script, instead of just showing the ET display.

Using *ET* in a script

Here's a simple script that you can run from a CLI or Shell window, and see all the definitions and synonyms for a given word right on your console window. It adds the refinement of running the ET program if it's not already running (provided the program "ET" is somewhere in your command path).

```
/* EThes.rexx: look up word in Electric Thesaurus */
parse arg word .
options results

/* check for valid argument    */
if word='' then do
   say "Usage: rx thes <word>"
   exit
   end

/* check for ET's ARexx port and run the program
   if it's not there.
*/
if ~show('p', 'ETHES_1') then do
   address command 'run h:apps/et_drawer/ET WB'
   address command 'WaitForPort ETHES_1'
   end

address 'ETHES_1' /* host address of first window */

'LOOKUP' word

if RC=0 then
   say result
else
   say "Couldn't find word!"
```

Save the above script as "rexx:EThes.rexx", and pass it the word to look up when you run it with RX. For example, to see all of the different definitions and synonyms for the word "amazing," you could use the following command:

```
Shell> rx thes amazing
```

In the above script, note the use of the *WaitForPort* program. This is used after running the ET program, and halts the script until the program is in place and its port has been opened.

Other ways to look up words

Rather than obtain all the information at once using LOOKUP, you can get information for one definition at a time using the following commands:

GETNUMDEFS	Get the number of definitions for a word
GETDEF <n>	Get the definition for definition #n
GETPOS <n>	Get the Part-of-Speech (Noun, etc.) for definition #n
GETSYN <n>	Get the synonyms for definition #n *

(*In version 2.05 of ET, GETSYN does not work properly. Use LOOKUP and extract the information using ARexx string functions.)

Telecommunications

In many ways, a terminal program is an excellent example of how much ARexx support can add to the functionality of a program. Both terminal programs discussed here provide complete control of incoming and outgoing text through ARexx. In everyday use, this lets you write simple scripts to automate routine operations like logging on, reading mail, downloading files, etc. On a larger scale, control of a terminal program through ARexx means you can write a complete BBS (Bulletin-Board system) in ARexx, customizing it to your exact requirements. Even the simplest 'personal BBS' will let you dial up your Amiga from a remote location and download files.

The terminal programs discussed here, *A-Talk III* and *Baud Bandit*, use a different set of commands, but in most respects implement ARexx in similar ways. Both programs have their own 'script language' that can be used without ARexx, but most of the commands in the language can be used as ARexx commands as well. This gives your ARexx scripts access to the program's capabilities, as well as the ARexx language's features (which are superior to those of the program's built-in script language).

A-Talk III
Oxxi

Host address:	"ATK"
Added commands:	47
Macro support:	Runs scripts selected from file requester
ARexx applications:	ARexx scripts can provide automated control of logging on to online services, downloading, logging on and downloading messages at off-hours, access to AmigaDOS from a remote location, etc.

A-Talk III (ATalk) has a powerful script language that doesn't require ARexx, but its commands can be used from ARexx scripts. ARexx scripts can be run by selecting a menu option and choosing the script from a file requester, or from an ATalk script using the special 'RX' command (not to be confused with the RX program). Since script files can be run automatically when a number is selected from A-Talk's 'phone book', ARexx scripts can be launched automatically as well.

A lot of operations can be performed using just a few powerful commands. For example: dialing a remote host computer, waiting for a prompt from the remote host, and giving a reply can be accomplished with this simple script:

```
/* A-Talk III - dial host and give password */
address 'ATK'
'REPLY "ATDTxxxxxxx"' /* dial host number */
'WAIT "Password:"'
'REPLY "password"'    /* give password */
```

The WAIT command waits until the given string arrives from the modem, and the REPLY command sends the given string out to the modem. These simple commands alone, coupled with the power of ARexx, are often enough to create useful telecommunications scripts.

Note the use of single quotes around the entire commands, and double-quotes to delimit the strings inside the commands. This is necessary with A-Talk, since it expects to see the double-quotes around the strings.

As with most ARexx-compatible programs, A-Talk does not require the explicit setting of the host address as in the above example, if the script is executed from within the A-Talk program. You may wish to

put it in, however, in case you want to run the script from a CLI window using the ARexx RX program.

Examples

The ATalk manual lists some simple scripts, but the installation disk contains some real ARexx treasures. A complete small-scale BBS (Bulletin-Board System) is included, in the form of an easy-to-read, well formatted and documented ARexx script. This is an excellent starting point for creating your own custom BBS. Another script, called "A-Talk-Remote", lets you call up your Amiga from a remote terminal and access it using standard AmigaDOS CLI commands. There are also a few scripts for more specific applications.

Baud Bandit
Progressive Peripherals and Software

Host Address:	"BAUD"
Added commands:	30 regular commands plus 43 embedded text codes
Macro support:	Runs scripts selected from file requester, or by special 'backslash' codes in character strings
ARexx applications:	ARexx scripts can provide automated control of logging in to online services, downloading, logging on and downloading messages at off-hours, access to AmigaDOS from a remote location, etc.

Baud Bandit takes a slightly different approach to ARexx support than *A-Talk III*. The relatively small number of its ARexx commands is deceptive, since most of the program's features can be accessed through special 'backslash' commands that can be embedded in any text string. Also, the single command STATUS allows 18 different options, compacting diverse functions into one command. Like ATalk, Baud Bandit has its own simple script language that can be used without ARexx. The program is flexible and complete enough in its ARexx support that an entire BBS has been written for it in ARexx.

Running ARexx macros

ARexx scripts can be run by selecting "Start ARexx" from the HELP window, or pressing right Amiga-A. A file requester then lets you select the ARexx script to be run. Scripts invoked from within the program don't need to explicitly address the host, but (as usual) if they do—with ADDRESS "BAUD"—they can be called externally via RX.

A more seamless way of using ARexx macros in your telecommunication sessions is to embed the special "\m" character sequence into one of your 'script pairs'. A script pair in Baud Bandit is similar to WAIT and REPLY in *ATalk III*: they specify what response to give for a stream of characters received from the modem (i.e. from the remote host). One or more script pairs may be assigned to each entry in your phone book, and will be used when calling that number. An example of a simple script pair might be:

```
{Password:=Aardvark\r}
```

This tells Baud Bandit to wait for the string "Password:", and reply with "Aardvark" plus a carriage return when it arrives. The "\r" is just one of many special codes allowed, and the one of concern to us here is "\m" to run an ARexx macro. By specifying "\mrexx:Login.rexx" as the reply string in the script pair, you could execute your 'Login' macro when the "Password:" prompt arrives.

Graphics

Art Department Professional
ASDG Incorporated

Host address:	"ADPro"
Added commands:	55 (plus Operator, Saver and Loader commands)
Macro support:	Runs up to 50 specially-named scripts via function keys
ARexx applications:	Convert graphic file formats and process images from another ARexx application; internal macros to automate graphics processing

Art Department Professional (ADPro) is a graphic conversion-processing system that is designed to function as the 'hub' through which other graphics programs can work. Because ADPro is specifically designed to work with other programs as much as possible, it implements an extensive ARexx interface to control every aspect of the program. In addition, ARexx commands are provided that go beyond the program's regular user interface. For example, user input of numbers, text or file names (using a file requester) can be done from an ARexx script.

Macro support Support for internal macros is by function key, running specially-named scripts that must be in the 'REXX:' directory. The macros called "F1.ADPRO" through "F0.ADPRO" are invoked by pressing F1 through F10 respectively. Similar names using the prefixes SF, LF, AF, and CF work with the function keys in combination with the SHIFT, ALT, AMIGA, or CTRL keys, respectively. In addition, a special F10 macro is provided ("F0.ADPRO") that lets you execute any ARexx script by file name.

Idiosyncrasies The host address must be explicitly addressed (ADDRESS "ADPro") even for scripts executed as macros from within the program. ADPro uses the RC variable in the usual way—to return error results from commands—but other results are returned in the special ADPRO_RESULT variable instead of the usual RESULT. OPTIONS RESULTS must be specified in order to get the results in this variable.

Extensible commands ADPro is somewhat different from most software in that it is designed to continually expand as new graphics formats and display hardware emerges. It supports add-on files known as 'Loaders' and 'Savers' for reading and writing graphics files in any format, as well as 'Operators' for processing the graphics in memory. These add-on files are actually separate programs that function within ADPro itself. Because of this, the use of many ARexx commands depends on the specific Loader, Saver or Operator being used. For example, the "FC24" Saver, which displays its output to a special graphics board, has 16 ARexx commands of its own. As you add new Loaders, Savers and Operators to your basic ADPro system, you'll have to look up the new ARexx commands in the documentation provided.

ADPro Example Here is a simple example of an ADPro macro to scale the current picture in memory to half size, recompute the new picture ('Execute'), and then display the resulting graphic. This script could be saved as "REXX:F1.ADPRO" and run from ADPro by pressing F1.

```
/* ADPro: scale to half size and redisplay */
address "ADPro"
options results

'PCT_SCALE' 50 50

if RC~=0 then do
   'OKAY1' 'Scaling operation failed!'
   exit
   end

'EXECUTE'
'ADPRO_DISPLAY'
```

Note the use of the RC variable to check whether the scaling operation succeeded or not—this and other commands in ADpro can fail due to lack of memory. The OKAY1 command puts up a small requester with the given message, ideal for reporting errors in this way. EXECUTE recomputes the new graphic after the scaling operation, and ADPRO_DISPLAY shows the picture on the screen.

Digi-Paint 3
NewTek

Host address:	"DigiPaint" (or name used to run program)
Added commands:	161 (Mostly direct user-interface equivalents)
Macro support:	None
ARexx applications:	Image processing from an external host; program-controlled drawing; program-controlled titling

Digi-Paint 3 (DigiPaint) has a rather unusual ARexx interface: all of the commands are oriented to operating the user interface controls, but there is no support for executing macros from within the program! This doesn't prevent you from defining scripts to perform common operations within the program, but it does make using them less convenient. You'll have to switch to the Workbench screen and type an 'RX' command into a CLI window every time you wish to execute a script.

Commands

All ARexx commands are four-character codes (referred to as "action codes" in the DigiPaint manual) that are case-sensitive, with an upper case character followed by three in lower case. The case-sensitivity forces the use of quotes around the commands in your scripts, a practice which is generally recommended anyway. The codes can also be used without ARexx at all, using the "Hey" program provided on the DigiPaint disk. A demonstration on the DigiPaint disk provided is in the form of an AmigaDOS script, and consists of a call to the *Hey* program for each command. Converting 'Hey' scripts to ARexx scripts involves putting quotes around the commands, removing excess text, and changing the format of comments.

Interface discussion

The ARexx interface is quite straightforward, with each of the commands directly mimicking a control that could be operated manually. Changes to the drawing are done by controlling the mouse using the **Move**, **Pend** (Pen Down), and **Penu** (Pen up) commands. You can think of DigiPaint ARexx scripts as automatic hands, pushing the right buttons and moving and clicking the mouse to achieve the desired effects. This can simplify thinking through your programs, since you just have to translate your manual operations to commands. On the other hand, more operations are often involved than would be the case with a more conventional command interface; loading a picture file, for example, is a five-step procedure.

Host address

Another slight idiosyncrasy is the host address of the program, which is not fixed, but is always the same as the name used to run the program. This can cause a problem if you've copied the DigiPaint program with the CLI *Copy* command, since file names are not case-sensitive, but host addresses are. A more probable cause of trouble is running DigiPaint from the CLI by typing its name all in lower case, then trying to access it as 'DigiPaint' when the host address is really 'digipaint'. The actual host address is shown in the bottom left corner of the DigiPaint menu area. It is normally 'DigiPaint', the name of the program on the release disk.

One way to avoid having to worry about this problem is to use the following lines to replace the usual ADDRESS "DigiPaint" instruction:

```
dgp = find(upper(show('p')),'DIGIPAINT')
address value subword(show('p'), dgp, 1)
```

Using ARexx and Digi-Paint 3

Since there is no convenient way to execute macros directly from the DigiPaint user interface, you will probably not create many short ARexx macros as shortcuts for common operations. Probably the best use for the rather extensive ARexx interface in the program (other than demonstrations to show off DigiPaint itself!) is using DigiPaint as an external image processor. This usage, similar to the way ADPro can be used (see previous section), is ideal when using an ARexx macro-supporting graphics program that lacks some of the image processing capabilities that DigiPaint has.

Example: picture blending

As an example, the following script uses DigiPaint to blend two pictures together, then saves the result as a third picture file. It can be used as a stand-alone program with the following CLI command:

```
Shell> rx DGPblend File1 File2 Outfile
```

'File1' and 'File2' must be the file names of two HAM-mode IFF picture files, and 'Outfile' will be used as the name of the resulting

picture. This picture will appear as a sort of 'multiple exposure' of the two input pictures, overlaying one on the other as if it were a translucent overlay. (The palette is taken from the first picture.) The script could easily be called from another graphics program that lacked blending capabilities, and added as a macro called 'blend'. DigiPaint would have to be running for it to work, of course, and the computer will need enough memory available to support this. The script could be made to run DigiPaint if it was not already up at the time, in the same way "EThes.rexx" runs the *ET* program (see *Electric Thesaurus*).

```
/* DGPBlend.rexx: blend two HAM pictures and save */

parse arg fname1 fname2 outfile .
address "DigiPaint"

say "Blending pictures with Digi-Paint 3..."

'Aoff' /* all modes off */
/* 'Frbx' */ /* show DigiPaint screen */

/* load first picture
*/
say 'Loading' fname1
'Fnam'fname1 /* load name */
'Load' /* load requester */
'Okls' /* okay to load */
'Pfil' /* palette from file */
'Oklo' /* continue load */

/* load second picture as brush
*/
say 'Loading' fname2
'Fnam'fname2 /* load name */
'Lobr' /* load brush */
'Okls' /* okay to load */

say 'Blending pictures...'

/* set blend controls
*/
'Hvof' /* 2-way blend off */
'Midc' /* center blend, middle (50%) position */
'Mide' /* edge blend, middle (50%) position */

/* stamp down brush on page
*/
'Pend' 160 100 /* Pen down at center */
'Dotb'   /* back to single-dot brush */

/* save resulting picture
*/
say 'Saving' outfile
'Fnam'outfile /* picture name */
'Save'        /* save requester */
'Okls'        /* okay to save */
```

```
'Babx' /* push DigiPaint screen to back */

say 'Completed. Display "'outfile'" to see result.'
```

Test the above script using the images provided on the DigiPaint disk ("PaintBench"). With the DigiPaint program running (push its screen to the back), type the following command at a CLI/Shell window (on one line):

```
Shell> rx DGPblend PaintBench:Images/Fashion
PaintBench:Images/Lady ram:blend.pic
```

You should be forewarned that this procedure will require a lot of memory to complete, and may not work on systems with less than one megabyte. You can reduce memory requirements somewhat by saving the final image on a device other than 'RAM:'.

The script will report its progress to the CLI, and create the blended picture for you (saved as "ram:blend.pic") without showing the process in action. If you wish to see the DigiPaint screen as it happens, change the script by removing the comment characters around the 'Frbx' command in line 9.

To use this script as a macro from another program, you'll have to be able to supply the filename arguments; not all programs allow this. Another approach would be to use fixed file names, and always save and use those specific names for the conversion process.

Other applications

DigiPaint's advanced graphics capabilities coupled with its text features make it an excellent tool for creating title screens for presentations or output to video. Creating these screens from externally-loaded text using ARexx scripts could allow you to make many more screens than would be practical by hand. This is an excellent approach for displaying credits, information, interactive storybooks, etc.

IntroCAD Plus
Progressive Peripherals and Software

Host address:	"ICAD"
Added commands:	50 menu commands; 35 option settings; mouse control
Macro support:	Command console; access through native scripts; execute native scripts by cursor and function keys or by name.
ARexx applications:	Program-generated drawings; numerical interface to drawing creation; some use as key-driven macros.

IntroCAD Plus (IntroCAD) uses ARexx as a fundamental component of the package, and integrates macros as a kind of command language that can be typed at a special console window. This provides a powerful programmable, numeric command-driven interface that can work in parallel with the mouse and menu graphical user interface. In addition, IntroCAD supports its own scripts that do not require ARexx; these can call ARexx scripts, however, adding another level of ARexx integration. Since separate scripts are invoked every time you press a cursor key or function key, you can customize each of these actions with your own ARexx macros.

Added commands

The ARexx commands in IntroCAD work slightly differently than in most other programs. Most features of the program are accessed by simply specifying a menu selection, using a somewhat unusual command syntax. For example, to place text in your drawing, you would begin by selecting the *Text* item in the *Draw* menu as follows:

```
address 'ICAD'
'/draw/text'
```

Text is entered (when in text mode) using the '$' command, like this:

```
string = 'Text to be entered in drawing'
'$'string
```

Putting the command in quotes ensures that ARexx does not try to interpret it, but simply passes it on to the ICAD host address. (The host address must be specified, even for macros executed from within IntroCAD.)

Since many features can be accessed using menu commands, there are not a lot of new commands to learn; simply select the same menu items you would if you were using the program manually.

To move the mouse and click the left mouse button at a specific point on the screen, you simply use a pair of coordinates followed by an exclamation-point character, like this:

```
1.0 0.5'!'
```

Another category of commands are the 'set' commands used to control a number of options. These begin with an asterisk, but use more conventional keywords to choose the settings. There are about 35 of these commands, some of them having a number of sub-options, also specified by keywords.

Calling ARexx scripts

ARexx scripts are expected to be in the REXX: directory, and end with the prefix '.irx'. Some conventions regarding comments and error handling are specified in the IntroCAD manual; if your scripts adhere to these conventions, they will work better as commands tailored for the IntroCAD environment.

IntroCAD provides a *command console* that is similar to a CLI or Shell in AmigaDOS, but provides direct access to IntroCAD's command language. Anything you type at the console is interpreted as a script-style command, and typing the name of an ARexx script causes that script to be executed. For example, typing the following into the IntroCAD console:

```
box 1 2 3 4
```

Will execute the ARexx script 'rexx:box.irx' and pass it the parameters given. This script simply draws a box of a given size and position, but many more complex scripts are provided. Accessing your own custom-written scripts is done in the same way.

Key-driven macros

Every cursor key and function key, along with the ALT, SHIFT, and CTRL key qualifiers, are bound to an IntroCAD script file. IntroCAD scripts do not themselves use ARexx commands, but can invoke ARexx scripts by name. IntroCAD scripts are in a separate directory from the ARexx scripts, and end in the suffix '.is'. They are bound to the appropriate key by their names, e.g. "F1.is", "Shift-up_arrow.is", etc. Invoking an ARexx macro with a special key involves creating the ARexx script (e.g. 'rexx:script.irx'), and calling that script from the IntroCAD macro (e.g. 'is_dir/F1.is'). The directory that IntroCAD searches for its script files can be set by the '*is_dir' command, from the console or within a script.

Interface discussion

Because of the way ARexx and IntroCAD script commands are applied to the drawing, ARexx macros are of limited value for editing—as opposed to creating—a drawing. For example, there is no way to program something like "double the size of selected object", since there is no such thing as a "selected object"; moreover, such operations are normally done via the edit menu, which cannot be accessed from script commands.

The real benefit of using scripts—and it is a significant one—is in program-generated drawings. Complicated shapes can often be created with relatively simple scripts, but could not be easily or accurately done by hand. The IntroCAD disk provides examples of such ARexx scripts, such as a 'spirograph' toy simulation. In real applications, complex grid networks or repetitive patterns would be ideal candidates for scripts.

Another good use of scripts is numerical creation and placement of objects using commands typed at the console window. The simple 'box' example shown above illustrates this: if you need to create a box at precisely defined numeric coordinates (like 4 3/16 in.), you can simply type the value '4+3/16' as an argument to the BOX command. Contrast this with the manual method of calculating the value in inches, zooming in tightly, and trying to accurately position the mouse by hand while watching the coordinate display.

Things to watch out for

IntroCAD's ARexx support seems geared to 'power users', and not all of its intricacies are immediately apparent. Here are a few pointers:

- The console window (select *Script/Console* from the *Project* menu) uses the *ConMan* console handler by ARexx author William Hawes. Unless you have ConMan installed, the console window will not open. (ConMan is provided on the IntroCAD disk.)
- You'll have to copy the '.irx' scripts from IntroCAD's 'rexx' directory to your system REXX: directory, or reassign REXX: to IntroCAD's 'rexx' directory. You can then access these by typing their name at the console (type 'help <scriptname>' for usage information).
- Many of the provided scripts assume that the ARexx libraries 'rexxmathlib' and 'rexxsupport' are available. You can provide these by using the RXLIB program:

```
Shell> RXLIB rexxmathlib.library 0 -30
Shell> RXLIB rexxsupport.library 0 -30
```

- Use quotes around the IntroCAD commands in ARexx scripts, but not in the '.is' scripts.

Multimedia/Hypermedia

While the first three products in this category are concerned with the interactive display of a variety of different formats of visual data, the similarity ends there. There may be a small amount of overlap for some applications, but each program has its specialty, and takes advantage of ARexx in a different way. Just to clarify the differences: *CanDo* (Inovatronics) is ideal for creating applications yourself, without learning a lot of programming. *HyperBook* (Gold Disk) is optimized for direct use, keeping track of notes, pictures and everyday information. And Commodore's *AmigaVision* is well-suited to producing the kind of interactive multimedia presentations you might find at an information kiosk.

Before continuing, a final prefatory remark is in order. *HyperBook* was designed and created by the authors of this book. Despite this, we hope that you will find no evidence of bias in the discussion that follows. On the other hand, it should come as no surprise if the ARexx support in *HyperBook* seems extensive, given the authors' preoccupation with the language.

CanDo

Inovatronics, Inc.	
Host address:	User-definable
Added commands:	User-definable
Macro support:	None
ARexx applications:	ARexx support in CanDo-based applications; user interfaces to control other ARexx command hosts

CanDo is an applications generator, and functions as a graphical interface to a programming language. You are essentially writing a program—in an easy, graphical way—when you use CanDo. CanDo uses its own script language, and ARexx scripts cannot be used instead. CanDo doesn't support ARexx as a macro language: it does not run ARexx macros, and it adds no ARexx commands of its own. Instead, CanDo allows you to put your own ARexx support in the applications you create.

Receiving commands

To allow your CanDo application to receive commands, you must first establish the port name (host address) to use. This is done with the 'ListenTo' CanDo script-language command. This is usually done initially in a card's attachment script. Once this has been done, ARexx commands are added to your application by creating ARexx objects. Each of these objects specifies the command they will respond to, and the script to be executed when that command arrives at the current 'ListenTo' port. The script for an ARexx object can perform any action required, and can find any parameters passed along with the command through the special system variable 'TheMessage'. By adding as many ARexx objects as are required, you can create a full-featured ARexx host as a CanDo application. The host's commands could be sent from RX at a CLI, or from an ARexx macro being executed by any other program.

CanDo automatically replies to messages with a return code of 0, indicating no error. CanDo version 1.5 adds a new command, 'ReplyARexxWith'. This lets you specify the error code and/or the result to return for any given command.

Sending commands

When you use any ARexx command provided by a host program, you are sending a message to the host's ARexx port. Before sending messages from a CanDo script, the port name must be established with the 'SpeakTo' CanDo command. This is analogous to using ADDRESS in an ARexx script. After that, commands are sent to that host address with the 'SendMessage' command. You can find the error return code (what would be RC in an ARexx script) using the system variable 'MessageErrorCode'. You can find the result from the command (usually RESULT in an ARexx script) from the variable 'MessageReturned'.

Version 1.5 of CanDo adds the capability to send a message without waiting for results, using the ASYNC option to SendMessage. Another option, NORESULTS, will send the message without requesting results from the host (like running an ARexx script with no OPTIONS RESULTS instruction).

Summary

CanDo commands for ARexx support:

ListenTo	Establish port (host address) for receiving commands
SendMessage	Send command to host address
SpeakTo	Establish port for sending commands (like ARexx ADDRESS)

Commands are handled by ARexx objects, and optionally replied using the ReplyARexx command.

HyperBook

Gold Disk	
Host address:	"HB_REXX" (not used in most macros; works as function host)
Added commands:	137 (implemented as added ARexx functions)
Macro support:	Built-in storage and editing of macros; macros executed by selecting from list, by object 'actions', or by function keys
ARexx applications:	Macros for defining common operations; creating applications; user interface for controlling other hosts

HyperBook is a hypermedia program for organizing graphical and other information in everyday use. It uses ARexx as its internal script language, and with ARexx macros HyperBook can be used as a simple applications generator. ARexx can also be used to help create very large or complex documents (hyperbooks) that would be time-consuming to produce otherwise.

HyperBook's ARexx interface is different from most others in one important respect: Most ARexx-supporting software functions as a *command host*, adding commands to ARexx when its port is ADDRESSed. HyperBook is instead a *function host*, adding functions to the ARexx language that are available to any ARexx script while the HyperBook program is operating. Rather than give a command and then use the RESULT variable in an expression, a function call can be used in an expression directly. The functions provided allow the manipulation of objects, control over most program facilities, and additional user input and output mechanisms that are not available from the regular user interface.

HyperBook's special functions and the way they work together form a language called 'HML' (Hyperbook Macro Language). Programming in HML is largely programming in ARexx, but there are quite a number of functions to learn and a few new concepts to grasp. One of these is 'ObjNums', the codes by which objects are uniquely identified.

HyperBook macros do not need to be stored as separate scripts, but are created with a built-in text editor and accessed from a list by name. The macros are stored in memory, and are saved along with the project they apply to. Macros can be run as 'actions' by clicking on objects, or by selecting a macro directly from the list. There are also optional specially-named macros (starting with 'F1_', 'F2_', etc.) that can be run

by pressing right-Amiga and a function key. Any macro can be called as a function (with arguments) by other macros. Single ARexx commands—which may invoke other macros as function calls—may be typed into a special text gadget, or may be executed as the result of an object's action.

Macros as design tools

HyperBook macros can be useful as an aid to the creation and layout of pages. Using the HyperBook functions to position and size objects, macros can accomplish many layout tasks numerically that might be tedious or inaccurate to do by moving and clicking with the mouse.

An example of such a utility is 'SetObjSize' in the Macros directory. It lets you match the size of one object on the page to the size of any other object, by clicking on each of the two objects in turn. When executed, the macro prompts the user with messages displayed in the title bar.

```
/* SetObjSize - Set size of one object to match size
   of another. The ☞ character indicates that a
   line should be entered as a single line
*/
row = getclickrow('Click on the object whose size you ☞
            want to change')
col = getcolumn()
ob1 = getobjectat(col, row)
if length(ob1) > 0 then do
   row = getclickrow('Click on the object whose ☞
               size you want to match')
   col = getcolumn()
   ob2 = getobjectat(col, row)
   if length(ob2) > 0 then
      call scaletosize(ob1, getwidth(ob2),☞
                  getheight(ob2))
   end
```

You can see the use of HyperBook functions 'getclickrow', 'getcolumn', 'getobjectat', 'scaletosize', 'getwidth', and 'getheight'. The call to 'scaletosize' near the end of the macro shows how the result of a function may be used as part of an expression passed to another function.

Other utility-type macros are designed to be used as functions that are called from other macros and passed parameters. Some of these, such as 'FileToList' and 'SaveListText', provide functions that are not available through other means. In this way, ARexx macros can be used to add custom features that HyperBook lacks.

Macros are also good for repetitive tasks like applying a change to a large number of objects or pages. If you just created a 200-page hyperbook, for example, and decide that you wish to change the

background color of every page, it would be quite a job to change each one manually. On the other hand, a macro to do the job would be as simple as this:

```
/* change background of all pages to blue (pen 3) */
do page=1 to numpages()
   call setbackground(page':', 3)
   end
```

Macros as 'smarts'

An entirely different use for ARexx macros in HyperBook is as a way to program your hyperbooks to be 'smarter' than simple object actions alone would allow. Since clicking on an object can execute any macro, objects can be made to perform any complex action required. A few applications, like an appointment calendar, daily diary, address book, and a calculator have been created in HyperBook using macros in this way. HyperBook is not an applications generator like *CanDo*, but with the use of ARexx scripts, it can be used to implement some applications.

Learning HML

The HyperBook Macro Language is quite extensive, and the best way to learn how to use it is by seeing examples. The 'ARexx.hb' hyperbook included on the HyperBook Samples disk shows an example of a short script for each function, and lets you run the example script by clicking on the note containing the text. This is done by using the following very short macro as the action of a Note or Button:

```
/* ExecuteText - Run text of initiating note as a
    macro. */
interpret readnotetext(initiator(),0,-1)
```

The ExecuteText macro is useful for experimenting with the language, since you can just modify a program directly on a Note on the page, then simply click on it to see the results.

Things to watch out for

- ARexx is normally used by macros from within HyperBook, not externally using HyperBook as a host program. If you do wish to use HyperBook's functions using RX or from another program, use the prefix 'HML_' before every function name. This is to avoid possible name collisions with non-HyperBook functions.

- ARexx only works in HyperBook when the program is in its full-screen state. In 'tiny' mode, ARexx commands are not processed, and any attempt to use a function in this mode will 'hold off' the caller until HyperBook is made full-size again. (Version 1.0 of HyperBook also has the undesirable behavior of causing this to happen with *any* unknown function. When experimenting with functions externally, make sure that HyperBook is in full-screen state.)

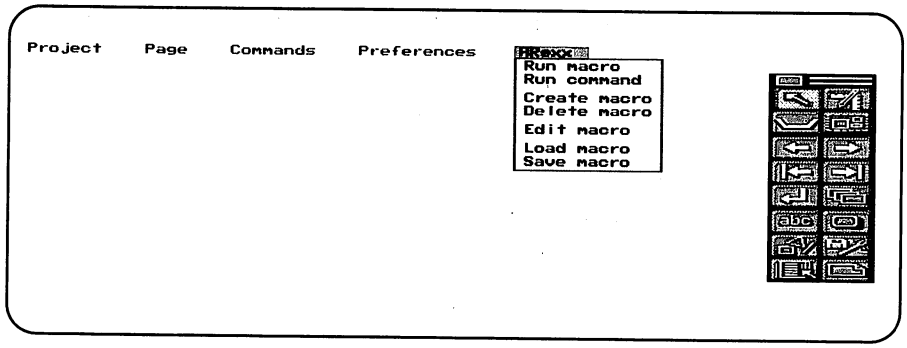


Figure 13-2: HyperBook's ARexx menu

AmigaVision

Commodore	
Host address:	"AV.REXX"
Added commands:	3 (exchange data with AmigaVision variables)
Macro support:	Execute ARexx scripts or commands within a 'flow'
ARexx applications:	Control of external host from an AmigaVision flow

AmigaVision is in some ways similar to *CanDo*, since it lets you create programs (called 'flows') by using a graphical user interface. Like CanDo, AmigaVision does not support ARexx as a macro language to control the user interface, but allows ARexx support to be added to the user's application. While CanDo applications can be made to receive or send commands, AmigaVision only allows you to run macros. For the most part, this means just standard ARexx programming and sending commands to other hosts. ARexx support in AmigaVision is intended as a way to extend the abilities of your presentation by linking with other ARexx host programs.

Adding ARexx scripts

ARexx is handled through the *Execute* icon, which appears as a disk in the main icons menu. When you select 'ARexx Appl.' in an Execute icon's editor window, you can enter the name of a script to be executed, or type an ARexx command directly as a string file (in quotes). The default extension for script names is '.av', but of course any file name for a script may be used. You can also specify the AmigaVision variable names used for error returns from commands (RC in ARexx), and for results from commands (RESULT in ARexx).

Example: device remote control

A good example of using ARexx in AmigaVision is controlling some audio-visual device like a VCR or Laser Disc player through infrared remote control. The ARexx-controlled *IllumiLink* from GeoDesic publications (see the *IllumiLink* entry in this section) will take care of the infrared magic for us, so all we need to do is send ARexx commands to its 'AIR' host program. By creating an Execute icon that sends a command to AIR, you can send remote control commands from an AmigaVision flow (providing the IllumiLink hardware is connected and the AIR program is running). If your current 'AIR window' responds to the PLAY command, for example, you could send this command with the following string file in an AmigaVision Execute window:

```
"address 'AIR' 'output ir<PLAY>'"
```

The surrounding quotes are important to distinguish the command as direct ARexx code and not a call to a script file. This works in exactly the same way as the arguments to the RX program.

Using a *Wait Mouse* icon and objects to define various buttons on the screen, you could easily create an AmigaVision on-screen remote control panel for an IllumiLink-controlled device. Simply use a number of *If-Else* icons in a loop under the Wait Mouse, each checking for a particular response and performing the appropriate Execute icon when it is received. (See picture.)

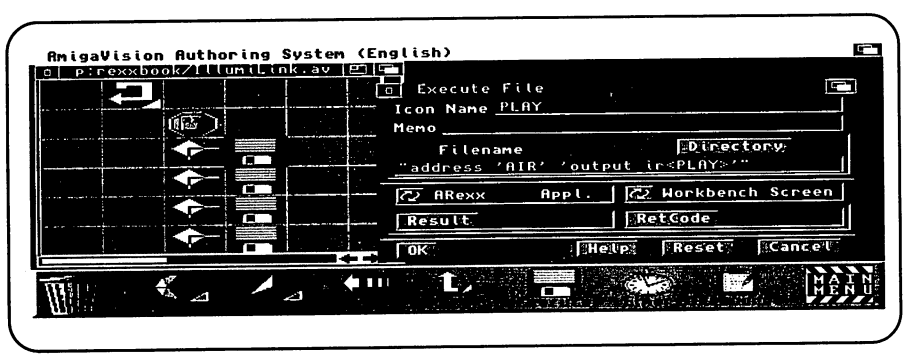


Figure 13-3: AmigaVision Execute window showing an ARexx command to operate a VCR by remote using the IllumiLink 'AIR' program. The flow shown in the left window creates an on-screen remote control and responds to each button press with an Execute icon

Exchanging data If you want to do more with your AmigaVision flow than simply send commands, you can use the SETVAR and GETVAR ARexx commands to modify or read AmigaVision variables from an ARexx script. This gives you direct communication between an ARexx script and an AmigaVision flow. ARexx can read data from an AmigaVision database, look up or process information based on that data, and return results in other variables that the AmigaVision flow can read.

ShowMaker
Gold Disk Inc.

Host Address:	"ShowMakerArexx.port"
Added commands:	7
Macro support:	Execute ARexx script or command at any point in a presentation
ARexx applications:	Starting ShowMaker presentations under external control; Controlling external hosts in synchronization with a presentation

ShowMaker is a presentation program that uses ARexx in two distinct ways: it can receive commands to control the display of a presentation, and it can send commands to other hosts at specific times while a presentation is playing.

Controlling ShowMaker with ARexx

ShowMaker makes use of a small set of commands to load, start, stop and pause a presentation. This is essential when using ShowMaker as a 'player' to add its presentation capabilities to other programs.

Using these commands, a simple ARexx macro could be attached to a 'button' on a *HyperBook* page, for example. Clicking on the button would then display animation and music—capabilities that HyperBook lacks—via ShowMaker. This would simply involve running ShowMaker in the background while HyperBook had its screen up front. The same technique could be used with *AmigaVision* or *CanDo* applications, or with any such program that can send ARexx commands under user control.

Like *Bars & Pipes*, ShowMaker supports a 'preload' command to prepare a presentation for immediate playback at the required time. This lets you load the presentation while the user is reading text or viewing a still graphic, then jump into an animation or song without any perceptible delay.

Sending ARexx commands from ShowMaker

ShowMaker's other form of ARexx support is analogous to that in *AmigaVision*. By using an ARexx 'event type', ShowMaker allows the execution of an ARexx command or script at a precise point in the 'timeline' of a presentation.

The application for this is to further extend ShowMaker's display abilities by linking to another host that can perform any display or sound tasks that may be required. Examples might be playing music through an ARexx-controlled MIDI player like Bars & Pipes, using advanced display hardware such as 24-bit color graphics boards through *Art Department Professional*, or sending a remote control command to a CD player using *IllumiLink* (covered next).

Both uses of ARexx may be linked together to create a multi-level presentation, starting with an interactive program like CanDo, HyperBook or AmigaVision, then calling ShowMaker to display the presentation, which in turn calls other hosts to provide additional audio/visual control.

This is the kind of multitasking control that ARexx proponents envisaged from the outset, and with currently available software applications, it is now a practical reality.

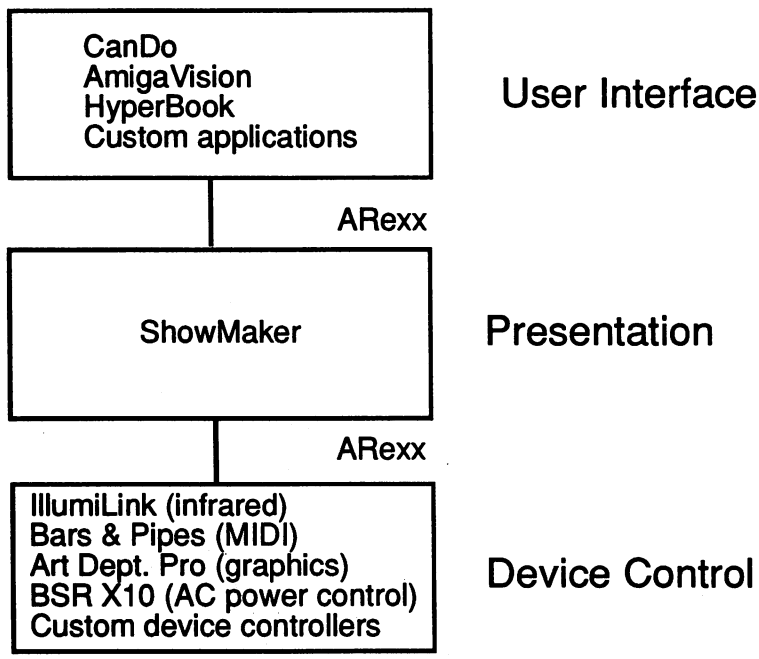


Figure 13-4: Control Hierarchy

IllumiLink
Geodesic Publications

Host Address:	"AIR"
Added commands:	28 plus user-defined commands
Macro support:	Link ARexx commands and scripts to any key or input event
ARexx applications:	Controlling remote-controlled devices via ARexx; computer control from cordless telephone

The *IllumiLink* is a small hardware device that sends and receives infrared signals from a remote control unit. The IllumiLink software provided allows sampling and controlled playback of these signals. Playback may be controlled directly through custom-designed user interfaces or by ARexx commands.

Another user of the IllumiLink is in decoding touchtone signals from a cordless telephone. By plugging the telephone base unit into the IllumiLink, you can control the computer and send ARexx commands by pressing keys on the remote telephone handset.

Infrared remote control

Once you've sampled a remote control's infrared signals with the sampler program and set up an on-screen remote control using the 'AIR' program, you can send ARexx commands to control the device.

All of the setup information is saved in special IFF files, so the device interface can be set up in the future by double-clicking an icon or giving a CLI command. If you are using AIR from ARexx only and don't require on-screen controls, it can shrink and hide its window to get it out of the way.

The names you choose for each of the buttons on the remote—common choices might be 'PLAY', 'STOP', 'PAUSE', etc.—are the names you send along with the ARexx commands to activate the controls.

Any buttons available on the remote unit can be programmed, and any names might be used: 'Back_One_Song' and 'Forward_Scan' are CD player examples suggested in the IllumiLink manual, for example.

The most straightforward way to send infrared remote control commands with ARexx is to send it an OUTPUT command:

```
address AIR 'output ir<PLAY>'
```

This could be sent from any application that allows you to execute ARexx commands or scripts. Obvious examples are hypermedia programs like the ones discussed above.

The OUTPUT command causes immediate output of the specified type of event. The IR keyword is one possibility; others are AREXX_OUT, KEY, and EXEC. The IR keyword accepts the name of a user-defined 'button' on a remote-control window, which in the above example is assumed to be called 'PLAY'.

The IllumiLink provides a lot of flexibility in its ARexx interface that is not required if you just want to use it for simple ARexx control of infrared devices. The Illumilink and AIR program allow input from the keyboard, ARexx, or a cordless telephone base unit and can output infrared signals, send ARexx commands, simulate key presses and mouse movements, or run any program. Inputs are linked to outputs with the LINK command. Links need not be established through ARexx commands, by can be set up in the Workbench 'tool types' for an AIR icon.

Links can also be used to add new ARexx commands to AIR's repertoire. For example, to select a specific CD and start playing a song when the ARexx command 'Play_Misty' is received, you might use this link command:

```
link arexx_in<Play_Misty> ir<CD_Disk 3 @10 Play 6>
```

The above command is an example from the IllumiLink documentation. It sends several pre-defined remote control commands to the AIR program, pausing 10 seconds after selecting CD #3 from a multi-disc changer and before playing track number 7.

Infrared control is supported as output only: the signals are only input when 'training' the program to respond to a particular remote control unit. Since the precise timing required by infrared sampling places heavy demands on the Amiga's processor, the IllumiLink cannot be used to receive infrared commands directly and pass them on to an Amiga application to be acted upon. For this kind of remote control application, a cordless telephone can be used instead.

Cordless telephone input

When you plug the base of a cordless telephone into the IllumiLink, the handset can be used to send signals to the AIR software. These signals can be directly translated to any output type including keystrokes, mouse movements or ARexx commands.

By establishing links from telephone input to ARexx commands, the IllumiLink acts as an input device that can be used to control other software. Figure 13-4 shows the IllumiLink at the bottom of the control hierarchy, performing the final access to a VCR, CD player or other remote-controlled device. Using cordless telephone input could place the IllumiLink in the top box of the diagram or even in one above, controlling a HyperBook application to control ShowMaker and so on down the line.

One or two-digit codes entered by pressing keys on the telephone handset can be mapped to whatever functions you set up with links. Some examples using software discussed in this chapter might be:

- Playing MIDI through Bars & Pipes
- Showing a presentation with ShowMaker
- Turning pages or displaying data in HyperBook
- Controlling a CanDo or AmigaVision application
- Loading and displaying HAM-mode graphics with DigiPaint
- Displaying business graphs and figures using Superplan or Advantage

Every telephone input that you wish to respond to must have a corresponding LINK command. These may be set up as icon tool types, or created dynamically by external ARexx commands.

For example, to tell Bars & Pipes to start playing the current song when the telephone user keys in '23', you could use the following LINK:

```
link phone<23> arexx_out<address 'Bars&Pipes ARexx' 'START'>
```

To turn to page four in the currently displayed hyperbook when the user keys in '04':

```
link phone<04> arexx_out<call HML_NEXTPAGE()>
```

To show a graph of the currently defined *SuperPlan* spreadsheet:

```
link phone<55> arexx_out<address 'SpRexx' '/vs>ENT'>
```

Music

Bars and Pipes Professional (with Multi-Media kit)
The Blue Ribbon Soundworks

Host address:	"Bars&Pipes ARexx"
Added commands:	16 (mainly to control playing of songs)
Macro support:	None
ARexx applications:	Loading and playing of Bars&Pipes songs by other applications

Bars&Pipes Professional (B&P) is a MIDI-based musical composition system. It has an extensive user interface, however B&P's ARexx support is limited primarily to controlling the playback and recording of songs.

Using B&P's ARexx commands from RX or another program, you can load and save songs, start playing the song from any point, stop it, switch to record mode, and push the B&P screen to the front or back. With these commands, you could add MIDI music to an AmigaVision presentation, for example (if your system has enough memory!). Like *DigiPaint 3*, B&P does not support macros—you can't run ARexx scripts from within the program.

B&P's ARexx support is actually provided by an "accessory", a separate module on the program disk that can be opened from within B&P. Once the ARexx accessory has been opened, B&P's ARexx port and added commands are available.

Preloading songs

If you are controlling the playback of more than one song stored on disk from AmigaVision or another presentation program, you won't want to wait in silence while a new song loads from disk. The PRELOAD ARexx command allows you to load a song from disk, even while the current song is playing. When you wish to play the preloaded song, use the INSTALL command, then START to play it as usual.

MIDI playback

Running B&P in the background to use merely as an ARexx-controlled MIDI playback tool could be considered overkill, like running *Deluxe Paint III* simply to display a picture. Few Amigas will have the memory resources to run AmigaVision and B&P at the same time, and B&P's memory requirements may preclude the use of other presentation software as well. If all you need is ARexx-controlled playback of your songs, you can save your songs as standard MIDI files (using the 'MuFFy' accessory), then play them back using the *Bars&Pipes MIDI Player*. This is a small application that uses a subset of B&P's ARexx commands. The B&P MIDI Player is available on the optional Bars&Pipes Multi-Media Kit disk, and is highly recommended for B&P ARexx users.

B&P provides some commands, like RECORD, PUNCHIN, PUNCHOUT, TOFRONT SCREEN and TOBACK SCREEN, that are not concerned with the mere playback of the song. It also supports FRAME to read or set the current frame type, and TEMPO to read the current tempo. These features are not provided in the MIDI Player. However, if you don't need those capabilities—and you can get by with specifying times in SMPTE format only—you can use just the MIDI player. This may be the only way you can play your B&P compositions from other programs, given the amount of memory available in your system. The MIDI player *does* support the PRELOAD command, so you can still play uninterrupted music. The MIDI player's host address is 'B&P MIDI ARexx', and its ARexx commands are documented in the Multi-Media Kit documentation.

Database/Scheduling

SuperBase Professional 4
Precision Software

Relational database	
Host Address:	"SBpro4"
Added commands:	Over 250 (all commands and functions in language)
Macro support:	Call ARexx scripts from within DML programs
ARexx applications:	Linking database fields to external hosts; access to database and DML commands from any ARexx script

SuperBase Professional 4 (SBpro) is a 'high-end' database application and incorporates DML, a Database Management Language that allows complete control of data and the user interface to a database.

ARexx communication can work both ways in SBPro: you can access the data through DML commands in external scripts run via RX or another host, or you can call ARexx scripts from within a DML program. In other words, SBpro can be controlled by other programs or can control other programs through ARexx commands.

External control of SBpro

SBPro's language, DML, provides access to all database functions and is also a complete programming language with most or all of the features found in ARexx. For that reason, ARexx scripts are not required so much as macros to be used from within the program, since most operations can be just as easily programmed using the native DML.

Instead, ARexx can be used from another host program to extract data from the currently opened SBpro database. Since complete access to all DML commands and functions is available to ARexx, opening new files, sorting on new indexes and modifying records are all examples of operations that can be performed.

Using DML commands from ARexx is not as convenient as programming in pure ARexx or pure DML. Giving DML commands is straightforward: they are simply given as host commands in the usual way. The difficulty comes in obtaining results from DML: the contents of fields in a database record, results of expressions and special variables. This is done by specifying OPTIONS RESULTS in ARexx, then simply giving the expression on its own. Before using another DML command, results have to be turned off again.

The following script illustrates the above concepts. It assumes that the 'Clients' database from the SBPro examples disk is currently opened, using its Form file called 'stkc'. The script will read and display the name (first and last) for each client record in the database, along with the 'Net Due' amount for each one.

```
/* SuperBase Professional 4 ARexx example
   List data from currently opened "CLIENTS" database
*/
address 'SBpro4'
'SELECT FIRST'
options results
'RECCOUNT("CLIENTS")'
numrecs=result

do numrecs
   options results
```

```
'Firstname.CLIENTS'
first=result
'Lastname.CLIENTS'
last=result
'STR$(Net_Due.CLIENTS, "-9,999,999.00")'
due=result
say left(first last, 40) due
options no results
'SELECT NEXT'
end
```

As you can see, issuing the DML command 'SELECT FIRST' was done as a simple host command in the usual way, after the 'SBpro' port had been addressed.

The next step—reading the number of records in the current file—was accomplished by setting OPTIONS RESULTS and sending a DML expression as a command. The expression was simply a call to DML's RECCOUNT function. Since results were requested, DML interpreted the host command as an expression rather than a DML command. The result from the expression is returned in the ARexx RESULT variable.

Once in the loop, reading the contents of database fields is a simple matter of referring to the special DML variable names. The call to 'STR$' is a DML string function and is used in the above script to format a numeric value. ARexx functions and variables can be combined with the use of DML functions and variables in this way, creating a 'super-language' consisting of the combination of capabilities of both languages.

Before sending the DML 'SELECT NEXT' command to select the next record in the database, the script must turn off the RESULT option. The ARexx command OPTIONS NO RESULTS turns off the result option without affecting any of the other options. The OPTIONS instruction by itself would also turn off results, but would turn off or reset all other options as well. The NO keyword was added in ARexx version 1.10 to allow individual control of options.

Using ARexx from SBpro

The above example showed you how to use DML in an ARexx script; it is also possible to use ARexx in a DML program.

Using DML's CALL command, you can send an ARexx command to any host address. As an example, if you wanted to dial the telephone number of a selected record in the currently open 'clients' database, you could pass the number to an ARexx-controlled speed dialer (Gramma Software's *FreD*) using a DML command like this:

```
CALL "rexx_fred" EXECUTE "DIAL " + Telephone.CLIENTS
```

The first part of the CALL is equivalent to an ADDRESS instruction in an ARexx script, and the part after the EXECUTE keyword is the actual host command sent to the specified host address. In this case, we send a 'DIAL' command to FreD (the host) using a DML string expression containing the telephone number from the selected record.

By using the RETURN and TO keywords with the CALL command, you can obtain results from a host and put them into any DML variable or database field. This is equivalent to specifying OPTIONS RESULTS in an ARexx script, sending a command to a specified host, then examining the RESULT variable.

As an example, let's access the 'FreD' host again, but this time we'll look up a name to see if it's in the currently loaded telephone list. This might be used in a database application to see if a client's telephone number is already on file in the speed dialer, and add it automatically to the database if it is.

```
CALL "rexx_fred" EXECUTE "CLEAR"
name$=Firstname.CLIENTS + " " + Lastname.CLIENTS
CALL "rexx_fred" EXECUTE "MATCH" name$
CALL "rexx_fred" RETURN "GET" TO fred$
```

This sends a CLEAR and MATCH command to the 'FreD' host in the usual way, then sends a GET command and requests results. Note the alternate CALL syntax required to do this. The result—consisting of the person's name, number and a note separated by linefeed characters—is placed in the DML string variable 'fred$', where the telephone number can be parsed out. If the entry was not found, the string will contain three newlines alone. All of this information is specific to the FreD program, which is discussed later in this chapter.

The above example shows how ARexx can be used as the 'glue' that binds together different applications running at the same time. Even though no ARexx programming was involved at all—the above script is an SBPro DML program executing FreD commands—ARexx was used for its standardized inter-process communications facilities.

Executing ARexx scripts in SBPro

Short ARexx scripts can be performed by a number of separate CALL commands as in the above example. To run a script file, you simply use the CALL command with the 'REXX' host address: recall that a command sent to ARexx's own REXX port is taken as the name of a script to be executed. This script will usually be in the 'REXX:' directory and have a '.rexx' filename extension, but the complete pathname of any file may be specified. For example:

```
CALL "REXX" EXECUTE "myscript"
```

This is equivalent to typing 'rx myscript' in a Shell window, and will run the ARexx script called 'rexx:myscript.rexx' if one exists, according to the usual rules.

FreD Speed-Dialer
Gramma Software

Host Address:	"rexx_????" (based on data file used)
Added commands:	13
Macro support:	Clicking right mouse button over data runs standard script with arguments
ARexx applications:	Accessing telephone database from other applications; Dialing any number from another application; slight support for extending program's features with a macro

FreD is a relatively small utility program that uses ARexx to allow external programs to control its list of names, telephone numbers and short notes. It has a small number of ARexx commands to allow looking up and selecting entries in the list, dialing numbers, and a few other functions.

Host address

FreD's host address is not fixed, but depends on the current data file in use. The host address is always nine characters long and begins with 'rexx_'. The remaining four characters are the first characters of the data file name. Scripts run externally via RX or from another application must be made to work with a specific data file. In the examples given here, we will assume the data file name begins with 'fred' (all in lower case).

If you use several different data files with FreD, it is a good idea to adopt this naming convention to allow universal ARexx access: FreD data files all begin with 'fred_' as in 'fred_personal', 'fred_business', etc. This way, you can always assume 'rexx_fred' as the host address for any scripts you write.

Using another name prefix for a data file can then be used as a sort of security measure, since the scripts using the standard host name won't be able to communicate with FreD when it is using that data file.

Macro feature

Full macro capabilities are not provided, but FreD will execute the script called 'fred.rexx' (in the REXX: directory) whenever the user clicks the right mouse button over an entry in the telephone list. FreD passes information about the selected entry as well as other data to the script as an argument.

Unlike many applications, FreD does not set up the host address automatically when invoking the script. It does, however, pass the current port name as one of the pieces of information in the argument to the script, so an ADDRESS instruction along with the VALUE keyword can be used to set the host address from the data parsed from the argument.

The ARexx script could then go on to send commands to FreD, or—more likely—simply use the name and telephone data to send to another host or use it for some other purpose. The 'fred.rexx' example included on the FreD program disk runs a text editor and loads a file specified by the first name of the clicked-upon entry.

In order to extract the pertinent information passed by FreD into variables, your 'fred.rexx' script should parse the passed argument as follows (on one line):

```
parse arg name '0a'x phone '0a'x note '0a'x port editor notedir
```

After this parse operation, the variables NAME, PHONE and NOTE contain the data from the clicked-upon entry, PORT contains FreD's current host address, and EDITOR and NOTEDIR are user-configured options specifying text editor preferences and a directory to hold files containing additional information pertaining to the current data file.

If the script needed to send commands to FreD, the following ADDRESS instruction could be used:

```
address value port
```

Database locking

Since FreD provides ARexx commands to modify as well as read the telephone list data, it provides a method of 'locking' the data so that it can only be accessed by one ARexx script at a time. When the LOCK command is given, FreD returns a new host address to be used for exclusive access to the data. Before completing, the script must give an UNLOCK command to relinquish control to other scripts that may want access to the data.

Example: list FreD entries

The following script 'listfred.rexx' shows the use of database locking and a few of FreD's commands. It is designed to be used from a Shell window with the RX program, and will display a formatted list of all the entries in the currently loaded FreD data file, one per line.

The script assumes 'fred_rexx' as the host address, using the data file naming convention proposed above.

```
/* list FreD entries */
options results
address 'rexx_fred'
'LOCK'
port=result
address value port

'CLEAR'
'REVERSE' /* select all entries */

'GET'
do while result ~= '0a0a0a'x
   parse var result name '0a'x number '0a'x comment '0a'x
   say left(name,30) left(number,15) left(comment,30)
   'GET'
   end

'CLEAR'
'UNLOCK'
```

Save the script as 'rexx:listfred.rexx'. To use it, FreD must be running either in its full-size or 'shrink' mode and contain some names and phone numbers. From a Shell window, type the command:

```
Shell> rx listfred
```

You should see all the names, numbers and notes displayed on your Shell console window.

Compatible programs

FreD is one of a series of utilities from Gramma Software that support ARexx. Other programs are *Nag Plus*, an appointment calendar-schedule assistant, and *Cal*, a calendar program. Nag supports ARexx in a similar way to FreD, using record locking and automatic running of a script with arguments. Cal contains only four ARexx commands used to load, save and print a calendar. (The fourth is QUIT.)

Since the programs make low demands on memory and can 'shrink' their displays when not in use, they can all be run at the same time and communicate with each other using ARexx commands. For example, clicking on a Nag reminder to call Henry Smith could send FreD commands to look him up and dial the number.

Business/Financial

SuperPlan
Precision Software

SpreadSheet/Time Management/Business Graphics	
Host Address:	"SpRexx"
Added commands:	Complete access to user interface: 65 'slash' commands, 45 auto/macro commands, 60 graph commands, text/expression entry.
Macro support:	None
ARexx applications:	Stand-alone spreadsheet applications written in ARexx; Use SuperPlan as a calculating 'engine' from another application; Complicated spreadsheet manipulations

SuperPlan contains a vast number of features and capabilities, all of which can be controlled through ARexx. As useful as ARexx control in this program would be for programming common macros to help while creating or editing a spreadsheet, the current version of the program (1.06) does not provide a facility to run ARexx macros from within the program. Because of this and the fact that SuperPlan's native macro facilities are so convenient, ARexx scripts are not suitable for quick operations to aid in spreadsheet development.

The lack of macro capabilities in SuperPlan means you'll have to run ARexx scripts using RX or from another application that can run ARexx scripts. Once you get the script going, however, SuperPlan's ARexx support itself is extensive.

There are several ways to control SuperPlan from ARexx:

1) Since the program can be operated completely from the keyboard, the user interface can be largely controlled by simply sending keystrokes to SuperPlan as commands. This includes the entire group of basic 'slash' commands, graph commands (entered with a comma), '+' to enter an expression into a cell, '=' to go to a specific cell, and others.

2) SuperPlan has its own programming language using three-letter commands beginning with a '>'. These commands are used when defining 'autos' (sequences of commands assigned to any control-key) and SuperPlan 'macros', which are entered into cells in the spreadsheet and executed by cell reference. There are over forty of these commands. Along with commands to control the user interface and create a spreadsheet application, there are also keyboard equivalents like cursor movements and the RETURN key.

3) When ARexx is in OPTIONS RESULTS mode, commands sent to SuperPlan are interpreted as spreadsheet cell references, and the contents of the requested cell is returned in the ARexx RESULT variable.

The first two methods above can be combined in a command string. In fact, the '>ENT' command is used at the end of many direct keyboard commands to simulate pressing the RETURN key. For example, to move the cursor to a specified cell, you could use the cell goto command. Within SuperPlan, this can be done by pressing '=' (goto), typing a cell name and pressing RETURN. From ARexx, the following command could be used to go to cell C3:

```
address 'SpRexx' '=C3>ENT'
```

The '>ENT' is a macro-language command that simulates the ENTER or RETURN key. The cursor keys, escape key and others can also be simulated with '>' commands.

When using the key-equivalent commands only, operating SuperPlan from ARexx is simply a matter of 'pressing the right keys,' a simulation of using SuperPlan directly. This means you don't have to learn any new commands to program SuperPlan in ARexx if you can already use it from the keyboard.

Using some of SuperPlan's more advanced 'macro' commands can be a bit more complicated. Some of these, however, are designed to give SuperPlan's macro language basic capabilities like conditional branching, subroutines, and other constructs that would be much more convenient if programmed in ARexx. For this reason, there are several macro commands like '>IFT' and '>SUB' that you would probably never use in an ARexx script.

Custom spreadsheet applications

Other SuperPlan macro commands are designed for creating a custom spreadsheet application, and let you redefine menus and key operations, inhibit access to certain parts of the spreadsheet, and completely take control over the entire user interface.

By running SuperPlan from an ARexx macro (using ADDRESS COMMAND) and using these commands, ARexx scripts can be used to write stand-alone programs that use SuperPlan 'beneath the surface,' but present a bulletproof interface that automates standard operations for a specific application.

Home Office Advantage
Gold Disk Inc.

Spreadsheet/Flat-file Database/Business Graphics	
Host Address:	"Advantage"
Added commands:	13 basic commands including access to predefined 'macros' which can access all menu commands and keystrokes.
Macro support:	Execute ARexx script by name; double-click spreadsheet cell to execute attached script
ARexx applications:	Macros to automate spreadsheet editing operations; access spreadsheet functions from another program

Home Office Advantage (Advantage) has less direct ARexx support for its commands than SuperPlan, but it has the macro support that SuperPlan lacks. Not only can macros be run by filename, but a macro can be 'attached' to any cell in the spreadsheet. An attached cell is executed simply by double-clicking on the cell.

The small number of ARexx commands are for reading the contents of the current cell, moving the cursor to a cell, and selecting cell ranges. There is also a command to execute any named 'macro'—these are not ARexx macros, but stored multiple key sequences and menu selections. Since there are no ARexx commands to access most operations, the 'Macro' command is the ARexx interface to complex operations. The macro must be defined in Advantage by performing the sequence of operations manually, then the macro can be executed from an ARexx script. This is not as powerful as having direct ARexx control of all features of the program, but it is a way to work around the lack of ARexx commands in some situations.

Idiosyncrasies

On the subject of ARexx commands, Advantage has a unique syntax for passing arguments. Commands that take arguments don't accept them after the command as usual (which results in the message and the arguments being passed along in the same message), but as *separate commands*. For example, the 'SelectCell' commands takes the name of a cell as an argument. To select cell C1, for example, you would use the following ARexx script:

```
/* select cell A1 in Advantage */
address "Advantage"
"SelectCell"
"A1"
```

The reasons behind this unorthodox method of specifying arguments are unclear, but the syntax can be made less visually confusing by putting the arguments on the same line as the command separated by a semicolon, i.e. "SelectCell"; "A1". (The commands are shown in mixed case for clarity, but they are not case-sensitive.)

Another thing to watch out for is that Advantage will not respond to any ARexx commands until its ARexx port is activated with the "ARexx ON" option in Advantage's preferences requester. Until this has been done, all commands will return with an error code of 100.

Program Development

A number of program development tools are adopting ARexx 'hooks' to provide an integrated programming environment. A common approach is to link a text editor with a compiler or assembler in some way, the goal being to allow editing, compiling, and error reporting and correction to all take place within the text editor environment.

There are several excellent ARexx-supporting text editors, and some integrated development environments with ARexx support. In this section, we look at a representative example of such an environment, C.A.P.E. for assembler programming.

For C programmers, Manx Aztec C 5.0 and SAS/Lattice C 5.10 are covered only briefly—Manx because of its minimal ARexx support, and SAS/C because the materials were received at the very last moment before press time and too late for a detailed analysis. Future editions of this book will no doubt provide coverage of these and many new ARexx-supporting products in detail.

C.A.P.E. 68k Version 2.5
Inovatronics Inc.

Assembler programming environment	
Host address:	"CAPE"
Added commands:	All editor commands (over 60) and keystrokes
Macro support:	Execute macro by name, with arguments
ARexx applications:	Macros to automate editing sequences; macro control of assembly/linking process

C.A.P.E. is an assembly-language programming system featuring an integrated text editor/assembler. All features of the editor can be controlled by keystrokes, menu items or ARexx macros. ARexx macros simply feed keystrokes to the editor, which are then interpreted as if they were typed from the keyboard. Since all input is done in this way (there are no requesters or control panels required to access most features), the editor can be completely controlled by macros. Assembly of the current code in memory is one of these options, so assembly can be controlled through ARexx macros as well.

ARexx macros are executed by specifying the script name with one menu item or key sequence, and specifying arguments to the macro with another. The macro is only invoked when the arguments are specified, so invoking a different macro is a two-step operation.

Since all functions are accessed by keystrokes, many of them involving control characters, CAPE macros are not filled with commands in the usual style, but are filled with hexadecimal character specifiers. To alleviate this confusion, a macro is supplied to store the key sequences as aptly named 'clips' using SETCLIP; the commands can then be invoked with the appropriate calls to GETCLIP. This is still a bit confusing, but better than using key codes alone.

Stand-alone assembler

If you prefer to use your own editor, you can use the CASM assembler, which works from a CLI without bringing up a text editor. CASM has an ARexx 'resident' mode, where it will accept assembly commands sent to its port (also called 'CAPE'). In this way, you can assemble files using CASM directly from within any ARexx-supporting text editor.

When used in its regular 'master' mode, the CASM assembler can also issue its own ARexx commands. There are assembler directives to get values, data or source code from an ARexx script. This could be used to assign various constants from environment variables or user input, import data dynamically at assembly time, attach different code

segments based on decisions made by an ARexx script, etc. Any of this data could also be imported from an external host, allowing for interesting possibilities such as importing graphics data directly from a graphics program host at assembly time.

Other products

Manx Aztec C 5.0

The Aztec C compiler does not provide ARexx support *per se*, but can provides a kind of simple 'hook' with an editor for error correction that could use ARexx scripts as an interface.

Use of the compiler's 'qf' or 'QuikFix' option will send error messages to a file in a special format, then execute a specified command if errors were encountered during compilation. This command is by default used to run the 'Z' editor with a special option to read the compile errors and allow cursor positioning to errors in the C source file.

Instead of running the Z editor, the RX program could be used to execute an ARexx script. The script could control any ARexx-supporting text editor, which could then read the error file and use the information within it to provide direct access to the errors in the source file. No such ARexx scripts are provided with this release of the compiler, but there are probably quite a few developers working on their own integrated systems using this feature right now.

SAS/C Lattice C 5.10

SAS/C features a non-ARexx integrated environment between the LSE text editor and the compiler. You can compile a source file directly from memory, and errors will be displayed directly in the editor. You can then correct errors to the source file, recompile, and continue the cycle without leaving the text editor.

Where ARexx fits in is through the ARexx interface to the text editor. Since the text editor is the master controller of the entire edit, compile, and correct cycle, ARexx control of the editor allows script control of the entire cycle as well. LSE's ARexx support was only added in the latest SAS/C release, and is not documented in the manual. All editor operations can be controlled with ARexx commands, including text entry. LSE also allows ALT-function key sequences to run specially-named ARexx scripts from disk, or can run any script by name.

SECTION IV
REFERENCE

Chapter 14
Reference

Preface

With 30 or so instructions, nearly 90 built-in functions, and 20-odd more functions in the support library ('rexxsupport.library'), the newcomer to ARexx is faced with an apparently unending supply of new names to be learned, syntax to be mastered, concepts to be absorbed. Even old hands may find they do not always remember every function argument or mode name, and every subtlety of behavior. We hope this Reference Section is of value to both groups, and we have tried to make it useful in a number of ways.

The bulk of the section consists of an alphabetic listing of every instruction, built-in function and support function, with a syntax summary and brief description that should be enough to jog your memory if you've met the keyword or function name before. Most of the entries include example code, sometimes quite extensive, in one of several forms:

- Complete scripts, that you can execute on your own system. When space permits and the need arises, the output for these is listed along with the script. Even in these cases, we feel that if you're new to the language it's worth running the script yourself to verify the output (though with some scripts you may get different results on your own system). For your convenience, all the scripts are included on the disk that accompanies the book.

- Sessions with the *Dialog* program listed in Chapter 5. These sessions are interactive: the intention is that you will type in the lines with the '->' prompt, and verify that the output is as shown—*and* that you will explain to yourself the reason why each expression you enter does what it does.

- Code examples, consisting of fragments of ARexx code to demonstrate a particular point or technique. These won't work on their own, but are meant to provide a foundation for applying similar techniques in your own scripts.

- Complete functions, not runnable as scripts in themselves, but capable of being included in your own scripts, and used as-is or perhaps modified to suit your own purposes.

Many of the examples in the section assume the availability of *rexxsupport.library*, and will not execute correctly unless you have added it beforehand, for instance with:

```
Shell> rxlib rexxsupport.library 0 -30 0
```

All the entries have in addition a 'Discussion', sometimes quite lengthy, that gives further syntax details, provides background material on technical points, and in other ways enlarges the scope of the treatment beyond what you might ordinarily expect in a 'Reference Section': where possible, we try to show not merely how, but also *why*, and in what ways, a particular function or instruction is used. In fact, some readers may find the section prosier than they would like. To them, we say: The first time around, skim or browse, and absorb what you need. After that, use the *Reference Guide...*

The Reference Guide

On the next several pages, every entry in the main body of the Reference Section is given in abbreviated form. Discussion and examples are omitted, leaving only the essentials of syntax and a capsule description. Unlike the full entries, these are organized in functional groups rather than alphabetically. If you are looking for a particular string function for a particular need, you are likely to be able to find it quickly under 'Strings—pattern-matching', or 'Strings—formatting', or a similar brief listing in the Guide. If you want to get a quick idea of what facilities are available in ARexx as a whole, a short study of the Guide will give you the big picture. And if you just want to remind yourself of the order of arguments to a function, or the syntax of an instruction, either the Guide entry or the main reference entry will serve.

The Reference Guide Format

A listing of all ARexx instructions, built-in library functions, and support library functions, by functional group. Each keyword or function name in the Guide is preceded by one of the letters I, F or S in square brackets. The meanings of these are:

I Instruction

F Built-in function

S Support library function (requiring 'rexxsupport.library')

Flow of control

[I] **BREAK** : BREAK
Exit from innermost DO-END block or INTERPRET instruction

[I] **DO** : DO
DO FOREVER
DO [FOR] count
DO WHILE/UNTIL test
DO var=start [TO limit] [BY step]
DO [FOR] count WHILE/UNTIL test
DO var=start [TO limit] [BY step] FOR count
DO var=start [TO limit] [BY step] WHILE/UNTIL test
DO var=start [TO limit] [BY step] FOR count WHILE/UNTIL test
Begin a block of instructions or a loop

[I] **ELSE** : ELSE [;] instruction
Introduce code to be executed when an IF test fails

[I] **END** : END [var]
Terminate a block of instructions beginning with DO or SELECT

[I] **EXIT** : EXIT [expr]
Terminate a script

[I] **IF** : IF test [;] THEN [;] instruction [ELSE [;] instruction]
Introduce code to be executed if *test* expression is True

[I] **INTERPRET** : INTERPRET [expr]
Execute ARexx instructions contained in a string

[I] **ITERATE** : ITERATE [var]
Skip to the end of the current iterative loop

[I] **LEAVE** : LEAVE [var]
Break out of current iterative loop

[I] **NOP** : NOP
Do nothing—a dummy instruction

[I] **OTHERWISE** : OTHERWISE [;] [instruction(s)]
Introduce default case for SELECT control structure

[I] **SELECT** : SELECT; WHEN ... [OTHERWISE [;] [instructions]] END
Branch to first case whose controlling condition is met

[I] **SIGNAL** : SIGNAL ON type
SIGNAL OFF type
SIGNAL [VALUE] label-expression
Turn interrupt *type* on/off; transfer control to label

[I] **THEN** : THEN [;] instruction
Execute dependent instruction if preceding expression was True

[I] **WHEN** : WHEN test [;] THEN [;] instruction
In SELECT, introduce code to be executed if *test* is True

Functions and arguments

[I] **ARG** : ARG [template] [,template ...]
Shorthand form of PARSE UPPER ARG

[F] **ARG** : numargs = ARG()
argn = ARG(num)
bool = ARG(num, mode)
Return information about script or function arguments

Modes: E - Exists O - Omitted

[I] **CALL** : CALL function(arg, arg, arg)
CALL function arg, arg, arg
Invoke a function but ignore the result

[I] **PROCEDURE** : PROCEDURE [EXPOSE var [var ...]]
Protect function caller's variables from name collisions

[I] **RETURN** : RETURN [expr]
Return control (and optionally a value) to caller

Strings (editing)

[F] **COMPRESS** : s = COMPRESS(str, [list])
Remove the characters in *list* (default=space) from *str*

[F] **DELSTR** : s = DELSTR(str, start, [len])
Delete *len* (default=rest of *str*) characters from *str*, from *start*

[F] **INSERT** : s = insert(istr,str,[start],[len],[pad])
Insert *len* characters of *istr* into *str* at *start*

```
[F] OVERLAY   : s = OVERLAY(new, old, [start], [len], [pad])
```
Overlay *len* characters of *new* on *old*, from *start*

```
[F] STRIP     : s = STRIP(str, [mode], [list])
```
Strip leading and/or trailing characters in *list* from *str*

```
Modes:   B - Both (default)   L - Leading   T - Trailing
```

```
[F] SUBSTR    : s = SUBSTR(str, start, [len],[pad])
```
Extract *len* (default=rest of *str*) characters of *str* , from *start*

```
[F] TRANSLATE : s = TRANSLATE(str)
                s = TRANSLATE(str,[output],[input],[pad])
```
Translate *str* using *input* and *output* character tables

```
[F] TRIM      : s = TRIM(str)
```
Remove trailing blanks from *str*

```
[I] UPPER     : UPPER var [var ...]
```
Convert contents of variables to upper case

```
[F] UPPER     : s = UPPER(str)
```
Convert *str* to upper case

Strings (pattern matching)

```
[F] ABBREV    : bool = ABBREV(str, abbr, [len])
```
Test if *abbr* is an abbreviation of *str* with at least *len* characters

```
[F] COMPARE   : n = COMPARE(str1, str2, [pad])
```
Determine position at which strings differ (0 if identical)

```
[F] INDEX     : n = INDEX(str,pat,[start])
```
Return position of *pat* in *str* , from 1 or *start* (0 if not found)

```
[F] LASTPOS   : n = LASTPOS(pat,str,[start])
```
Return position of *pat* in *str* , searching backwards from end or *start*

```
[F] POS       : n = POS(pat,str,[start])
```
Return position of *pat* in *str* , from 1 or *start* (0 if found)

```
[F] VERIFY    : n = VERIFY(str,list,[match],[start])
```
Search *str* from 1 or *start* for characters in *list*; return index or 0

```
Modes (match): M - Return index of first list character of str
               Other/default - Return index of first non-list character
```

Strings (formatting)

```
[F] CENTER    : s = CENTER(str, width, [pad])
```
Center *str* in a field of given *width*

```
[F] LEFT      : s = LEFT(str,count,[pad])
```
Extract leftmost *count* characters of *str* , pad on right if needed

[F] **REVERSE** : s = REVERSE(str)
Reverse *str*

[F] **RIGHT** : s = RIGHT(str,width,[pad])
Extract rightmost *width* characters of *str* pad on left if needed

[F] **SPACE** : s = SPACE(str, [len], [pad])
Set word-breaks in *str* to *len* spaces (or *pad*)

Strings (word-oriented)

[F] **DELWORD** : s = DELWORD(str, start, [len])
Delete *len* words from *str* beginning at *start* word

[F] **FIND** : n = FIND(str, phrase)
Return position of word-string *phrase* in *str*

[F] **SUBWORD** : s = SUBWORD(str, start, [count])
Extract *count* words (default=rest of *str*) from *str* from *start* word

[F] **WORD** : s = WORD(str,n)
Extract word *n* from *str*

[F] **WORDINDEX** : n = WORDINDEX(str,n)
Determine character position in *str* of start of word *n*

[F] **WORDLENGTH**: n = WORDLENGTH(str,n)
Determine length of word *n* in *str*

[F] **WORDS** : n = WORDS(str)
Return number of words in *str*

Strings (miscellaneous)

[F] **COPIES** : s = COPIES(str, n)
Concatenate *n* copies of *str*

[F] **HASH** : n = HASH(str)
Calculate a hash value for *str*

[F] **LENGTH** : n = LENGTH(str)
Return length of *str* in characters

[I] **PARSE** : PARSE [UPPER] source [template] [,template]

Source name	**Source string contents**
ARG	Arguments to script or function
EXTERNAL	Input via STDERR file
NUMERIC	NUMERIC options: *digits fuzz form*
PULL	Input via STDIN file
SOURCE	*type result called resolved ext host*
VALUE expr WITH	Result of expression
VAR varname	Contents of varname
VERSION	*version cpu mpu video freq*

Split input string(s) into substrings

[F] **XRANGE** `: s = XRANGE([c1],[c2])`
Build character string from *c1* to *c2* with consecutive ASCII values

Numbers

[F] **ABS** `: absval = ABS(num)`
Return absolute value of numeric expression

[F] **DIGITS** `: n = DIGITS()`
Return current NUMERIC DIGITS setting

[F] **FORM** `: f = FORM()`
Return current NUMERIC FORM setting

[F] **FUZZ** `: f = FUZZ()`
Return current NUMERIC FUZZ setting

[F] **MAX** `: n = MAX(n1, n2 [, n3 ...])`
Find the largest of a set of numbers

[F] **MIN** `: n = MIN(n1, n2 [, n3 ...])`
Find the smallest of a set of numbers

[I] **NUMERIC**
```
: NUMERIC DIGITS num
  NUMERIC FUZZ   num
  NUMERIC FORM SCIENTIFIC
  NUMERIC FORM ENGINEERING
```
Set numeric calculation and display options

[F] **RANDOM** `: n = RANDOM([low],[high],[seed])`
Return pseudo-random integer between *low* and *high* inclusive

[F] **RANDU** `: n = RANDU([seed])`
Return pseudo-random number between 0 and 1

[F] **SIGN** `: n = SIGN(num)`
Determine the sign of a number, return -1, 0, or 1

[F] **TRUNC** `: n = TRUNC(number,[places])`
Truncate *number* to *places* decimal places (default 0)

Bit manipulation

[F] **BITAND** `: s = BITAND(str1, [str2], [pad])`
Return bit-wise AND of two strings, padded to equal length

[F] **BITCHG** `: s = BITCHG(str, n)`
Invert bit *n*, counting from right, in *str*

[F] **BITCLR** `: s = BITCLR(str, n)`
Clear bit *n*, counting from right, in *str*

[F] **BITCOMP** : n = BITCOMP(str1, str2, [pad])
Return first bit, counting from right, at which *str1* ~= *str2* (-1 if equal)

[F] **BITOR** : s = BITOR(str1, [str2], [pad])
Return bit-wise OR of two strings, padded to equal length

[F] **BITSET** : s = BITSET(str, n)
Set bit *n*, counting from right in *str*

[F] **BITTST** : bool = BITTST(str, n)
Test bit *n*, counting from right, in *str*

[F] **BITXOR** : s = BITXOR(str1, [str2], [pad])
Return bit-wise exclusive-OR of two strings, padded to equal length

Data conversion

[F] **B2C** : s = B2C(binstr)
Return character equivalent of binary string

[F] **C2B** : b = C2B(str)
Return binary digit string equivalent to character string

[F] **C2D** : n = C2D(str, [len])
Return integer number corresponding to string (of *len* bytes)

[F] **C2X** : h = C2X(str)
Return hexadecimal number corresponding to string

[F] **D2C** : s = D2C(num,[count])
Return string equivalent, *count* bytes long, of *num*

[F] **D2X** : h = D2X(num,[count])
Return hexadecimal string, *count* bytes long, equivalent to *num*

[F] **X2C** : s = X2C(hex)
Return character equivalent of *hex* string

[F] **X2D** : n = X2D(hex,[n])
Convert *n* rightmost digits (or all) of *hex* string to decimal

Values and variables

[F] **DATATYPE** : type = DATATYPE(str)
bool = DATATYPE(str,mode)
Test attributes of string (*type* = 'NUM' or 'CHAR')

```
Modes:   A - Alphanumeric  B - Binary        L - Lower case
         M - Mixed case    N - Numeric       S - Symbol
         U - Upper case    W - Whole number  X - Hexadecimal
```

[I] **DROP** : DROP var [var ...]
Restore a variable to its uninitialized state

[F] **SYMBOL** : s = SUBWORD(str)
Determine if a string is a valid ARexx symbol

[F] **VALUE** : val = VALUE(str)
Treating str as an ARexx symbol, return its value

Console input/output

[I] **ECHO** : ECHO expr
Send the expression result to the standard output

[F] **LINES** : n = LINES([file])
Return number of lines queued for interactive stream

[I] **PULL** : PULL [template] [,template ...]
Shorthand form of PARSE UPPER PULL

[I] **PUSH** : PUSH expr
Pre-load the standard input in 'last-in, first-out' order

[I] **QUEUE** : QUEUE expr
Pre-load the standard input in 'first-in, first-out' order

[I] **SAY** : SAY [expr]
Send expression result to standard output (usually console)

File input/output

[F] **CLOSE** : bool = CLOSE(file)
Close the file with the given identifier

[F] **EOF** : bool = EOF(file)
Return 1 if end of file has been detected; else 0

[F] **OPEN** : bool = OPEN(file, name, [mode])
Open a file called *name* in given *mode*
Modes: R - Read (default) W - Write (create) A - Append

[F] **READCH** : s = READCH(file, [count])
Return *count* characters from *file*

[F] **READLN** : s = READLN(file)
Return a line from *file* as a string

[F] **SEEK** : n = SEEK(file,offset,[mode])
Move *file* position to *offset* according to *mode*
Modes: C - Current (default) B - Beginning E - End

[F] **WRITECH** : n = WRITECH(file, str)
Write *str* to *file*, return count written

[F] **WRITELN** : n = WRITELN(file, str)
Write *str* plus linefeed to *file*, return count written

Files

```
[S] DELETE     : bool = DELETE(name)
```
Delete a file or directory

```
[F] EXISTS     : bool = EXISTS(name)
```
Return True if given file or directory exists

```
[S] MAKEDIR    : bool = MAKEDIR(dirname)
```
Create a directory of the given name

```
[S] RENAME     : bool = RENAME(oldname, newname)
```
Rename a file or directory

```
[S] SHOWDIR    : filelist = SHOWDIR(dir, [mode], [pad])
```
List the file and/or directory names in a directory

```
Modes:   A - All (default)   F - Files only   D - Directories only
```

```
[S] STATEF     : filestring = STATEF(pathname)
```
Obtain information about a file or directory

```
Format: type size blk bits day min tick com
```

Script environment

```
[F] DATE       : d = DATE([outmode],[indate],[inmode])
```
Find today's date, or info about a specified date

```
Modes:   B - Base (days since 01/01/0000)
         C - Century (days this century)
         D - Days (days since start of year, counting today)
         E - European (dd/mm/yy - e.g. 19/11/76)
         I - Internal (days since 01/01/1978)
         J - Julian (yyddd - 2-digit year, 3-digit days this year)
         M - Month name in English mixed case (e.g. 'November')
         N - Normal, the default (dd mmm yyyy, e.g. 01 Jun 1986)
         O - Ordered (yy/mm/dd, e.g. 84/05/24)
         S - Standard (yyyymmdd, e.g. 19921005)
         U - USA (mm/dd/yy, 12/21/88)
         W - Weekday name in English mixed case (e.g. 'Thursday')
```

```
[F] ERRORTEXT  : text = ERRORTEXT(n)
```
Return a description of syntax error number *n*

```
[I] OPTIONS    : OPTIONS
                 OPTIONS [NO] RESULTS
                 OPTIONS [NO] CACHE
                 OPTIONS PROMPT [expr]
                 OPTIONS FAILAT expr
```
Set script options

```
[F] SOURCELINE: n = SOURCELINE()
                s = SOURCELINE(num)
```
Read count of source lines, or line *num* from current script

```
[F] TIME      : t = TIME([mode])
```
Find current or elapsed time

```
Modes:   C - Civil (h:mmAM or h:mmPM, e.g. 3:07AM)
         E - Elapsed (s.cc, seconds and hundredths, in interval)
         H - Completed hours since midnight (e.g. 4)
         M - Completed minutes since midnight (e.g. 243)
         N - Normal, the default (hh:mm:ss, e.g. 17:04:41)
         R - Same as elapsed, but resets timer to 0.00
         S - Completed seconds since midnight (e.g. 17353)
```

```
[I] TRACE     : TRACE mode
                TRACE [VALUE] expr
                TRACE num
```
Set tracing mode

```
Modes:   A - All clauses   B - Background      C - Commands
         E - Errors        I - Intermediates   L - Labels
         N - Normal        O - Off             R - Results
         S - Scan
Special features:
           ? - Interactive          +num - skip interactive pauses
           ! - Command inhibition   -num - trace suppression count
```

```
[F] TRACE     : t = TRACE([mode])
Get/set tracing mode (see modes under TRACE instruction)
```

ARexx environment

```
[F] ADDLIB    : ADDLIB(name, pri, [offset, version])
```
Add function library/host name to library list

```
[I] ADDRESS   : ADDRESS
                ADDRESS name
                ADDRESS VALUE name-expression
                ADDRESS name command-expression
```
Modify host address; send command to a host

```
[F] ADDRESS   : host = ADDRESS()
```
Return host address string

```
[F] FREESPACE : n = FREESPACE()
                n = FREESPACE(addr, size)
```
Return *size* bytes of memory at *addr* to ARexx's internal memory pool

```
[F] GETCLIP   : s = GETCLIP(clip)
```
Return value string associated with clip name

```
[F] GETSPACE  : addr = GETSPACE(size)
```
Allocate *size* bytes of memory in ARexx's internal memory pool

```
[F] REMLIB      : bool = REMLIB(name)
```
Remove an entry from the Library List

```
[F] SETCLIP     : bool = SETCLIP(name, [value])
```
Set string *value* for clip *name* (remove *name* if *value* omitted)

```
[I] SHELL       : SHELL
                  SHELL name
                  SHELL VALUE name-expression
                  SHELL name command-expression
```
Modify host address; send command to a host

```
[F] SHOW        : s    = SHOW(mode,,[pad])
                  bool = SHOW(mode,name)
```
Return information about a resource

```
Modes:   C - Clips                F - Files       I - Internal ports
         L - Function Libraries   P - Public ports
```

Messages, packets and ports

```
[S] CLOSEPORT : bool = CLOSEPORT(portname)
```
Close a message port opened with OPENPORT

```
[S] GETARG    : arg = GETARG(packet, [whicharg])
```
Obtain an argument string from a message packet

```
[S] GETPKT    : pkt = GETPKT(portname)
```
Pick up a message packet from a message port

```
[S] OPENPORT  : bool = OPENPORT(portname)
```
Open a public message port with the given name

```
[S] REPLY     : 1 = REPLY(pkt, [result], [result2])
```
Return a message packet to its sender, default results = 0, 0

```
[S] TYPEPKT   : cmd   = TYPEPKT(pkt)
                count = TYPEPKT(pkt,'a')
                bool  = TYPEPKT(pkt,mode)
```
Extract information from a message packet

```
Modes:      F - Function      C - Command
```

```
[S] WAITPKT   : bool = WAITPKT(portname)
```
Wait for a message packet to arrive at a port

Operating system

```
[S] ALLOCMEM  : mem = ALLOCMEM(size, [type])
```
Allocate *size* bytes of system memory, with *type* attributes

```
[S] BADDR     : addr = BADDR(bptr)
```
Convert BPTR to address

[S] **DELAY**
```
: 0 = DELAY(n)
```
Pause for *n* 50ths of a second

[F] **EXPORT**
```
: count = EXPORT(addr,[str],[len],[pad])
```
Copy *len* bytes of *str* to memory at *addr*

[S] **FORBID**
```
: count = FORBID()
```
Turn off multitasking

[S] **FREEMEM**
```
: 1 = FREEMEM(addr, size)
```
Free *size* bytes of memory at *addr* allocated by ALLOCMEM

[F] **IMPORT**
```
: s = IMPORT(addr,[len])
```
Return contents of *len* bytes (or 0-terminated) memory at *addr*

[S] **NEXT**
```
: value = NEXT(addr,[offset])
```
Return the 4-byte value stored at *addr* + *offset*

[S] **NULL**
```
: '00000000'x = NULL()
```
Return a 4-byte string corresponding to a null address

[S] **OFFSET**
```
: addr = OFFSET(addr,amount)
```
Return the address *addr* + *amount*

[S] **PERMIT**
```
: count = PERMIT()
```
Re-enable multitasking after FORBID

[F] **PRAGMA**
```
: oldcd = PRAGMA('d', [newcd]) /* Current directory */
  oldpri = PRAGMA('p', newpri)  /* Task priority     */
 oldsize = PRAGMA('s', size)    /* Stack size        */
       1 = PRAGMA('w', [mode])  /* DOS requesters    */
      id = PRAGMA('i')          /* Task ID (address) */
    bool = PRAGMA('*', [file])  /* Console handler   */
```
A grouping of system-specific facilities as one function

[S] **SHOWLIST**
```
: list = SHOWLIST(mode,,[pad])
  bool = SHOWLIST(mode, name)
  addr = SHOWLIST(mode, name,, 'a')
```
Return information about a shared system list

```
Modes:   A - Assigns       D - Devices       H - Handlers
         I - Interrupts    L - Libraries     M - Memory
         P - Ports         R - Resources     S - Semaphore
         T - Ready tasks   V - Volumes       W - Waiting tasks
```

[F] **STORAGE**
```
: n = STORAGE()
  s = STORAGE(addr,[str],[len],[pad])
```
Copy *len* bytes of *str* to memory at *addr*

Instruction and Function Reference

Name:	ABBREV
Type:	built-in function
Format:	bool = ABBREV(str, abbr, [len])
Description:	test if *abbr* is an abbreviation of *str*

***Dialog* examples:**

```
->abbrev("Delete","D")
1
->abbrev("Delete","0")
0
->abbrev("Delete","Del")
1
->abbrev("Delete","Del",3)
1
->abbrev("Delete","Del",4)
0
->abbrev("Delete","",0)
1
```

Discussion: ABBREV's boolean return value reflects whether *abbr* is a valid abbreviation of *str*. The optional third argument, *len*, which must be numeric, specifies a minimum length for *abbr* if it is to be considered a valid abbreviation; the default is the length of *abbr* itself.

One typical use for ABBREV is in scripts that allow the user to enter special commands or keywords in incomplete form. Having obtained the input with, say:

```
pull cmd
```

the script can go on to test the input and take appropriate actions with SELECT or an IF/ELSE chain (see Chapter 7):

```
select
   when abbrev("LIST",cmd)   then call ListThings()
   when abbrev("DELETE",cmd) then call DeleteThings()
   when abbrev("DECIDE",cmd) then call DecideThings()
   otherwise
      say "Invalid command!"
   end
```

In this example, there is an ambiguity between the 'DELETE' and 'DECIDE' commands: if the user typed either 'D' or 'DE', 'DELETE' would be understood even though 'DECIDE' may have been intended. Moreover, a null input would actually be taken as an abbreviation for 'LIST'! The remedy is to provide *len* explicitly. Using 3 for the third argument in each call to ABBREV would be sufficient to distinguish all the commands in this small set. If preferred, the minimum length for the first call (testing the input against 'LIST') could be set to 1, since a single 'L' could have no other valid meaning.

Name:	ABS
Type:	built-in function
Format:	absval = ABS(num)
Description:	return absolute value of numeric expression

***Dialog* examples:**

```
->abs(-3)
3
->abs(11)
11
->abs(-24.75e-12)
2.475E-11
```

Discussion: ABS is often used in expressions that check whether a value deviates by more than a given amount, above or below, from another value. If you were writing a military simulation, you might want to test whether a dropped bomb fell close enough to a particular target to destroy it. You could use ABS in a *CheckBlast* function, like this:

```
/* hit = CheckBlast(bombx, tgtx, blast)

   bombx: Bomb position
   tgtx : Target position
   blast: Radius of bomb blast
   hit  : Function returns 1 if blast destroys target
*/
CheckBlast: procedure
   parse arg bombx, tgtx, blast
   return abs(tgtx-bombx) <= blast
```

See also: sign

Name:	ADDLIB
Type:	built-in function
Format:	ADDLIB(name, pri, [offset, version])
Description:	add function library/host name to library list

Dialog examples:

```
->addlib('rexxsupport.library',0,-30,0)
1
->addlib('rexxsupport.library',0,-30,0)
0
->addlib('MyFunctionHost',20)
1
```

Discussion: The search order for ARexx functions has several phases, beginning (typically) with internal functions, then proceeding to the built-in functions, then to the function libraries and hosts on the Library List. These are searched in priority order, from a maximum priority of 100 to a minimum of -100. New library and host names are added to the Library List with ADDLIB, although no other action takes place (loading the library, or establishing communications with the host) until ARexx actually encounters the newly-added name during a function name search. This means that the name is not tested for validity by ADDLIB; it is quite possible to add an invalid name to the Library List. If you do so, the first you will likely hear about is when you get one of the following error messages during an attempt by ARexx to match a function name:

```
Error 13: Host environment not found
Error 14: Requested library not found
```

The first two arguments to ADDLIB are the name of the library or host to be added, and the priority the library or host should have in the list. In the case of a function library, the name given must match the library's file name in the 'libs:' directory, and the name stored within the library itself. Only the latter must match for case; as usual, file names are case-insensitive. The name given for a function host is that of the host's public message port. The correct name for a host or library should be provided as part of its documentation. The priority argument, from -100 to 100, is up to the caller. Values close to 0 are fine in most situations. Priority values of -60 or lower should rarely be used, since -60 is the priority of the ARexx Resident Process, which is responsible for the more time-consuming search for external functions.

ADDLIB's *offset* and *version* arguments apply only for function libraries, and may be omitted for function hosts. *Offset* is invariant for

a particular library, and will be given in the library's documentation. It is essential that this argument be given correctly, as an erroneous value will probably result in a system crash. *Version,* which specifies the minimum version number of the library acceptable to the caller, can usually be given as 0 (or omitted). The library's documentation will give information about the *version* argument if it is important.

The return value from ADDLIB tells you whether the name you supplied was in fact added to the list. The normal reason for a False (0) return is that the name was on the list already, as in the second example above. (The first example, incidentally, assumes the support library is not already loaded. If it was, of course, the first ADDLIB call would also return 0.)

In practice, it is much more usual to use ADDLIB on function libraries than on function hosts, since the latter are usually embedded in programs that add the host for you when run.

See also: remlib, show

For further discussion of the Library List and function name searches, see Chapter 6.

Name:	ADDRESS
Type:	instruction
Format:	ADDRESS
	ADDRESS name
	ADDRESS VALUE name-expression
	ADDRESS name command-expression
Description:	modify host address; send command to a host

Script example:

```
/* Script: Features of the ADDRESS instruction
*/
say address()          /* With rx, initial host is REXX      */
address                /* toggle addresses                   */
say address()          /* output: COMMAND                    */

newhost = 'GRACIOUS HOST' /* set up a variable               */
address newhost        /* ...but newhost is taken literally */
say address()          /* output: NEWHOST                    */
address 'newhost'      /* ...again literally                 */
say address()          /* output: newhost                    */
address value newhost 'INDEED'  /* but with VALUE keyword */
say address()          /* output: GRACIOUS HOST INDEED       */
address rexx 'hello' /* executes 'hello.rexx'                */
```

Discussion: Every ARexx script has a 'command host' at its service from the time it begins to execute. The command host is identified by its name, which may be retrieved at any time with the ADDRESS function (not the ADDRESS instruction, the subject of this entry). The job of the command host is to process any lines in the script that are not recognized as valid instructions by ARexx itself. In actuality, the line may be an error—a mistyped instruction, perhaps—rather than a command. It is up to the command host to make that determination.

When a script is launched by the RX program, the initial command host has the name REXX. So what does the REXX host (which is built right into the ARexx system) do when it receives a potential command? Something quite interesting: It tries to match the first part of the name with that of an existing ARexx script file. If it is successful, it executes this secondary script (which may in turn launch other scripts before it terminates), then gives control back to the original script, which has been been waiting patiently all the while to resume operations.

When a script is launched by an application program, rather than by RX, the initial command host will normally be built into that program, and named accordingly. A-Talk III, a terminal program, uses ATK for its host name. Electric Thesaurus uses ETHES_1. And so on—the host name used by a particular application is really quite arbitrary.

Now consider—as a thought experiment—running an ARexx script first from RX, and subsequently from an imaginary application whose host name we'll (arbitrarily, of course) decide is 'IMAG_APP'. The first line in the script is a comment, as usual, but the second line is:

```
'hello world'
```

Since 'hello' is not an ARexx instruction keyword, the ARexx interpreter will decide that this line must be a command intended for the current command host. If the script was run from RX, the command host is REXX, so 'hello world' will be sent to REXX for further processing. As we have seen, REXX will try to locate a script file to execute. It will try several different variants on the name 'hello' before either finding one and executing it or giving up and returning an error.

But what if we ran the script from our imaginary application? In that case, the 'hello world' line would be sent to the 'IMAG_APP' host, and then... well, anything might happen. Although we expect that 'IMAG_APP' will do something logical and gratifying with the

command in accordance with whatever type of application it may be, the nature of ARexx itself does not restrict it in any way.

Every script also has an 'alternate' command host, and may toggle back and forth between the principal and alternate hosts at will. Initially, the alternate host has the name COMMAND. Its behavior is somewhat similar to that of REXX, but instead of trying to execute the command as an ARexx script, it searches the current AmigaDOS path for an ordinary AmigaDOS command of that name, and executes it if the search is successful.

To switch between the main and the alternate command hosts, use the first form of the ADDRESS instruction, with no arguments.

The second form of ADDRESS specifies a new host address, which replaces the current principal address (leaving the alternate address undisturbed). The name given may be in quotes or not as you choose: the only difference is that if it is not the name will be taken as being in upper case no matter how you actually type it. In particular, even if the name is a variable, the name itself rather than the variable's value is taken as the new host name.

Since that may not be what you want, a third form of ADDRESS uses the sub-keyword VALUE to announce that the expression following should be evaluated to determine the new host name. Even if the 'expression' is just a variable name, this time it is the contents of the variable rather than the name that will be used.

All three forms of ADDRESS so far discussed change the current host address for the script until further notice, and that is all they do. The fourth and final form instead sends out one command to a particular specified host, but has no effect on subsequent commands. This form is like the second in that the host name is taken literally, even though it may also be the name of an existing variable. The command itself is evaluated as an ARexx expression before being sent to the temporarily selected host.

And what happens if you try to address a non-existent host? Nothing—until you try to use a command (meaning, as you recall, an instruction ARexx doesn't recognize.) Then you will get the error message 'Host environment not found', as ARexx suddenly realizes your so-called host is only a figment.

See also: address (built-in function), shell

Discussions of the ADDRESS instruction and command hosts in a real-world context may be found in Chapter 3 and Chapter 12.

Name:	ADDRESS
Type:	built-in function
Format:	host = ADDRESS()
Description:	return host address string

Script example:

```
/* ADDRESS example */
say address()
address command
say address()
address 'Ebenezer'
say address()
```

Example output:

```
REXX
COMMAND
Ebenezer
```

Discussion: When ARexx encounters an instruction it does not recognize, the instruction is evaluated as an ARexx expression and the result passed off to the current 'host address'. By default, for scripts launched with the RX program, the host address is REXX; this default host will search the usual places for an ARexx script whose name matches the first word in the instruction. Other hosts, which can be selected with the ADDRESS instruction, will treat the command in ways appropriate to their purposes.

One use for the ADDRESS function is to determine the current and alternate host addresses before switching to a new host, so that the old ones can be restored afterwards. For instance:

```
HA_main = address()
address               /* switch to alternate */
HA_alt  = address()

address HA_newmain    /* change host address */
address HA_newalt     /* ... completely      */

/* Now do things involving new hosts ... then restore */

address HA_alt
address HA_main
```

Name:	ALLOCMEM
Type:	support function
Format:	mem = ALLOCMEM(size, [type])
Description:	allocate memory from operating system

Script example:

```
/* ALLOCMEM demo - get then free 10000 bytes PUBLIC CHIP memory
   The ☞ character indicates that a line
   should be entered as a single line
*/
signal on syntax  /* Do this to trap allocation failures */
size = 10000
say "The system currently has" storage() "bytes of memory ☞
   available."
mem = allocmem(size, '00000003'x)  /*CHIP = 2, PUBLIC = 1 */
say "After allocating" size "bytes, there are" storage() ☞
   "bytes."
say "The address of our allocation is $"c2x(mem)"."
call freemem(mem, size)
drop mem
say "After freeing our memory, there are now" storage() ☞
   "bytes."
exit

syntax:
   if rc=3 then
      say "A memory allocation has failed!"
   else
      say "Syntax error #"rc"."

   if symbol('mem')='VAR' then do
      call freemem(mem,size)
      say "Allocated memory has been freed."
      end
```

Discussion:

The occasions on which an ARexx programmer has to allocate memory are few. ARexx itself, along with most well-written function libraries, insulates the programmer from the operating system so completely that such a low-level operation is hardly ever needed. And even on those rare occasions, the STORAGE function in the built-in library is generally a better choice than ALLOCMEM, since the memory it allocates is a controlled resource that ARexx will free automatically when your script ends, even if the ending is premature due to a syntax or other error.

There are two reasons why ALLOCMEM would be required in some instances, however:

1) Since, as just mentioned, allocations made with STORAGE are freed when the script ends, ALLOCMEM is needed if the allocation must outlive the script.

2) Only ALLOCMEM lets you use the special attribute flags, such as the CHIP memory flag (see below).

The first argument to AllocMem is the size of the allocation needed, in bytes. The second argument, which is optional, is a 4-byte string specifying special attributes, as follows:

Attribute	Value	Meaning
PUBLIC	1	Memory is sharable between tasks
CHIP	2	Memory can be used by custom chips
FAST	4	Memory cannot be used by custom chips
CLEAR	65536	Memory will be zeroed on allocation

These attributes can be combined (except that CHIP and FAST are obviously incompatible and will cause the allocation to fail no matter what its size). The default attribute is PUBLIC, and this should always be used by ARexx scripts, along with such other attributes as may be required. To convert the attributes to a 4-byte string as required by the function, add the values together and convert with D2C:

```
attr = d2c(1+2+65536,4) /* PUBLIC CHIP memory, pre-cleared*/
say c2x(attr)           /* output: 00010003              */
```

The return value from ALLOCMEM is the address of the allocated memory, in the form of a 4-byte string.

A final consideration in using ALLOCMEM is ensuring that any allocations made are properly freed. This can get especially complicated if multiple allocations are made, but even in simple cases a syntax trap should be set to ensure that the memory is freed under all possible terminations of the script. See the example script above for one method of handling the problem in the elementary case.

See also: freemem, export, import, getspace, freespace, storage

Name:	ARG
Type:	instruction
Format:	ARG [template] [,template ...]
Description:	shorthand form of PARSE UPPER ARG

Script example:

```
/* ARGTEST.rexx       Usage: rx argtest <any text> */
arg line
do i=1 to words(line)
   say word(line,i)
   end
```

Example session:

```
Shell> rx argtest Where Alph the sacred river ran
WHERE
ALPH
THE
SACRED
RIVER
RAN
```

Discussion:

Since ARG means exactly the same thing as PARSE UPPER ARG, you will find a detailed treatment of it in the entry for PARSE. The advantage of ARG is simply its conciseness.

The fact that ARG implies the UPPER subkeyword of PARSE is a drawback or a bonus depending on the context. If the arguments you're extracting are numbers, the conversion obviously doesn't matter; if they are to be checked against stored string patterns in a case-insensitive match, it cuts out the step of using the UPPER instruction separately. If the argument string(s) will be echoed back to the user in some form, though, it is probably best to use PARSE ARG.

See also:

parse, pull, arg (built-in function)

Chapter 10 discusses PARSE extensively, and should be read by those who are not yet familiar with its many features.

Name:	ARG
Type:	built-in function
Format:	numargs = ARG()
	argn = ARG(num)
	bool = ARG(num, mode)
Description:	return information about script or function arguments

Code examples:

```
if arg() ~= 2 then
   say "Incorrect number of arguments"

if arg(2,'e') then
   width = arg(2)
else
   width = 1
```

Discussion: ARG has several forms.

1) If no arguments are given, ARG returns the number of arguments available to the present script or function. In the case of a script called as a command with the RX program, the number of arguments will be 0 or 1, depending on whether command-line arguments were given. Functions (including scripts called as external functions) may have multiple arguments.

```
num_args = arg()
```

2) If a single numeric argument is given, ARG returns the corresponding argument to the present script or function. If there is no such argument, a null string is returned. The argument number given must be an integer greater than 0.

```
width = arg(2)
```

3) A *mode*, either 'e' for 'Exists' or 'o' for 'Omitted', may be used along with the numeric first argument to test whether the corresponding argument to the present script or function was supplied or omitted. ARG's boolean return reflects whether the given condition is satisfied.

```
if arg(2,'Omitted') then width = 100 /* default */
```

Extracting the arguments given to a function or script is most simply done with the PARSE instruction in ordinary situations. Some functions, though, are designed to behave differently given different numbers of arguments, or to supply default values if specific

arguments are not given. Other functions, especially those designed to be used over and over in different programs, may need to validate their arguments in some way (see the first code example above). In both these cases, the ARG function should be used as either an adjunct to or a substitute for PARSE.

See also: parse

For further discussion of ARG, see Chapter 8 (Internal Functions).

Name:	B2C
Type:	built-in function
Format:	s = B2C(binstr)
Description:	return character equivalent of binary string

***Dialog* examples:**

```
->b2c('01000110')
F
->b2c(reverse('01000110'))
b
->b2c(translate('-=------- -=---=--','01','-='))
OK
```

Discussion: As explained in Chapter 5 (under 'Other String Forms'), character strings can equivalently be represented as a string of the binary digits 0 and 1. ARexx lets you use such binary strings directly, for example:

```
->'01000110'B
F
```

Note that this bit-string is completely different from this string of digits:

```
->'01000110'
01000110
```

Appending a 'B' to a string to make it binary works only for 'literal' (quoted) strings. The B2C function lets you build the binary string as an expression, rather than having to give it literally. Otherwise, the conversion is exactly the same, and the same rules apply as to the literal strings: only the digits 0 and 1 are allowed, with optional blanks at byte boundaries (every 8 digits, counting from the right).

B2C is of interest mainly to advanced programmers who need to build 'bit-masks' for use with ARexx bit-manipulation functions like BITAND.

See also: c2b, bitand, bitchg, bitclr, bitcomp, bitor, bitset, bittst, bitxor

Name:	BADDR
Type:	support function
Format:	addr = BADDR(bptr)
Description:	convert bptr to address

***Dialog* examples:**

```
->c2x(baddr('01010101'x))
04040404
->c2x(baddr('01f87de5'x))
07E1F794
->baddr('UUUU')
UUUT
```

Discussion: A consequence of the colorful history of the Amiga's operating system is that there are two ways of referring to memory addresses: as bytes, and as long words (4-byte chunks). In general, AmigaDOS uses the second method, while the rest of the operating system uses the first method.

Fortunately, there is a simple and exact correspondence between the two systems: a byte address will be exactly 4 times greater than the equivalent long-word address. The byte address is known as a C-pointer (or simply pointer) to the memory in question; the long-word address is called a B-pointer, customarily abbreviated to BPTR.

Although it is simple enough to translate from pointers to BPTRs by hand, it is even simpler to call the BADDR function, which takes a BPTR in the form of a 4-byte string, and returns a CPTR as another 4-byte string, obtained by multiplying the input string by 4. It doesn't make sense to apply this treatment to a string of ordinary characters, but it can be done, as the third example illustrates. If after a minute or two you don't know why taking the BADDR of 'UUUU' returns the curious result 'UUUT', try putting yourself in the place of BADDR and figure out what it would mean to multiply 'UUUU' by 4.

See also: The script 'asnvol.rexx' on the disk accompanying this book uses BADDR in context, though like most uses of this function it is rather esoteric.

Name:	BITAND
Type:	built-in function
Format:	s = BITAND(str1, [str2], [pad])
Description:	return bit-wise AND of two strings

Dialog examples:

```
->c2b(bitand('10110001'b,'11001111'b))
10000001
->bitand('ono','1110001'b,'c')
abc
->'['bitand('12345')']'
[     ]
```

Discussion: BITAND takes three arguments. The first two are strings that are ANDed together bit by bit: the result string will have 1 bits in those positions where both argument strings have 1 bits, and 0 bits in all other positions. If one argument string is shorter than the other, it is padded on the right with spaces or with the pad character given as the optional third argument. The result string is always the same length as the longer of the two argument strings.

As the third of the Dialog examples shows, the second argument to BITAND is also optional; it will be taken as an empty string if not given, then duly padded with spaces to the length of the first argument. As it happens, any digit (or any lower-case letter, or almost any punctuation character) ANDed with a space character yields a space as the result, hence the output in the example. It is not suggested that you take advantage of this behavior in real scripts, however.

A typical use of a bit-wise AND operation in languages like C is to clear individual bit-flags in a set of flags packed into one or a few bytes. Another use is to test individual flags as a basis for subsequent actions. In ARexx, clearing individual flags is most efficiently done with BITCLR, and testing them is best done with BITTST. BITAND is thus relegated to more specialized duties, such as clearing an entire bit-field (several related flags within a larger set), or masking out several flags as a prelude to some operation involving only them.

See also: b2c, c2b, bitchg, bitclr, bitcomp, bitor, bitset, bittst, bitxor

Name:	BITCHG
Type:	built-in function
Format:	s = BITCHG(str1, n)
Description:	invert specified bit in string

Dialog examples:

```
->bitchg('A',5)          /* bit-pattern for 'A': 01000001 */
a                        /* bit-pattern for 'a': 01100001 */
->bitchg('a',5)
A
```

Discussion: BITCHG takes two arguments: a string, and the number of the bit in the string that should be inverted (changed to 0 if it is 1, to 1 if it is 0). Bits are numbered from the right, starting at 0, so the maximum value for the second argument is 1 less than 8 times the number of characters in the string (there being 8 bits in a character, or byte).

As with several of the other bit-oriented functions, the applications of BITCHG in normal scripts are rather limited.

See also: b2c, c2b, bitand, bitclr, bitcomp, bitor, bitset, bittst, bitxor

Name:	BITCLR
Type:	built-in function
Format:	s = BITCLR(str1, n)
Description:	clear specified bit in string

Dialog examples:

```
->bitclr('o',0)            /* bit-pattern for 'o': 01101111 */
n                          /* bit-pattern for 'n': 01101110 */
->bitclr(bitclr('o',0),2)
j                          /* bit-pattern for 'j': 01101010 */
```

Discussion: BITCLR takes two arguments: a string, and the number of the bit in the string that should be cleared (changed to 0). Bits are numbered from the right, starting at 0, so the maximum value for the second argument is 1 less than 8 times the number of characters in the string (there being 8 bits in a character, or byte).

As with several of the other bit-oriented functions, the applications of BITCLR in normal scripts are rather limited.

See also: b2c, c2b, bitand, bitchg, bitcomp, bitor, bitset, bittst, bitxor

Name:	BITCOMP
Type:	built-in function
Format:	n = BITCOMP(str1, str2, [pad])
Description:	compare two strings bit by bit

Dialog examples:

```
->bitcomp('10010000'b,'10011000'b)
3
->bitcomp('Aab','aab')
21
->bitcomp('Aab','b','a')
21
->bitcomp('10101010'b,'10101010'b)
-1
```

Discussion: BITCOMP compares its two argument strings from the rightmost bit leftward, halting when the bits fail to match. The bit-number of the non-matching pair of bits, counting from zero as the rightmost bit, is returned. If the strings are identical, -1 is returned. If one string is shorter than the other, it is padded on the left with spaces, or with the pad character supplied as the optional third argument.

As with several of the other bit-oriented functions, the applications of BITCOMP in normal scripts are rather limited.

See also: b2c, c2b, bitand, bitchg, bitclr, bitor, bitset, bittst, bitxor

Name:	BITOR
Type:	built-in function
Format:	s = BITOR(str1, [str2], [pad])
Description:	return bit-wise OR of two strings

Dialog examples:

```
->bitor('A','B')                    /* 'A': 01000001  'B': 01000010 */
C                                   /* 'C': 01000011                */
->c2b(bitor('10101010'b,'01010101'b))
11111111
->bitor('00000000 00000000'b,'','*')
**
->bitor('The Quick Brown Fox!')
the quick brown fox!
```

Discussion: BITOR takes three arguments. The first two are strings that are ORed together bit by bit: the output string will have 1 bits in those positions where either or both of the argument strings has a 1 bit, and 0 bits where both argument strings have 0 bits. If one argument string is shorter than the other, it is padded on the right with spaces or with the pad character given as the optional third argument. The result string is always the same length as the longer of the two argument strings.

As the third of the above examples shows, the second argument to BITOR is also optional; it will be taken as an empty string if not given, then duly padded with spaces to the length of the first argument. Provided the given string does not contain any 'control characters' (linefeed, carriage return, formfeed, tab, and the like), nor any of these punctuation characters:

```
@ [ \ ] ^ _
```

you can use BITOR as a LOWER (force to lower case) function, which is otherwise not included in the built-in library. This is demonstrated in the fourth example. It is safer, however, even if more cumbersome, to use the TRANSLATE function for this purpose.

A typical use of a bit-wise OR operation in languages like C is to set individual bit-flags in a set of flags packed into one or a few bytes. In ARexx, setting individual flags is most efficiently done with BITSET. BITOR is thus relegated to more specialized duties, such as setting all the bits in an entire bit-field (several related flags within a larger set), or creating a mask for several flags as a prelude to some operation involving only them.

See also: b2c, c2b, bitand, bitchg, bitclr, bitcomp, bitset, bittst, bitxor

Name:	BITSET
Type:	built-in function
Format:	s = BITSET(str1, n)
Description:	set specified bit in string

Dialog examples:

```
->bitset('A',5)          /* bit-pattern for 'A': 01000001 */
a                        /* bit-pattern for 'a': 01100001 */
->bitset('a',1)
c                        /* bit-pattern for 'c': 01100011 */
```

Discussion: BITSET takes two arguments: a string, and the number of the bit in the string that should be set (changed to 1 if it is not 1 already). Bits are numbered from the right, starting at 0, so the maximum value for the second argument is 1 less than 8 times the number of characters in the string (there being 8 bits in a character, or byte).

As with several of the other bit-oriented functions, the applications of BITSET in normal scripts are rather limited.

See also: b2c, c2b, bitand, bitchg, bitclr, bitcomp, bitor, bittst, bitxor

Name:	BITTST
Type:	built-in function
Format:	bool = BITTST(str, n)
Description:	test specified bit in string

***Dialog* examples:**

```
->bittst('A',5)          /* bit-pattern for 'A': 01000001 */
0
->bittst(import('00bfe001'x,1),7)
                         /* check joystick fire button    */
1
```

Discussion: BITTST takes two arguments: a string, and the number of the bit in the string that should be tested. Bits are numbered from the right, starting at 0, so the maximum value for the second argument is 1 less than 8 times the number of characters in the string (there being 8 bits in a character, or byte). The result is 1 if the bit was set; 0 if the bit was clear.

One of the factors limiting the usefulness of the other bit-oriented functions is that on the Amiga, hardware registers—to which bit operations are often applied—should not be modified without observing access rules that are hard to comply with from ARexx. BITTST is a 'read-only' operation, however, and may be used with hardware registers more freely. A good example is the use of BITTST in a routine to read a joystick:

```
/* ReadJoyStick

   This function returns a string of 1 to 3 characters
   describing the state of the joystick in port #2. The
   first character describes the fire button, either 0 (up)
   or 1 (down). The second and third characters give the
   direction of the stick, as n, ne, e, se, s, sw, w, nw. If
   there is no second character, the stick is centered.
*/
```

```
ReadJoyStick: procedure

   fb = ~bittst(import('00bfe001'x,1),7)
   js = import('00dff00c'x,2)

   if bittst(js,1) then
      jd = 'e'
   else if bittst(js,9) then
      jd = 'w'
   else
      jd = ''

   if bittst(js,1) ^ bittst(js,0) then
      jd = 's'jd
   else if bittst(js,9) ^ bittst(js,8) then
      jd = 'n'jd

   return fb || jd
```

See also: b2c, c2b, bitand, bitchg, bitclr, bitcomp, bitor, bitset, bitxor

Name:	BITXOR
Type:	built-in function
Format:	s = BITXOR(str1, [str2], [pad])
Description:	return bit-wise exclusive-OR of two strings

***Dialog* examples:**

```
->bitxor('A','7')              /* 'A': 01000001  '7': 00110111 */
v                              /* 'v': 01110110                */
->bitxor('TheQuickBrownFox')
tHEqUICKbROWNfOX
->bitxor('the quick brown fox','','D')
0,!d51-'/d&6+3*d"+<
```

Discussion: BITXOR takes three arguments. The first two are strings that are exclusive-ORed together bit by bit: the output string will have 1 bits in those positions where exactly one of the argument strings has a 1 bit, and 0 bits elsewhere. If one argument string is shorter than the other, it is padded on the right with spaces or with the pad character given as the optional third argument. The result string is always the same length as the longer of the two argument strings.

As the second of the above examples shows, the second argument to BITXOR is also optional; it will be taken as an empty string if not given, then duly padded with spaces to the length of the first argument. As it happens, any letter XORed with a space character will be flipped in case, giving the output in the example. It is not suggested that you take advantage of this behavior in real scripts, however.

The third example shows a use for BITXOR in a primitive encryption scheme, in which every character of the text to be encrypted is exclusive-ORed with a key-character (here 'D') to produce impressively garbled output. Repeating the operation will restore the original text. Although it is very easy to encrypt large amounts of text or any other data with this simple method, it is obviously also quite easy to crack.

See also: b2c, c2b, bitand, bitchg, bitclr, bitcomp, bitor, bitset, bittst

Name:	BREAK
Type:	instruction
Format:	BREAK
Description:	exit from innermost DO-END block or INTERPRET instruction

Script example:

```
/* BREAK example: the job interview */
options prompt "Are you here about the job (y/n)? "
pull yn
hire = 0

if left(yn,1)='Y' then do
   options prompt "So you want work... How old are you? "
   pull age; if age < random(16,60,time('s')) then break

   options prompt "That's old enough! Your name? "
   pull name; if length(name)>random(4,20) then break

   options prompt "That's short enough! Your weight? "
   pull weight; if weight<random(75,200) then break

   say "That's heavy enough! You're hired!"
   say "Now, we'll be starting you off as a brain surgeon..."
   hire = 1
   end
else do
   say "Well, the door is over that way."; exit
   end

if ~hire then
   say "Sorry, you're not qualified for this position."
```

Discussion: It is a generally-held programming ideal that every logical unit of a program should have a single entry point, and a single exit point. Efforts to preserve that ideal in the rough and tumble of real-world programming have led to the development of the 'control structures' nearly every program (or ARexx script) requires: if-then-else constructs, 'while' and 'until' loops, counted loops, 'case' structures (like ARexx's SELECT), functions and procedures. With all these tools

available, the use of instructions that contravene the single-entry, single-exit rule is seldom desirable.

Nonetheless, most languages provide at least a couple of types of these abuse-prone instructions; it is up to the programmer to apply them properly. In ARexx, the BREAK, LEAVE, ITERATE and SIGNAL instructions are of this type, along with some usages of EXIT and RETURN.

BREAK, LEAVE and ITERATE have a family resemblance: they are all concerned with escaping the confines of a currently active control structure. Their main benefit is in avoiding the excesses of fussy indentation: try recoding the example above without using BREAK and you'll discover that in some cases a small cost in structural purity is amply repaid by improved readability. And the example could easily have been made longer...

BREAK's special property is that it can be applied to any DO-END block, even a 'non-iterative' (non-looping) one—a mere compound statement like the one in the example. LEAVE is legal only in an iterative DO.

Another special ability of BREAK is that of aborting from an INTERPRET instruction. It being difficult to devise a clear and meaningful but short example of this, here is an example that is, well, at least reasonably short. If you find it opaque, treat it as a puzzle to be solved... You will notice that the word 'BREAK' does not occur in it anywhere:

```
/* BREAK from INTERPRET */
rc=0; options prompt ">"; say "ARexx Command Shell"
s = "pull t 'EXIT' 0 'RETURN' 0 t; interpret t';'s"
syntax: signal on syntax; if rc>0 then say errortext(rc)
interpret s
```

See also: do, interpret, leave, iterate, signal

Name:	C2B
Type:	built-in function
Format:	b = C2B(str)
Description:	return binary digit string equivalent to character string

***Dialog* examples:**

```
->c2b('F')
0100010

/* Count 1 bits in 'Rumpelstiltskin' */
->length(compress(c2b('Rumpelstiltskin'),'0'))
64
```

Discussion: This function is sometimes useful as a binary conversion calculator (as in the first example), and may be handy in occasional bit-oriented operations that deal with substrings rather than individual bits. The bit-counting trick of the second example illustrates the kind of higher-level operation for which it is more appropriate to process the bits as a character string than individually with BITAND and its fellows.

See also: b2c, c2d, c2x

Name:	C2D
Type:	built-in function
Format:	n = C2D(str, [len])
Description:	return integer number corresponding to character string

***Dialog* examples:**

```
->c2d('a')
97
->c2d('0')
48
->c2d('00'x)
0
->c2d('ff'x)
255
->c2d('03ff'x,1)
-1
```

Discussion: This function takes a character string 1 to 4 bytes in length, and returns the value obtained by treating the string as a 4-byte binary number. For instance, the string 'frog' consists of 4 characters, which appear in memory as follows (represented in hexadecimal):

```
66 72 6f 67
```

Taken together, these bytes could also be viewed as the hexadecimal number $66726f67, which is equivalent to the decimal number 1718775655. This is the number that would be returned by:

```
c2d('frog')
```

More typical are the first two examples above, which apply C2D to a single character in order to determine its ASCII equivalent. The second example shows that the character '0' and the number 0 are quite different things: the character is represented by 48 in the ASCII system. On the other hand, the hexadecimal string '00' of the third example really is zero.

C2D always works with a 4-byte string, padding the given string on the left with null bytes as necessary. But if the optional second argument is given, as in the last example, the string is padded or truncated as required to the given number of characters, then sign-extended to 4 characters. In the example, the '03ff', which is 2 characters long, is truncated to the single character 'ff', which is then sign extended to 4 characters giving 'ffffffff', which has the decimal equivalent -1.

See also: c2x, c2b, d2c

Name:	C2X
Type:	built-in function
Format:	h = C2X(str)
Description:	return hexadecimal number corresponding to character string

***Dialog* examples:**

```
->c2x('a')
61
->c2x('Cc')
4363
->c2x('123456'x)
123456
->c2x(import('00000004'x,4))
07e007d8  /* ExecBase address (value is system-dependent) */
```

Discussion: This function takes a character string of arbitrary length, and returns the hexadecimal equivalent of the string. The final example above shows this function being used to display the contents of memory in hexadecimal (as preferred by many programmers).

See also: c2b, c2d, x2c

Name:	CALL
Type:	instruction
Format:	CALL function(arg, arg, arg)
	CALL function arg, arg, arg
Description:	invoke a function but ignore the result

Script example:

```
/* CALL example */
call left('scores',5)
call right result, 4
say result                /* output: core */
```

Discussion: The original purpose of calling functions in any language was to obtain a value, either for its own sake or as a component of a larger expression. Though it is still useful to think of this as the primary use of functions, in many functions the computation of a value is only one part of their behavior, while for others it has become a mere vestige—for form's sake only—or has been abandoned altogether. In all of these cases, the generation of 'side effects', such as opening a file or allocating memory, is the main point, and any computation as such is secondary.

The CALL instruction provides a way of explicitly invoking a function that does not return a value, or whose return value is not of interest. If there is a return value, however, it is not simply thrown away, but is stored in the RESULT variable, regardless of whether OPTIONS RESULTS has been used. If the function does not return a value, the RESULT variable is 'dropped'.

Another special quirk of CALL is that the parentheses normally required around the function argument list are optional, as the second line of the example demonstrates. ARexx's parent language REXX actually disallows parentheses in this context, so the examples of CALL in this book are in a sense non-standard. Since nothing is gained by having two forms of function argument lists, however, and consistency is lost, we recommend you follow our practice and use the parentheses.

Function calls made with CALL are in all other respects the same as function calls made within expressions. The function name is matched by the usual process of searching first internal, then built-in, then library, then external functions, and the internal functions can as usual be bypassed by putting the function name in quotes.

See also: Chapter 6 (Compound Variables and Built-in Functions) covers the search order for matching function names in detail, and provides a general introduction to functions and function libraries.

Name:	CENTER or CENTRE
Type:	built-in function
Format:	s = CENTER(str1, width, [pad])
Description:	center a string in a field of given width

***Dialog* examples:**

```
->'['center('How I Spent Last Summer',31)']'
[    How I Spent Last Summer    ]
->'['center('How I Spent Last Summer',31,'=')']'
[====How I Spent Last Summer====]
->center('How I Spent Last Summer',15)
I Spent Last Su
```

Discussion: This useful function lets you center a string on a field of either spaces (the default) or some other character you give as the *pad* argument. The result string will have the length given in the *width* argument, even when (as in the third example) this is shorter than the string to be centered.

Though CENTER is particularly handy for centering titles and the like on the monitor display, or on a printed page, it also lends itself to slightly more exotic uses as in this little script:

```
/* Draw diamond pattern */
n = 22
do i=-n to n by 2
   say center(copies('/',n-abs(i)),n+2,'\')
   end
```

See also: left, right

Name:	CLOSE
Type:	built-in function
Format:	bool = CLOSE(file)
Description:	close the given file

Code example:

```
call close('datafile')
```

Discussion: Attempts to close an open file nearly always succeed, and in any case CLOSE does not seem to be able to distinguish successful closures from failed ones. In fact, it appears that the only condition that causes CLOSE to return False is the file not having been open in the first place. Accordingly, it is normal practice not to test the return from CLOSE, and the example reflects this. It is nevertheless possible for a file not to be closed after CLOSE has been called, though it does not happen very often.

ARexx automatically closes all open files when a script exits, but it is considered better practice to close them explicitly.

See also: open

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	CLOSEPORT
Type:	support function
Format:	bool = CLOSEPORT(portname)
Description:	close a message port opened with OPENPORT

***Dialog* examples:**

```
->openport('OldPort')
1
->closeport('OldPort')
1
->closeport('OldPort')
0
```

Description: CLOSEPORT closes a public message port previously opened with OPENPORT. The only argument is the name of the port, and the return value is boolean, confirming that the port did exist and has been closed. Because ports opened with OPENPORT are a controlled resource that ARexx will automatically free when a script ends, calling CLOSEPORT may be considered optional in most scripts. Most programmers would agree, however, that closing the port explicitly is a good practice.

One thing you must not do is try to use CLOSEPORT on a message port created other than by OPENPORT. The failure of the port to close properly is by far the best of several bad things that might result from this error.

See also: openport, getpkt, waitpkt, typepkt, reply

Name:	COMPARE
Type:	built-in function
Format:	n = COMPARE(str1, str2, [pad])
Description:	return index at which argument strings differ

Dialog examples:

```
->compare('rain','snow')
1
->compare('trials','tribulations')
4
->compare('endpad    ','endpad')
0
->compare('employee','employ','e')
0
```

Discussion: COMPARE is used when you want to know not just whether but where two strings differ. The comparison starts with the leftmost character. The shorter string is padded on the right with spaces (the default) or with the optional *pad* character. The return value is the index of the first character that does not match (numbering the leftmost character as 1), or 0 if the strings are equal.

This function is of limited application, since most string comparisons are performed with the equality and exact equality operators (see Chapter 5).

Name:	COMPRESS
Type:	built-in function
Format:	s = COMPRESS(str, [list])
Description:	remove the characters in list (default = space) from str

Dialog examples:

```
->compress("Now here")
Nowhere
->compress("Aloha Oahu - Aloha Hawaii!","AIOUaiou")
lh h - lh Hw!
->compress("'Oh!' he said, sighing - and vanished!",".!?,;:-'")
Oh no he said sighing  and vanished
```

Discussion: Like the SPACE function, the default form of COMPRESS (represented by the first example), simply removes all spaces from the string. If the second argument is given, however, COMPRESS removes from the first string every character that is in the second string. This may be used, for instance, to remove characters that would complicate parsing (as in the third example).

See also: space, strip, translate, trim

Name:	COPIES
Type:	built-in function
Format:	s = COPIES(str, n)
Description:	concatenate n copies of str

***Dialog* examples:**

```
->copies("+",16)
++++++++++++++++
->copies("Rah! ",3)
Rah! Rah! Rah!
->copies("-",length("Chapter Two: Elementary Concepts"))
------------------------------
```

Discussion: COPIES forms a new string from the given number of repetitions of the argument string. This is particularly useful for quickly generating underlines, borders and separators. COPIES can also be used algorithmically to create various kinds of simple graphics, as in this little script that uses it to generate a histogram:

```
/* Quick histogram */
data = "30 44 11 17 45 30 39 28 26 40"
title = "Hat Sales, 1980-1989 ($Millions)"

say center(title,60)
say center(copies("=",length(title)),60)
say

do i=0 to 9
   say center(1980+i,10)copies("$",word(data,i+1))
   end
```

Name:	D2C
Type:	built-in functions, D2C
Format:	s = D2C(num,[count])
Description:	return character equivalent of number

***Dialog* examples:**

```
->d2c(65)
A
->d2c(65+32)
a
->d2c(1131573111)
Crow
->d2c(1131573111,2)
ow
```

Discussion: Any string of 4 characters has an equivalent representation as an integer; correspondingly, any integer in the range 0 to 2147483647 can be converted to an equivalent character string with D2C. The optional second argument, from 0 to 4, can be used to control the number of characters, starting from the end of the string, that will be returned. The default is to return as many characters as are needed to express the number (eliminating up to 3 leading zero-bytes).

A common use for D2C is to generate characters, such as the 'control' and the 'alt' characters, which may be impossible or inconvenient to enter via the keyboard. As an example, this little script outputs a table of all the alt characters (ASCII codes 160 through 255):

```
/* Show Alt characters */
say "  0 1 2 3 4 5 6 7 8 9 A B C D E F"
say "  --------------------------------"
do i=160 to 255 by 16
   line = d2x(i%16)
   do j=i to i+15
      line = line d2c(j)
      end
   say line
   end
```

See also: c2d

Name:	D2X
Type:	built-in function
Format:	h = D2X(num,[count])
Description:	return hexadecimal string equivalent to number

***Dialog* examples:**

```
->d2x(16)
10
->d2x(1023)
3FF
->d2x(2121212121)
7E6F20D9
->d2x(2121212121,5)
F20D9
```

Discussion: The number, which must be a positive integer between 0 and 2147483647, is converted to a hexadecimal string of up to 8 characters. If the second argument, a number from 0 to 8, is present, zeros are added to or characters are removed from the left of the output string to achieve the specified length.

See also: x2d

Name:	DATATYPE
Type:	built-in function
Format:	type = DATATYPE(str)
	bool = DATATYPE(str,mode)
Description:	find out about attributes of string

***Dialog* examples:**

```
->datatype(123)
NUM
->datatype('One-Two-Three')
CHAR
->datatype('10011001','b')
1
->datatype(3173.89E-2,'n')
1
```

Discussion: DATATYPE is useful in determining whether a particular string conforms to some requirement. In the absence of the optional *mode* argument, DATATYPE returns either 'NUM' or 'CHAR', according to whether the string argument is or is not a valid number (including negative and fractional numbers, and numbers expressed in exponential notation). If the mode argument is given, the string is tested to see if it conforms with the requirements of that mode, and returns 1 if the string passes the test, 0 otherwise. The following modes are valid (only the first letter is significant in the actual call, as shown in the third and fourth Dialog examples):

Mode	The argument string...
Alphanumeric	May contain only alphabetic characters (a-z and A-Z) and digits (0-9).
Binary	May contain only the characters 0 and 1. Spaces are allowed every 8 digits counting from the right (e.g. '1010 11001111 01010010').
Lower case	May contain only the characters a-z.
Mixed case	May contain only the characters a-z and A-Z.
Numeric	Must be a valid number.
Symbol	Must be a valid ARexx symbol (such as could be used as a variable or function name).
Upper case	May contain only the characters A-Z.

Mode	The argument string...

Whole number	Must be an integer. This test accounts for the current NUMERIC DIGITS setting, as shown in the following fragment:
	`numeric digits 4`
	`say datatype(1.0001,'w') /* output: 0 */`
	`numeric digits 3`
	`say datatype(1.0001,'w') /* output: 1 */`

X (Hexadecimal)	May contain only the characters 0-9, a-f and A-F. Spaces are allowed every 2 characters counting from the right (e.g. '1 2A 30 E9')

Name:	DATE
Type:	built-in function
Format:	d = DATE([outmode],[indate],[inmode])
Description:	find today's date, or info about a specified date

Dialog examples:

```
->date()
21 Jul 1991
->date('u')
07/21/91
->date('e')
21/07/91
->date('w',19910815,'s')
Thursday
```

Discussion: ARexx's DATE function is a little complicated, but very comprehensive and versatile. Any of 12 different modes can be given for the first argument. In the absence of the other arguments, these specify different information to be returned about today's date, along with the date format. The modes, which may be abbreviated to a single letter as usual, are:

Mode	Date information returned...
Base	The number of complete days since the base date 01/01/0000. Current values are in the neighborhood of 727 thousand. DATE('b')//7 returns the day of the week as a number, with 0 representing Saturday. The returned string is an unpunctuated number with no leading zeros.
Century	The number of complete days since the beginning of the 20th century. The value returned is in the same form as DATE('b'), but is less by 693,960.

Mode	Date information returned...
Days	The number of days, including today, since the beginning of the current year. The returned string is a number between 1 and 366.
European	The date in the format dd/mm/yy (e.g. "21/07/91").
Internal	The number of days since the conventional Amiga start date of 01/01/78. The value returned is in the same form as DATE('b'), but is less by 722,450.
Julian	The date in the format yyddd, where ddd is the number of days since the beginning of the year (same as returned by DATE('d')). Both yy and ddd are padded on the left with zeros if necessary to make 2 and 3 digits respectively.
Month name	The full name of the month in English mixed case.
Normal	This is the default format returned by DATE with no arguments. It is in the form dd mmm yyyy. The day, dd, is padded to 2 digits with a leading zero if necessary. The month, mmm, is the first 3 letters of the full month name as returned by DATE('m').
Ordered	The date in the format yy/mm/dd, suitable for sorting. All 3 fields are padded to 2 digits if necessary.
Standard	The date in the format yyyymmdd, suitable for sorting. The mm and dd fields are padded to 2 digits if necessary.
USA	The date in the format mm/dd/yy (e.g. "07/21/91").
Weekday	The full name of the weekday in English mixed case.

The mode given as the first argument is normally applied to the current date as known to the system, but an alternate date can be given as the second argument if desired. This date can be given in either the Internal or the Standard formats. The Internal format is the default; if the Standard format is used, the 's' mode specifier must be used as the third argument, as in the final example above. The only other allowed mode for the third argument is the default, 'i'.

The DATE function has, in common with TIME, the special property that multiple calls within a single instruction will be mutually

consistent: neither the calendar nor the clock will (be observed to) advance between the calls.

See also: time

Name:	DELAY
Type:	support function
Format:	0 = DELAY(n)
Description:	pause for n 50ths of a second

```
Script example:
 /* DELAY */
 se = "SPECIAL EFFECT!"'0a'x

 do i=1 to length(se)
    call delay(6)
    call writech('STDOUT',substr(se,i,1))
    end
```

Discussion: DELAY lets your script do nothing at all for the given number of fiftieths of a second: it simply waits out the time. An important subtlety on the multitasking Amiga is that DELAY waits in the correct way, without hogging system resources. It provides a way of avoiding such barbarities as 'timing loops' like:

```
do 1000; end
```

which use up just as many processor cycles as they would in honest computation, simply by doing nothing in the wrong way.

The return value from DELAY is not meaningful.

Name:	DELETE
Type:	support function
Format:	bool = DELETE(name)
Description:	delete a file or directory

CLI example:

```
Shell> rx "say delete('ram:nullity')
   /* file does not exist */
0
Shell> echo >ram:nullity
Shell> protect ram:nullity -d
Shell> list ram:nullity NOHEAD
nullity                    empty ----rwe- Today      14:31:37
Shell> rx "say delete('ram:nullity')
```

```
   /* file delete-protected */
0
Shell> protect ram:nullity +d
Shell> list ram:nullity NOHEAD
nullity                    empty ----rwed Today      14:31:37
Shell> rx "say delete('ram:nullity')
   /* Should work this time */
1
```

Description: DELETE can delete either a file or an empty directory. As the example shows, the function will fail (returning 0) if the named file or directory does not exist, or is protected from deletion by clearing the 'd' file attribute with the AmigaDOS Protect command. DELETE will also fail if the file is currently in use (having been opened by another script, perhaps), if the disk it is on is write-protected, and so on.

See also: rename, makedir

Name:	DELSTR
Type:	built-in function
Format:	s = DELSTR(str, start, [len])
Description:	delete len characters from str

***Dialog* examples:**

```
->delstr('piglet',4)
pig
->delstr('piglet',2,3)
pet
```

Discussion: DELSTR's result string is the argument string with a specified run of characters removed. The deletion begins at the character whose position is given as *start*. Character positions are counted from 1, the leftmost character. If the start position is greater than the length of the argument string, no deletion is performed. The default length of the deletion is from the start position to the end of the argument string; the length may be specified in a third argument if desired.

See also: insert, overlay

Name:	DELWORD
Type:	built-in function
Format:	s = DELWORD(str, start, [len])
Description:	delete len words from str, beginning at start word

Dialog examples:

```
->'['delword('Always stand up for your beliefs',4)']'
[Always stand up ]
->'['delword('Never say never again',2,2)']'
[Never again]
```

Discussion: DELWORD's result string is the argument string with a specified run of words removed. The deletion begins at the word whose position is given as the second argument. Words are counted from 1, the leftmost word. If the start word is greater than the number of words in the argument string, no deletion is performed. The default length of the deletion is from the start word to the end of the argument string; the number of words to delete may be specified in a third argument if desired. Any spaces to the right of a deleted word are also removed.

As usual in ARexx's built-in word functions, only actual space characters are taken as word boundaries. Other characters often regarded as 'white space', such as tabs and linefeeds, are treated as belonging to words, not delimiting them.

See also: find, subword, word, wordindex, wordlength, words

Name:	DIGITS
Type:	built-in function
Format:	n = DIGITS()
Description:	return current NUMERIC DIGITS setting

Script example:

```
/* The DIGITS function */
say digits()          /* (default): 9              */
say 1/9               /* output: 0.111111111       */
numeric digits 14
say digits()          /* (maximum): 14             */
say 1/9               /* output: 0.11111111111111 */
```

Discussion: DIGITS returns the maximum number of significant digits used for the display of numeric values. Sometimes it is desirable to save the current precision setting before modifying it (with the NUMERIC DIGITS instruction), in order that it can be restored later on. DIGITS may also be used to validate the argument given to NUMERIC FUZZ, which must be less than the current DIGITS setting:

```
numeric fuzz digits()-1  /* maximum fuzz */
```

See also: form, fuzz

Name:	DO
Type:	instruction
Format:	DO
	DO FOREVER
	DO [FOR] count
	DO WHILE/UNTIL test
	DO var=start [TO limit] [BY step]
	DO [FOR] count WHILE/UNTIL test
	DO var=start [TO limit] [BY step] FOR count
	DO var=start [TO limit] [BY step] WHILE/UNTIL test
	DO var=start [TO limit] [BY step] FOR count WHILE/UNTIL test
Description:	begin a block of instructions or a loop

Script example:

```
/* DO - Random number test */
trials = 10       /* Adjust # of trials to suit */
target = random(1,100,time('s'))
total  = 0

do trials
   do i=1 while target ~= random(1,100)
      end
   total = total + i
   end

/* 'Expected' average is close to 100 over many trials */
say "Average of" trials "trials:" trunc(total/trials+.005,2)
```

Discussion:

Where most languages provide a number of different constructs for loops, ARexx provides just one, but that one has wide capabilities. The simplest form of DO simply introduces a block of instructions that looks 'from the outside' like a single instruction. An instruction block can be made conditional on a single IF/THEN, ELSE or WHEN, for instance.

All forms of the DO instruction require a matching END after the last instruction in the block. This is true even if the block contains no instructions at all, as shown by the inner loop in the example script above.

The instruction formats shown above give all the useful forms of DO. Although the ARexx interpreter allows some variations in the ordering, and some redundant combinations of forms, such as:

```
do i=3 by 2 to 15     /* by...to = to...by  */
do 20 forever         /* same as do 20      */
do forever 20         /* same as do forever */
```

none adds any advantages over the standard forms presented. The first of these is the simple form discussed above; the remaining 8 are the 'iterative' or 'looping' forms. Of these, the second 4 are combinations of the first 4. Here we'll briefly consider the iterative forms in turn:

```
DO FOREVER
```

This loop will never terminate of its own accord, just as advertised. There are nevertheless several ways to escape the loop (without rebooting!). The BREAK, LEAVE, RETURN, EXIT and SIGNAL instructions all provide means of breaking out of the loop, as documented in their separate entries; BREAK and LEAVE are the preferred choices as they are the least disruptive of the flow of control. Of course, the support program HI or interactive tracing can be used to break out of a runaway loop if need be.

```
DO [FOR] count
```

The numeric expression 'count' is evaluated, and the loop is repeated that many times (barring early exit with LEAVE etc). The FOR keyword was needed in the original release of ARexx but is now optional. The outer loop in the example above demonstrates this form of DO.

```
DO WHILE/UNTIL test
```

Either of the keywords WHILE or UNTIL may be used (not both) plus the logical expression 'test'. If WHILE is used, 'test' is evaluated upon entry to the loop, and after each iteration. The loop continues to execute as long as the result of the evaluation is 1 (True). If the result is 0 (False), control is passed to the first instruction immediately following the loop's END instruction. The inner loop in the example above demonstrates the WHILE form of DO. If UNTIL is used, the loop is executed once without first evaluating 'test'. The expression is evaluated then, and after each subsequent iteration. The loop terminates when the expression yields 1.

```
DO var=start [TO limit] [BY step]
```

Upon entry to this loop, the numeric expression 'start' is evaluated and the result is assigned to the variable 'var', which is called the 'index variable' or 'loop counter variable'. If the TO phrase is present, the numeric expression 'limit' is evaluated and the result is stored. If

the TO phrase is absent, there is no limit value for the loop. If the BY phrase is present, the numeric expression 'step' is evaluated and the result is stored. If the BY phrase is absent, the step value defaults to 1. None of these expressions is re-evaluated while the loop is executing, so the calculated values cannot be altered by instructions within the loop. The inner loop of the example above is an example of this form of DO with both the TO and BY phrases absent. The result is that the index variable is incremented by 1 on each 'pass' through the loop, without limit.

Before each iteration of the loop, including the first, the value of the index variable is compared with the stored limit value, if there is one. If the step value is positive and the index variable is greater than the limit, or the step value is negative and the index variable is less than the limit, the loop terminates. After each iteration of the loop, the index variable is incremented by the step value. The value of the index variable can also be modified directly (e.g. by assignment) within the loop, though this is generally discouraged as poor programming practice.

Combinations In all combinations of the preceding types, the loop will be ended by whichever terminating condition, of which there may be up to 3, is first activated.

See also: end, break, leave, iterate

For more examples and discussion of DO in its many forms, see Chapter 7 (Compound Statements and Loops).

Name:	DROP
Type:	instruction
Format:	DROP var [var ...]
Description:	restore a variable to its uninitialized state

Script example:

```
/* DROP example script */
say elephant
elephant = "pachyderm"; say elephant
drop elephant
say ouch    /* on the elephant's behalf */
say elephant
```

Discussion: Every ARexx variable name can be used in an expression wherever a string is expected, whether or not the variable has ever been assigned a value (initialized). When an uninitialized variable is used, its value is taken to be the variable name itself, nearly always in upper case (the exception being when a compound variable name is derived by substituting in a lower-case or mixed-case value).

DROP allows you to return one or more variables to their uninitialized state. If the stem of a compound variable is dropped, as in:

```
drop names.
```

all variables formed on that stem are dropped.

See also: symbol

Name:	ECHO
Type:	instruction
Format:	ECHO expr
Description:	send the expression result to the standard output

Shell example:

```
Shell> rx "echo 'Thane of Glamis'
Thane of Glamis
Shell> rx >ram:macbeth "echo 'Thane of Cawdor'
Shell> type ram:macbeth
Thane of Cawdor
```

Discussion: ECHO is a synonym of the more usual SAY instruction.

See also: say

Name:	ELSE
Type:	instruction
Format:	ELSE [;] instruction
Description:	introduce code to be executed when an IF test fails

Script example:

```
/* ELSE example: marry.rexx */
options prompt "Will you marry me (y/n)? "
pull yn

if yn = 'Y' then
   say "Oh joy! Oh bliss! (Am I ready for this?)"
else if yn = 'N' then
   say "Oh woe! Oh grief! (And a sense of relief.)"
```

```
else do
   say "Hmmm. I thought the question was clear enough."
   say "Shall I take your ambiguous reply for a maybe?"
   end
```

Discussion: ELSE is a valid instruction only when it immediately follows the instruction succeeding an IF/THEN. ELSE requires a dependent instruction of its own. The dependent instruction may be simple:

```
else
   a = 3
```

or it may be a compound instruction:

```
else do
   a = 3
   b = 4
   end
```

or it may be a complete IF/THEN/ELSE sequence, as in the example. The example also illustrates the way in which chained IF/THEN/ELSE instructions deviate from normal indentation, so that instead of:

```
if <test> then
   <instruction>
else
   if <test> then
      <instruction>
   else
      if <test> then
         <instruction>
```

and so on, we use the more natural, informal, compact and understandable:

```
if <test> then
   <instruction>
else if <test> then
   <instruction>
else if <test> then
   <instruction>
```

If a chain of this type goes on much further, however, it is both more readable and probably a bit more efficient to recode it as a SELECT instruction.

It is not always obvious to which IF a particular ELSE pertains. Consider this:

```
if <test1> then
   if <test2> then
      instruction
   else
      instruction
```

The indentation of the ELSE here suggests that it belongs with the second IF, not the first, and that is in fact the case: ELSE always attaches to the most recent unmatched IF available. So how would you force the ELSE to attach to the first IF instead? Here are two ways, equivalent in effect:

```
1) if <test1> then do
      if <test2> then
         instruction
      end
   else
      instruction

2) if <test1> then
      if <test2> then
         instruction
      else
         nop
   else
      instruction
```

See also: if, nop, select, do

Name:	END
Type:	instruction
Format:	END [var]
Description:	terminate a block of instructions beginning with DO

Script example:

```
/* END example - doesn't do a lot, but it's legal */
do
end
```

Discussion: END is valid in one context only: as the terminal instruction in a compound instruction or loop beginning with DO or SELECT. In a DO loop that uses an 'index variable', the name of the variable may optionally follow the matching END instruction. Following END with a non-matching variable name is an error, so this can be used to verify that an END and its corresponding DO are in the correct relationship.

See also: do, select

Name:	EOF
Type:	built-in function
Format:	bool = EOF(file)
Description:	return 1 if end of file has been detected; else 0

Script example:

```
/* This script assumes the file "ram:test" was
   created with the shell command:
      Shell> echo >ram:test "Any string"
*/
if open('testfile','ram:test','r') then do
   say "File newly opened, eof() returns" eof('testfile')
   call readln('testfile')
   say "One line read, now eof() returns" eof('testfile')
   call readln('testfile')
   say "Another line read, eof() returns" eof('testfile')
   call seek('testfile',0,'b')
   say "At start position, eof() returns" eof('testfile')
   call close('testfile')
   end
else
   say "Can't open file 'ram:test'."
```

Discussion: EOF reports on the end of file condition for a given file. As the example shows, the condition is not set until an unsuccessful read attempt has been made. The example also demonstrates that the end of file condition is cancelled by a call to SEEK. In fact, even a SEEK of zero bytes relative to the current position (that is, a null SEEK) will cancel the EOF until another unsuccessful read has been performed.

See also: readch, readln, seek

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	ERRORTEXT
Type:	built-in function
Format:	text = ERRORTEXT(n)
Description:	return a description of syntax error number n

***Dialog* examples:**

```
->errortext(1)
Program not found
->errortext(29)
Incomplete IF or SELECT
->errortext(36)
Keyword conflict
```

Discussion: In some kinds of scripts it is desirable to trap syntax errors, using the SIGNAL ON SYNTAX instruction. If this is done, ARexx gives you a numeric code for each error via the RC variable, but does not display a message describing the error in words. ERRORTEXT can be used to obtain the text message corresponding to the numeric code.

Only non-negative integers are accepted as arguments to ERRORTEXT. Integers that are not valid error numbers return the text "Undiagnosed internal error". In the version of ARexx current at the time this book was going to press (version 1.15), the valid error numbers ranged from 1 to 48. A few numbers in that range return an empty string, presumably because of an evolution in ARexx's error handling from previous versions.

See also: The Dialog script given in Chapter 5 contains an example of the kind of use of ERRORTEXT discussed above, and illustrates the difference between syntax errors and errors returned by commands.

Name:	EXISTS
Type:	built-in function
Format:	bool = EXISTS(name)
Description:	return true if given file or directory exists

***Dialog* examples:**

```
->exists('sys:c')
1
->exists('c:list')
1
->exists('ram:ectoplasm')
0
->exists('df0:')
1
->exists('df2:')
0
```

Discussion: Scripts will sometimes need to take different actions depending on the presence or absence of a particular file or directory. In the case of a database file, for example, the script may be required to open the file with 'r' (read mode) if it exists, or with 'w', creating it, if it does not. EXISTS provides a quick way of checking the existence of the file without bothering the user.

If the path name being checked refers to a device that is not in the system, a volume that is not mounted, or a logical directory that has not been assigned, EXISTS (through AmigaDOS) will normally post a requester asking the user to make the 'volume' available. If this requester would be inappropriate, the script can suppress requesters

before calling EXISTS by calling the PRAGMA function, then (typically) re-enable them afterwards with another call to PRAGMA:

```
call pragma('W','n')
result = exists('ichabod:crane')
call pragma('W','W')
```

Remember that the mere existence of a filing system object of a given name doesn't tell you anything about its type, since the naming rules for files and directories are the same. It is even possible for a file to masquerade as a device or volume by giving it a logical device name with the AmigaDOS Assign command. In most situations you may be able to ignore possibilities like these, but if you do need more information about the name than merely that it exists, you should use the STATEF function in the support library.

See also: statef, showdir

Name:	EXIT
Type:	instruction
Format:	EXIT [expr]
Description:	terminate a script

Script examples:

```
/* Script 1 - ExitTest.rexx
   Try this with no args, numeric args and string args.
*/
exit arg(1)

/* Script 2 - CallExitTest.rexx
   Try this with no args, numeric args and string args.
   ExitTest.rexx must be in your current directory or
   in rexx: for this to work.
*/
say exittest(arg(1))

/* Script 3 - DoExitTest.rexx
   Try this with various args. Try commenting out the
   OPTIONS RESULTS line and/or the OPTIONS FAILAT line.
   ExitTest.rexx must be in your current directory or
   in rexx: for this to work.
*/
options results
options failat 20
'exittest' arg(1)
say "RESULT = '"result"', RC =" rc
```

Discussion: EXIT terminates the current script. If an argument expression is provided, the expression is evaluated and the result returned to the caller. If the caller happens to be the RX program, only a numeric result is acceptable. RX will report an arithmetic conversion error if you use:

```
EXIT "Well, that's that!"
```

The outcome of passing back an expression result to other callers depends on the context of the call. If the called script is invoked as a function (the way the second script in the example invokes the first), the expression is simply returned as the function value in the usual way.

If the called script is invoked as a command (the way the third script in the example invokes the first), the meaning of the expression depends on whether OPTIONS RESULTS has been used. If it has, the returned value is stored in the RESULT variable, and the return code (RC) is 0. If OPTIONS RESULTS is not used, a numeric return is treated as a return code, and is stored in RC. If the returned value is greater than the current OPTIONS FAILAT threshold, the error will be reported. An attempt to return a non-numeric string when OPTIONS RESULTS is not used is treated as an error with a return code of 10.

See also: return, options

Name:	EXPORT
Type:	built-in function
Format:	count = EXPORT(addr,[str],[maxlen],[pad])
Description:	copy a string to a memory area

Script example:

```
/* Demo script using EXPORT with 100-byte buffer */
buf = getspace(100)

/* Export string to buffer, then fetch it with IMPORT */
say export(buf, "Good morning!") "bytes copied to buffer."
say "Buffer contains '"import(buf)"'."

/* Complete form of EXPORT, IMPORT just 40 bytes back */
say export(buf, import(buf) import(buf), 50, "+");
say "Buffer contains '"import(buf, 40)"'."

/* Free buffer (not strictly needed - see FREESPACE) */
call freespace(buf, 100)
```

Discussion: EXPORT provides a method of storing data into any memory location from within an ARexx script. In practice, this capability is rarely needed, though it might be useful in conjunction with some function libraries, or with specific ARexx-supporting application programs. The contrived nature of the example above reflects the abstruseness of this function.

EXPORT's first argument is a 4-byte address, which if given literally would normally be expressed in hexadecimal (e.g. '0024e098'x). This address is the destination of the exported data. The other three arguments are all optional. The second argument is a string to be copied to the given address; the third argument is the maximum number of characters to be copied (the default is the length of the string); the fourth argument is a pad character used to make up the length if the string is too short. The default pad character is the null byte ('00'x). EXPORT returns the number of bytes transferred to the buffer.

Because EXPORT writes directly into system memory, its incorrect use can cause the failure of, or aberrant behavior by, any task or tasks in the system, or a system crash. ARexx is unable to protect you from yourself when using this function.

See also: import, getspace, freespace, storage, allocmem, freemem

Name:	FIND
Type:	built-in function
Format:	n = FIND(str, phr)
Description:	return position of word or phrase in string

***Dialog* examples:**

```
->find("1 2 3 4 5 6 7", "3 4 5")
3
->find("1234567","345")
0
->find("12 345 67","345")
2
```

Discussion: FIND searches for a multi-word phrase (the second argument) in a string (the first argument), and returns the word index, counting from 1, of the first occurrence, or 0 if the phrase is not found. To be matched, the phrase must occur in the string as complete words, bounded by spaces or the ends of the string.

See also: delword, subword, word, wordindex, wordlength, words, index

Name:	FORBID
Type:	support function
Format:	count = FORBID()
Description:	turn off multitasking

Script example:

```
/* FORBID */
parse version . . cpu .
if right(cpu,2)>=20 then limit=20000; else limit=5000

say "Forbid() returns" forbid()
say "I'm going to count to" limit" (it'll take a few seconds)."
say "Until I'm finished you won't be able to use the computer,"
say "because I'm the only task in business, and I won't be"
say "listening to you. Try moving the mouse for instance..."
do limit; end
say "Permit() returns" permit()
```

Description: Not less often than every 4 video frames, 15 times a second, the Amiga's operating system takes control of the computer away from the currently executing task and gives it to the task who is next in turn to run. Tasks do not know, and generally need not be concerned about, exactly when their 'time slices' occur—from their point of view they are executing continuously. That's what multitasking is all about.

In some circumstances, though, switching away from a task at a particular moment could be dangerous. During a memory allocation, for instance, the information that all tasks share about system memory is being updated, and for some tiny fraction of a second is not internally consistent. If a task switch happened just at that point and another task tried to use the same information, it would almost certainly cause, or at least sow the seeds of, a system crash. Because memory allocations occur so often, this dire situation would surely happen eventually if measures were not taken to avoid it.

What actually happens in a memory allocation, as with many other activities involving shared data, is that task-switching is very briefly locked out so that the data can be modified safely. The lock-out mechanism can be accessed from ARexx with the functions FORBID and PERMIT. As you would expect from their names, FORBID prevents task-switching, and PERMIT re-enables it.

Calls to FORBID and PERMIT must balance: if you call FORBID twice you must be sure to call PERMIT twice for multitasking to be restored. The number of times FORBID has been called without a matching PERMIT is called the 'nesting count'; both functions return the value

of this count that results from calling them. When a call to PERMIT undoes the last level of FORBID, the resulting nesting count is -1.

Another way in which multitasking may temporarily be re-enabled is if your task enters a waiting state, which happens—to name only one of many instances—when your script communicates with an ARexx host. As soon as your script starts running again, though, the FORBID is renewed. If a section of code requires a FORBID to guarantee the continuing integrity of some data, you must be careful not to allow multitasking inadvertently by using some function or instruction that will put you into a wait. Commands that use external files, or functions that are handled by a function host, the SAY, (PARSE) PULL and TRACE instructions, to name only a few, will all have that effect, so don't use them within a FORBID.

In the example script, the loop in the second last line, which takes several seconds to complete, executes without ever going into a wait. Therefore during those seconds it as though no other task was running in the computer. Even the 'input task', which watches for keyboard and mouse activity, is paralyzed.

So, armed with all this theory, when should you call FORBID and PERMIT? Well, probably never. Unless you're going to write scripts that snoop system data structures, it isn't likely you'll ever need to block normal multitasking. And it's preferable not to, if you have the choice: FORBIDding is one of those things that well-behaved programs do only when necessary, and even then for as short a time as possible. But the capability is there if you need it.

See also: permit

For an example that actually does require the protection of FORBID and PERMIT, see the Reference Section entry for NEXT function.

Name:	FORM
Type:	built-in function
Format:	f = FORM()
Description:	return current NUMERIC FORM setting

Script example:

```
/* The FORM function */
say form()                  /* (default): SCIENTIFIC  */
say 1/9e12                  /* output: 1.11111111E-13 */
numeric form engineering
say form()                  /* output: ENGINEERING    */
say 1/9e12                  /* output: 111.111111E-15 */
```

Discussion: FORM returns the current exponential format used for displaying numbers that are too large or small to be expressed in normal numeric format. The function returns either 'SCIENTIFIC' or 'ENGINEERING'. Sometimes it is desirable to save the current form setting before modifying it (with the NUMERIC FORM instruction), in order that it can be restored later on.

See also: digits, fuzz

Name:	FREEMEM
Type:	support function
Format:	1 = FREEMEM(addr, size)
Description:	free memory allocated by allocmem

Script example:

```
/* FREEMEM */
mem = allocmem(5000)        /* allocate 5000 bytes    */
say "Obtained 5000 bytes at address $"c2x(mem)"."
call freemem(mem, 5000)     /* ... and free it again */
```

Discussion: FREEMEM's first argument is an address previously returned by ALLOCMEM, and its second argument is the size of the allocation at that address. FREEMEM always returns 1.

See also: allocmem, export, import, getspace, freespace, storage

The entry for ALLOCMEM contains both a detailed discussion of issues relating to memory allocation, and a fuller example of using both ALLOCMEM and FREEMEM.

Name:	FREESPACE
Type:	built-in function
Format:	n = FREESPACE()
	n = FREESPACE(addr, size)
Description:	return memory to ARexx's internal memory pool

Script example:

```
/* Allocate then free some memory */
say "ARexx memory pool contains" freespace() "bytes."
mem = getspace(1000)
say "Allocated 1000 bytes at address $"c2x(mem)"."
say "Freeing, freespace() returns" freespace(mem,1000)"."
say "ARexx memory pool now contains" freespace() "bytes."
```

Example output:

```
ARexx memory pool contains 960 bytes.
Allocated 1000 bytes at address $07EADF00.
Freeing memory, freespace() returns 2864.
ARexx memory pool now contains 2912 bytes.
```

Discussion: Behind the scenes of an executing script, the ARexx interpreter is constantly allocating and freeing memory. Rather than allocating many small amounts of memory from the system directly, ARexx maintains a sort of 'petty cash fund': it keeps an internal pool of memory, and extends it with larger allocations from the system whenever the pool is exhausted.

GETSPACE and FREESPACE allow a script to obtain memory from and release memory into the internal pool, on those occasions when a temporary buffer with a known address is needed.

When called with no arguments, FREESPACE returns the current size of the internal memory pool. This may vary widely in the course of an executing script, but is not usually of interest unless you are running near the limit of available memory. To actually free memory, the address (as returned by GETSPACE) and size (as given to GETSPACE) of the allocation must be supplied.

In most cases, it is not strictly necessary to call FREESPACE at all, since any memory allocated by GETSPACE will be freed automatically when the script finishes running. Nevertheless, it is probably a good idea to do so, just as it is a good idea to close files explicitly rather than leave it up to ARexx to do for you.

See also: export, import, getspace, storage, allocmem, freemem

Name:	FUZZ
Type:	built-in function
Format:	f = FUZZ()
Description:	return current NUMERIC FUZZ setting

Code example:

```
/* This could fail, depending on the value of FUZZ */
numeric digits 9

/* ...whereas this is guaranteed to work */
numeric digits max(9,fuzz()+1)
```

Discussion: FUZZ returns the current fuzz setting for numeric comparisons, as set with the NUMERIC FUZZ instruction. The value returned will be an integer from 0 to one less than the current significant digits setting. Sometimes it is desirable to save the current fuzz setting before modifying it in order that it can be restored later on. FUZZ may also be used, as in the second example, to validate the argument to the NUMERIC DIGITS instruction.

See also: digits, form

Name:	GETARG
Type:	support function
Format:	arg = GETARG(packet, [whicharg])
Description:	obtain an argument string from a message packet

Script example:

```
/* Ultra-simple command host written in ARexx. Assuming
   it is filed as rexx:SimpleHost.rexx, use it like this:
      Shell> run rx SimpleHost
      Shell> rx "address simple_host quit
*/
hname    = 'SIMPLE_HOST'
shutdown = 0

if openport(hname) then do
   if open('cf','con:50/50/200/20/Simple host') then do
      do until shutdown
         call waitpkt(hname)
         pkt   = getpkt(hname)
         rcode = 0

         if pkt ~= null() then do
            cmd = getarg(pkt)

            if upper(cmd)='QUIT' then
               shutdown = 1
            else do
               say 'Unknown command:' cmd
               rcode = 10
               end

            call reply(pkt,rcode)
            end
         end

      call close('cf')
      end

   call closeport(hname)
   end
```

Discussion: An ARexx script usually gets its arguments (if any) through the ARG function or the PARSE ARG instruction. When an ARexx script calls a command host or a function host, however, the host receives its command (or function) name and arguments in the form of a 'message packet'. Although hosts are normally written in high-performance languages like C or assembler, functions to manipulate message packets are provided in the ARexx support library. With their aid it is possible to write a command host completely in ARexx (although it does not appear that a well-behaved function host can be written with the current version of the library).

In the extremely short and simple command host of the example, message packets are awaited at the message port with the name 'SIMPLE_HOST'. When a packet arrives, its address is assigned to the *pkt* variable. After checking that the packet is valid by comparing it with the null address, we can proceed to examine it with GETARG.

GETARG's first argument is the packet from which we wish to extract information. The second argument specifies which packet argument, from 0 to 15, we want to look at. Packets sent to a command host have only one argument, numbered 0. Since 0 is the default value for the second argument, we can get the information we want with:

```
cmd = getarg(pkt)
```

As mentioned, you can't currently write a function host in ARexx (not straightforwardly, at least—you could always code your own support functions in another language). Since only packets sent via function invocation can have more than one argument (assuming the packets originate with a script), it is unlikely that you will ever need to use the second argument to GETPKT.

See also: getpkt, reply, typepkt, waitpkt, openport, closeport

Name:	GETCLIP
Type:	built-in function
Format:	s = GETCLIP(clip)
Description:	return value string associated with clip name

***Dialog* example:**

```
->'['getclip('charles')']'
[]
->setclip('charles','dickens')
1
->'['getclip('charles')']'
[dickens]
```

Discussion: ARexx maintains a list of strings called 'clips', which is globally available to all scripts. Each entry on the Clip List is known by a unique name; the entry associated with a given name may be retrieved with GETCLIP. If there is no entry on the Clip List with the specified name, GETCLIP returns an empty string.

The Clip List provides a means of making information available to a script that is analogous to the environment variables of AmigaDOS. The ARexx manual suggests using clips in conjunction with the INTERPRET instruction to create named constants. For instance, if there were a clip called 'Constants' that contained these initializations:

```
pi=3.14159265; e=2.71828183; sqrt2=1.41421356
```

they could be incorporated into a script with:

```
interpret getclip('Constants')
```

just as though the initializations had been performed by the script itself.

In another usage, set-up information like data file names and formatting parameters can be communicated through clips to a single script or a suite of related scripts without having to provide the information as command-line or function arguments. A script can assume a default set-up and go with that if the relevant clip has not been initialized.

See also: setclip

Name:	GETPKT
Type:	support function
Format:	pkt = GETPKT(portname)
Description:	pick up a message packet from a message port

Discussion: GETPKT removes a packet waiting at the message port named in its one argument, and returns the address of the packet as a 4-byte string. You use the string—without ever caring about its exact contents—as a means of referring to the packet when dealing with other functions like GETARG. You should check the value returned by GETPKT to make sure it is not the null address '00000000'x.

Packets picked up at a message port with GETPKT must eventually be returned to their sender with the REPLY function. This should be

done without undue delay, as the sender is most likely waiting in enforced idleness for the reply to arrive.

See also: getarg, reply, typepkt, waitpkt, openport, closeport

For a programming example including GETPKT, and an explanation of message packets and their role in some ARexx scripts, see the entry for GETARG.

Name:	GETSPACE
Type:	built-in function
Format:	addr = GETSPACE(size)
Description:	allocate memory in ARexx's internal memory pool

Discussion: GETSPACE is used to allocate a memory area that will not be required after the script that calls it has finished running (since ARexx will then free the memory even if the script has not). GETSPACE cannot fail as such—it will always return a valid address. Instead, if the requested memory allocation cannot be satisfied, ARexx generates error 3, 'No memory available'.

See also: For more information on GETSPACE and FREESPACE, see FREESPACE.

Name:	HASH
Type:	built-in function
Format:	n = HASH(str)
Description:	calculate a hash value for a string

***Dialog* examples:**

```
->hash('A')
65
->hash('AA')
130
->hash('AAAA')
4
```

Discussion: To 'hash' a string means to derive a numeric value from its component characters. That value can be used subsequently for fast look-up of the original string. Hash values are not (ordinarily) expected to be unique for a given string: many strings may hash to the same value in what is known as a 'hash collision'. Computer scientists have, however, put a lot of effort into devising sophisticated hash functions that minimize collisions as far as it is possible to do so.

ARexx's HASH has no such pretensions to sophistication. The value it returns is the sum, mod 256, of the ASCII values of the characters in the argument string.

Name:	IF
Type:	instruction
Format:	IF test [;] THEN [;] instruction [ELSE [;] instruction] (NB: 'instruction' here means either a single instruction ending at the end of a line, or a compound instruction such as a DO-END block or another IF instruction.)
Description:	introduce code to be executed if test expression is True

Script example:

```
/* IF example */
parse value date('e') with day '/' month '/' year

if month=12 & day=25 then
   say "Merry Christmas!"
else if year>93 then do
   say "Okay, now at last the truth can be revealed:"
   say "'Nick Sullivan' is the pseudonym of Navillus Kcin!"
   end
else do
   hours = time('h')

   if hours < 6 then
      say "Up all night worrying about insomnia?"
   else if hours <= 12 then
      say "Good morning!"
   else if hours < 18 then
      say "Good afternoon!"
   else
      say "Good evening!"
   end
```

Discussion: Instructions may come and go, but just about every high-level computer language ever devised has an instruction called IF. A good thing too, for IF (supplemented by younger relatives like WHILE and UNTIL) is what gives computers the power to make decisions based on present conditions, to cope with special cases, to select the correct course of action from a number of alternatives.

Structurally, an IF instruction consists of a boolean test expression sandwiched between the keywords IF and THEN, with a dependent instruction (simple or compound) coming after. The dependent instruction is executed only if the test expression returns True. The instruction may itself be followed by the ELSE keyword and a second

dependent instruction, to be executed only if the test expression returns False.

As in the example above, IF/THEN/ELSE clauses are often linked to form a chain, whose true structure is concealed by the conventional indentation. For further discussion of this see ELSE.

Sometimes, one member of an IF/THEN/ELSE tests for some case merely to exclude it from consideration further down; not to perform an action, in other words, but to avoid one. In ARexx, this means using the special instruction NOP as the dependent instruction for that case. NOP's great talent is that it does exactly nothing. Here is a schematic example to show one sort of logic that calls for NOP:

```
if a
   call handle_a()
else if b
   nop
else if c
   call handle_c()
else
   call handle_default()
```

If it seems odd to use an instruction that has no effect, consider the alternative:

```
if a then
   call handle_a()
else if ~b then do
   if c
      call handle_c()
   else
      call handle_default
   end
```

The effect is identical, and the oddity of NOP is avoided, but the second version is hardly clearer.

See also: else, nop, select, do

Name:	IMPORT
Type:	built-in function
Format:	s = IMPORT(addr,[len])
Description:	return contents of memory as a string

***Dialog* example:**

```
->c2x(import('00000004'x,4))    /* Address of ExecBase */
00000676
```

Discussion: IMPORT reads bytes from system memory and copies them to an ARexx string. The number of bytes to copy may be given as the second argument; the default action is for the copy to terminate when a zero byte is encountered.

See also: export, getspace, freespace, storage, allocmem, freemem

Name:	INDEX
Type:	built-in function
Format:	n = INDEX(str,pat,[start])
Description:	return position of pattern in string

***Dialog* examples:**

```
->index("rubadubadub","ub")
2
->index("rubadubadub","ub",2)
2
->index("rubadubadub","ub",3)
6
->index("rubadubadub","ub",7)
10
->index("rubadubadub","ub",11)
0
```

Discussion: INDEX is used to locate a substring within a string. The number returned is the start character of the pattern within the string, counting the leftmost character as 1. Zero is returned if the pattern is not found. The optional *start* argument specifies the first character within the string at which to begin looking for a match. This allows a string to be scanned iteratively, as in this function to count the number of times a specified character (or larger substring, for that matter) occurs in a string:

```
/* CharCount(s, c): count number of times c occurs in s */
CharCount: procedure
   parse arg s, c
   count = 0
   pos   = index(s, c)

   do while pos > 0
      count = count + 1
      pos   = index(s, c, pos + 1)
      end

   return count
```

See also: pos, lastpos, find

Name:	INSERT
Type:	built-in function
Format:	s = insert(istr,str,[start],[len],[pad])
Description:	INSERT istr into str at start position

Dialog examples:

```
->insert("tat","too")
tattoo
->insert("mat","too",2)
tomato
->insert("","Hell, Johnny!",4,20,"o")
Hellooooooooooooooooooo, Johnny!
```

Discussion: The first argument string is inserted into the second. By default, the insertion point is the beginning of the second string, but the point can be specified as the *start* argument, the insertion taking place after the given character position. Giving zero as the position produces the default behavior; giving a value greater than the length of *str* as the insert position causes *str* to be padded as necessary with spaces (the default) or with the *pad* character. The inserted string will be truncated or padded to the *len*, the default being the length *istr*.

INSERT is useful in the kind of string editing work exemplified by this function, which performs a search and replace operation within a given string:

```
/* replace - replace srch text with repl text in text
      ->replace("astrologer","log","nom")
      astronomer
*/

replace: procedure

   parse arg text, srch, repl

   slen = length(srch)
   tlen = length(text)

   do until tlen = 0

      tlen = lastpos(srch,text,tlen)

      if tlen ~= 0 then do
         text = insert(repl,delstr(text,tlen,slen),tlen - 1)
         tlen = tlen - 1
         end
      end

   return text
```

See also: delstr, overlay

Name:	INTERPRET
Type:	instruction
Format:	INTERPRET [expr]
Description:	execute ARexx instructions contained in a string

Script example:

```
/* INTERPRET example - arg specifies string justification */
jtype   = word('LEFT CENTER RIGHT', max(min(arg(1),3),1))
planets = "Mercury Venus Earth Mars Jupiter Saturn",
          "Uranus Neptune Pluto"

do i=1 to 9
   interpret "say" jtype"(word(planets,i),60)"
   end
```

Discussion:

INTERPRET is a uniquely flexible command that in effect allows your script to write scripts of its own. The expression you give to INTERPRET is evaluated, yielding a string. Then that string is interpreted as ARexx program code, and executed in the usual way.

In the example above, a numeric argument between 1 and 3 selects one of the strings 'LEFT', 'CENTER' and 'RIGHT', each of which is the name of an ARexx built-in string function taking two arguments: a string to be placed within a field of blanks, and a number giving the width of the field.

Let's suppose the argument to the script is 2, so the 'jtype' variable is set to 'CENTER'. Evaluating the string argument to INTERPRET now yields:

```
say CENTER(word(planets,i),60)
```

INTERPRET executes this dynamically-built line to create the script's formatted output. Compare this with an obvious alternative:

```
do i=1 to 9
   select
      when jtype=1 then say left(word(planets,i),60)
      when jtype=2 then say center(word(planets,i),60)
      when jtype=3 then say right(word(planets,i),60)
      end
   end
```

A classic use of INTERPRET is the script *calc.rexx*, which goes something like this:

```
/* calc.rexx */
interpret say arg(1)
```

Can such a trivial-looking script actually do anything? Look:

```
Shell> rx calc 307*9
2763
Shell> rx calc "24 bit-planes allow" 2**24 "colors!"
24 bit-planes allow 16777216 colors!
```

The simple *calc.rexx* script is the basis for the Dialog script used in this book, and for the much larger version of Dialog on the accompanying disk. It is almost meaningless to speak of a few 'typical' uses for INTERPRET; with a little imagination its uses are too many to enumerate.

In the examples so far, the string given to INTERPRET has contained only a single instruction. Multiple instructions can be separated with semicolons:

```
interpret "do i=1 to 5; say i*i; end"
```

When the same code is recast the following way, you can see that the principle could be extended much further if desired:

```
exec = 'do i=1 to 5;',
          'say i*i;',
          'end'

interpret exec
```

Indeed, it would be perfectly possible—though generally quite pointless—to read an entire ARexx script file from disk into a string, and pass that string to INTERPRET to be executed. The ability to read a specific set of instructions from a disk file, such as a series of assignments, or a set of formatting expressions, may be very useful in some situations.

Executing instructions via INTERPRET is not quite the same in all respects as if the same instructions were simply written into your script. INTERPRET stands as a kind of 'island' within your script, not fully connected with it. For instance, you cannot exit a DO loop in your main script with:

```
interpret 'leave'
```

Nor can you use a label defined within the interpreted string, though you can jump from within the interpreted string to a label outside it with SIGNAL, or call a function defined outside the interpreted string.

The BREAK instruction can be used to exit from the middle of an interpreted string; again, the BREAK has no impact on a DO-END block in which the INTERPRET instruction may be contained.

Name:	ITERATE
Type:	instruction
Format:	ITERATE [var]
Description:	skip to the end of the current iterative loop

Script example:

```
/* ITERATE example - show all unique pairs of 2 numbers 1-5,
   without regard for ordering.
*/
do i=1 to 5
   do j=1 to 5
      say i',' j
      if i=j then
         iterate i
      end j
   end i
```

Discussion: Like BREAK and LEAVE, ITERATE changes the normal flow of control within an 'iterative' DO: any DO loop, in other words.

Whereas BREAK and LEAVE terminate the entire loop to which they apply, ITERATE terminates only the current iteration, so that control passes immediately to the END statement, thence back to the DO itself. If the ITERATE is contained within a loop controlled by an index variable, and that variable's name is given in the instruction, the ITERATE applies to the named loop even if it is not the innermost (as in the example).

See also: do, leave, break

Name:	LASTPOS
Type:	built-in function
Format:	n = LASTPOS(pat,str,[start])
Description:	return start position of pat in str, searching backwards

***Dialog* examples:**

```
->lastpos("ub","rubadubadub")
10
->lastpos("ub","rubadubadub",10)
10
->lastpos("ub","rubadubadub",9)
6
->lastpos("ub","rubadubadub",5)
```

```
2
->lastpos("ub","rubadubadub",1)
0
```

Discussion: LASTPOS is used to locate a substring within a string, but unlike INDEX and POS, begins its search at the end of the string. The number returned is the start character of the pattern within the string, counting the leftmost character as 1. Zero is returned if the pattern is not found. The optional third argument specifies the first character within the string (working from right to left) at which to begin looking for a match. This allows a string to be scanned iteratively (for an example, see INDEX). One application for the backwards scanning capability of LASTPOS is parsing an AmigaDOS path name into a directory and a file. This function finds the file portion of a path name by scanning backwards first for a slash and then (if that fails) for a colon:

```
/* GetFileName(path): return file portion of path name

   ->GetFileName("sys:utilities/Clock.info")
   Clock.info
*/
GetFileName: procedure
   path = arg(1)
   n = lastpos("/",path)

   if n = 0 then
      n = lastpos(":",path)

   return substr(path, n + 1)
```

See also: index, pos, find

Name:	LEAVE
Type:	instruction
Format:	LEAVE [var]
Description:	break out of current iterative loop

Script example:

```
/* TextCreate.rexx - a tiny text editor with editing of
   current line only. Usage: rx textedit <newfilename>
*/
text. = ''
options prompt ">"
say "Enter text on following lines; end with /X:"

do count=1
   do until gotline
      parse pull line 1 slash 2 cmd 3 cmdtail
      gotline = slash ~= '/'
```

```
      if ~gotline then do
         select
            when cmd = 'X' then
               leave count
            otherwise
               say "*** Unrecognized command!"
            end
         end
      else
         text.count = line
      end
   end

if open('file',arg(1),'w') then do
   do i=1 to count - 1
      call writeln('file',text.i)
      end
   call close('file')
   end
```

Discussion: Situations can arise during the processing of an iterative DO (a loop, in other words) that require exiting the loop without waiting for one of the normal exit conditions to be satisfied. The LEAVE instruction provides the necessary means of emergency escape. When the name of a loop index variable is given in the instruction (like the name 'count' in the example), LEAVE causes an exit from the loop controlled by that variable; otherwise only the innermost active loop is affected.

Some loops—like the outer loop in the example—have no exit condition, and aren't intended to terminate until some event is detected (in the example, the event is the user entering a line beginning with '/X'). Especially if the test for the triggering event is not at the outermost level, but is nested several control structures deep within the loop, as in the example, it may be awkward and verbose to arrange the exit without LEAVE, requiring additional boolean variables and redundant tests.

Like ITERATE and BREAK, LEAVE should only be used where it will significantly simplify the code. All else being equal, it is better for script readability to handle loop exits in the normal way, with controlling expressions at the top of the loop, than to branch away from the middle of the loop unexpectedly.

See also: do, iterate, break

Name:	LEFT
Type:	built-in function
Format:	s = LEFT(str,count,[pad])
Description:	extract leftmost 'count' characters of str

***Dialog* examples:**

```
->left("Gorgonzola",6)
Gorgon
->left("Chapter 4",40)right("Page 23", 8)
Chapter 4                              Page 23
->left("Microwave oven",42,".")" $299.95"
Microwave oven.............................. $299.95
->left("Color TV with remote",42,".")" $399.95"
Color TV with remote....................... $399.95
```

Discussion: LEFT extracts and returns the given number of characters from the left of the argument string. If necessary to make up the requested count, the argument string is padded on the right with spaces (the default) or the *pad* character. Creating a column of left-justified text to a standard line length, and with any desired 'fill' character, is easily done, as the examples above demonstrate.

See also: center, right

Name:	LENGTH
Type:	built-in function
Format:	n = LENGTH(str)
Description:	return length of string in characters

***Dialog* examples:**

```
->length("")
0
->length("antidisestablishmentarianism")
28
->length(copies("abc",100))
300
->length(1/9) /* 0.111111111 */
11
->length('01'x)
1
```

Discussion: LENGTH returns the number of characters in the argument string. A representative use—among thousands—of this simple but important function would be to determine the length of an underline for a document title:

```
/* Center and underline a title */
title = "The Goldbach Conjecture - A Proof"
say center(title, 60)
say center(copies("-",length(title)),60)
```

Name:	LINES
Type:	built-in function
Format:	n = LINES([file])
Description:	return number of lines queued for interactive stream

Code examples:

```
/* 1: Using LINES */
queue "dir >ram:foo ram:"
queue "type ram:foo"
say lines(stdin)        /* output: 2 */
/* The queued lines now execute */

/* 2: Eliminating type-ahead */
do while lines()>0
   pull
   end
```

Discussion: The PUSH and QUEUE instructions provide a means whereby interactive input can be simulated by entering lines into an interactive input stream just as though they had been typed. Additionally, input lines can be typed ahead manually into a console while previously entered lines are being processed. LINES allows a script to determine how many lines of input, if any, are queued for a particular interactive file (by default, STDIN).

However, one important proviso severely limits the general applicability of LINES: unless the installed console handler provides the necessary support for the function, it will always return zero regardless of how many lines may actually be queued. At the present time, the only console handler that provides this support is ConMan, written by Bill Hawes (the author of ARexx). The standard AmigaDOS console handler does not, so LINES will not work on off-the-shelf Amigas.

See also: push, queue

Name:	MAKEDIR
Type:	support function
Format:	bool = MAKEDIR(dirname)
Description:	create a directory of the given name

***Dialog* example:**

```
->showdir('ram:','d')
env clipboards t
->makedir('ram:newdir')
1
->showdir('ram:','d')
newdir env clipboards t
->makedir('ram:newdir') /* Under Workbench 1.3, returns 1 */
0
->delete('ram:newdir')
1
```

Discussion: MAKEDIR creates a directory of the given name, if it is possible to do so, and returns a boolean result reflecting the success of the operation. There are several reasons MAKEDIR might fail:

- The directory already exists (Workbench 2.0 only)
- The pathname is invalid
- The volume is write-protected

If you are reasonably sure when you call MAKEDIR that the only possible cause of failure is that the directory exists, you need not worry about the returned value—either way you'll get what you want. If you aren't sure—because you don't know whether the volume is write-protected perhaps—you could use the substitute version of MAKEDIR below instead of calling the support library routine directly. It returns 1 either if the directory already exists or could be created:

```
/* MakeDir - If a directory under the name already exists, or
   can be created, return 1, otherwise return 0. Though this
   function works correctly under Workbench 1.3, it has the
   same effect as the existing MAKEDIR; hence it is useful only
   under 2.0.
*/
MakeDir: procedure
   ds = statef(arg(1))

   if ds='' then
      result = 'makedir'(arg(1))
   else
      result = left(ds,3) = 'DIR'

   return result
```

See also: delete, rename

Name:	MAX
Type:	built-in function
Format:	n = MAX(n1, n2 [, n3 ...])
Description:	find the largest of a set of numbers

***Dialog* examples:**

```
->max(1,2)
2
->max(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
15
->max(0,-1000)
0
->max(min(15,20),10)
15
```

Discussion: MAX takes from 2 to 15 numeric arguments and returns the largest of them. An idiomatic use of MAX and its counterpart MIN is demonstrated in the fourth example, which constrains a given value (here a constant 15 but typically a variable) to an allowed range (here 10 through 20 inclusive).

See also: min

Name:	MIN
Type:	built-in function
Format:	n = MIN(n1, n2 [, n3 ...])
Description:	find the smallest of a set of numbers

***Dialog* examples:**

```
->min(1,2)
1
->min(1,2,3,4,5,6,7,8,9,10,11,12,13,14,15)
1
->min(0,-1000)
-1000
```

Discussion: MIN takes from 2 to 15 numeric arguments and returns the smallest of them.

See also: max

Name:	NEXT
Type:	support function
Format:	value = NEXT(addr,[offset])
Description:	return the 4-byte value stored at addr+offset

Script example:

```
/* List currently-loaded fonts */
call forbid() /* Don't multitask while scanning shared data */

/* Find first node on system font list. The list header starts
   140 bytes into the graphics library data structure.
*/
gfxbase = showlist('l','graphics.library',,'a')
font    = next(gfxbase,140)      /* First node on font list */

/* Scan list, gathering information into compound variables
   name and size. The list scan ends when a node is found
   whose successor node address is 0.
*/
do i=1 while next(font)~=null()
   name.i = import(next(font,10))            /* font name  */
   size.i = c2d(import(offset(font,20),2))   /* font ysize */
   font   = next(font)                       /* next node  */
   end

call permit()  /* Finished looking at list, ok to multitask */

/* remove ".font" from font names, and display tidily */
do j=1 to i-1
  say left(left(name.j, length(name.j) - 5),15) right(size.j,2)
  end
```

Discussion:

Essentially everything the Amiga's operating system needs to know about the current state of the computer is stored in an extensive set of interlocking 'data structures' that are ultimately anchored in the 'ExecBase' structure whose address is stored in the 4 bytes beginning at location 4 in memory. If you needed to, you would find ExecBase's address on your system with:

```
->c2x(import('00000004'x,4))
```

Many information structures are tied together in the form of 'linked lists'. Many of the more important lists in the system can be perused in comfort with the aid of the support library's SHOWLIST function. SHOWLIST only knows about certain standard lists, however: it does not support linked lists in general. That job falls to NEXT.

The items on a linked list are termed 'nodes'. The usual type of node is laid out in memory like this, beginning at some address we'll call A:

Address	Contents
A	The address of the next node in the list
A+4	The address of the previous node in the list
A+8	A number identifying the type of node (not always used)
A+9	A priority number for this node (often unused)
A+10	The address where the node name is stored (not always used)
A+14...	The actual information content of the node

If the ARexx variable 'A' contains the address of a node in the standard 4-byte format, then the address of the subsequent node is given by:

```
nextnode = next(A)
```

The default for NEXT's second argument is 0, so this is short for:

```
nextnode = next(A,0)
```

The NEXT function adds its second argument (or 0) to the address given as its first argument, and returns the contents of the 4 bytes beginning at the resulting address. Compare this with the action of OFFSET, which merely computes and returns the address itself, rather than what is stored there. In fact, NEXT is really a mere convenience: you could easily code it yourself using IMPORT and OFFSET:

```
next: procedure
   return import(offset(arg(1), 0 || arg(2)), 4)
```

To find the previous item in a list, you look 4 bytes into the node, and extract the address from there:

```
prevnode = next(A,4)
```

Similarly, to find the address where the node name is stored, go in 10 bytes from the start of the node:

```
say import(next(A,10))
```

Unless you're sure that the name field of the node is actually in use, of course, you should test the value returned by the call to NEXT before passing it on to IMPORT, making sure that it's non-zero.

How can you tell when chaining along a list, either forwards or backwards, when you've come to the last node? You might reasonably

expect that the last node is the one whose successor's address (or predecessor's, going backwards) is 0, and it is indeed possible to organize a list that way. Most Amiga system lists, however, are set up so that the last node is the one whose successor's successor (or predecessor's predecessor) is 0. Failing to handle this list organization properly is a common trap for the unwary Amiga programmer.

The example script above scans and displays the list of currently open fonts that is maintained by the part of the operating system called the Graphics Library. Because the font list is information shared by all the tasks currently active in the system, it is theoretically possible for the activities of another task to result in it being modified even while our task is attempting to chain along it. This could be disastrous, so the example script shuts off multitasking with a call to FORBID before starting to look at the list, and enables it again afterwards with PERMIT. This is a standard and necessary precaution when handling any system list.

See also: import, null, offset, showlist

Name:	NOP
Type:	instruction
Format:	NOP
Description:	do nothing—a dummy instruction

Script example:

```
/* FIXCASE(): s = fixcase(s1, pattern)
   Three case structures are recognized: all caps, initial cap
   and lower. The case structure of the upper case string s1 is
   set to match that of the pattern string, which is
   categorized on the basis of its first two characters. This
   could be used in a spelling correction utility.

      DIOGENES <- fixcase('DIOGENES','DIOJENES')
      Paraguay <- fixcase('PARAGUAY','Parraguay')
      syzygy   <- fixcase('SYZYGY','syzygy')
*/
fixcase: procedure
   parse arg s, c1 2 c2

   if datatype(c1,'u') then
      if datatype(c2,'l') then
         s = left(s,1)lower(substr(s,2))
      else
         nop
   else
      s = lower(s)

   return s
```

```
/* LOWER(): s = lower(s1) - convert s1 to lower case
*/
lower: procedure
   return translate(arg(1),xrange('a','z'),xrange('A','Z'))
```

Discussion: The NOP (for 'No OPeration') instruction is occasionally needed to provide a place-holding dependent instruction for IF/THEN, ELSE and WHEN/THEN. (In fact it is legal anywhere in a script, but is very rarely useful except in the mentioned contexts.)

See also: if, else, select

Name:	NULL
Type:	support function
Format:	'00000000'x = NULL()
Description:	return a 4-byte string corresponding to a null address

***Dialog* example:**

```
->c2x(null())
00000000
```

Discussion: NULL might seem a strange function to put in a library, especially since it can always be replaced by the hex string '00000000'x without affecting any script that uses it. The null address is very often encountered, however, and it enhances the readability of scripts to refer to it by name rather than as a literal string.

See also: See NEXT for an instance of the very common use of the null address (and the NULL function) as an end marker in list chaining.

Name:	NUMERIC
Type:	instruction
Format:	NUMERIC DIGITS num
	NUMERIC FUZZ num
	NUMERIC FORM SCIENTIFIC
	NUMERIC FORM ENGINEERING
Description:	set numeric calculation and display options

Script example:

```
/* NUMERIC example */
pi = 3.14159265358979323846264338327950288419716 9
say "PI:" pi; say

numeric digits 14; call showsettings()
say "PI:" 0+pi                              /* 3.1415926535898 */
```

```
numeric digits 9; call showsettings()
say "PI:" 0+pi; say                          /* 3.14159          */
say "PI = 355/113?" (+pi = 355/113)          /* 0 (i.e. False)   */
numeric fuzz 3;   call showsettings()
say "PI = 355/113?" (+pi = 355/113); say /* 1 (i.e. True)    */
numeric digits 6; call showsettings()
say "PI * 1e8:" pi * 1e8                     /* 3.14159E+8       */
numeric form engineering; call showsettings()
say "PI * 1e8:" pi * 1e8                     /* 314.159E+6       */
exit

showsettings: procedure
   c1 = '1b'x"[33m"; c3 = '1b'x"[31m" /* colors 1 and 3 */
   say c1"[DIGITS="digits()", FUZZ="fuzz()", FORM="form()"]"c3
   return
```

Discussion: The three NUMERIC options govern the characteristics of ARexx numbers:

- DIGITS: The number of significant figures in numeric results. By default, 9 significant figures are used. NUMERIC DIGITS can vary this setting from 1 through 14. Values that are too big or too small to be expressed with the required number of digits are displayed in one of the two forms of exponential notation. Numbers within that range, once they have been converted to numeric form, are truncated (with rounding) to the specified precision. Numeric strings can be stored with arbitrary precision, as PI is stored in the example script. As soon as the value is involved in a calculation, though, it must be converted to the internal numeric form, and hence rounded to the current digits setting.

  It is an error to try to set DIGITS to less than or equal the current FUZZ setting, or greater than 14.

- FUZZ: It is possible to incorporate a 'fuzz factor' into numeric comparisons such that two numbers will be seen as equal even though they differ in their less significant digits. The FUZZ value specifies how many digits of precision should be ignored when comparisons are performed. In order to force a value into numeric mode so that numeric comparisons will yield predictable results, it may be necessary to involve them in a dummy operation, such as preceding them with a plus sign, or adding zero. For instance:

```
/* Numeric comparisons */
pi = 3.141592654
numeric digits 6

say  pi = 3.14159      /* output: 0 */
say +pi = 3.14159      /* output: 1 */
```

Under the version of ARexx available at the time this book was written (1.15), the effect of FUZZ on numeric comparisons is not always fully consistent; it may be best as a rule to allow an extra digit of 'fuzziness'. For example, let us continue now with these lines:

```
numeric fuzz 1
say +pi = 3.1416      /* output: 1   */
numeric fuzz 2
say +pi = 3.142       /* output: 1   */
numeric fuzz 3
say +pi = 3.14        /* output: 0?? */
numeric fuzz 4
say +pi = 3.14        /* output: 1   */
say +pi = 3.1         /* output: 0?? */
```

In view of these unexpected results, due caution should be exercised in using FUZZ.

It is an error to try to set FUZZ to greater than or equal the current DIGITS setting, or less than 0.

- FORM: There are two variants of the exponential notation used for numbers that are too large are too small to be written out in full under the current level of precision. In SCIENTIFIC form, the mantissa (the first part) of the number has only one digit before the decimal point. In ENGINEERING form, the exponent (the second part) of the number is always divisible by three, so the mantissa may have from 1 to 3 digits to the left of the decimal point. SCIENTIFIC notation is the default, and will not need to be changed for most scripts. It is (only just) worth noting that the ENGINEERING form does not display properly at very low precision (DIGITS settings of 3 or less) but this is hardly likely to be a problem in real scripts.

  The current values of DIGITS, FUZZ and FORM can be determined with the functions of those names in the built-in library.

See also: digits, fuzz, form

Name:	OFFSET
Type:	support function
Format:	addr = OFFSET(addr,amount)
Description:	return the address addr+amount

***Dialog* examples:**

```
->c2x(offset(null(),4))
00000004
->c2x(offset('01010101'x,c2d('20202020'x)))
21212121
```

Discussion: The arguments to OFFSET are a 4-byte address and a numeric value. The return value is the 4-byte address obtained by adding the arguments together. The function is equivalent to the expression:

```
d2c(c2d(addr)+amount)
```

combined with a check to make sure that the address argument is in the proper format.

The point of OFFSET is to handle a common requirement when dealing with Amiga data structures: given the address of a structure, and the offset of a field within it, calculate the address of the field. For instance, if we know that an Intuition Window structure is located at memory address $07f374e0, and that the offset of the BorderLeft field within a Window structure is 54 bytes, it follows that the address of the BorderLeft field is:

```
offset('07f374e0'x,54)
```

See also: next

Name:	OPEN
Type:	built-in function
Format:	bool = OPEN(file, name, [mode])
Description:	open a file called 'name' in given mode

***Dialog* examples:**

```
->open('w_file','ram:testfile','w')
1
->open('r_file','ram:testfile','r')
0
->writeln('w_file','A few well-chosen words')
24
->close('w_file')
1
->open('a_file','ram:testfile','a')
```

```
1
->open('r_file','ram:testfile')
1
->writeln('a_file','A terse supplementary phrase')
29
->readln('r_file')
A few well-chosen words
->readln('r_file')
A terse supplementary phrase
->close('a_file')
1
->close('r_file')
1
```

Discussion: The first argument to OPEN is an identification string by which the file can be referenced in subsequent use (e.g. with READLN, WRITELN, CLOSE). The identification string must not be in use for any other file. OPEN's second argument is a standard AmigaDOS file name (including the device and directory portions, as needed).

A file may be opened in any of 3 'access modes', as specified by the optional third argument: 'r' ('Read', the default), 'a' ('Append'), or 'w' (Write'). Read and Append modes are essentially identical, except that in Append mode the initial file position is the end, rather than the beginning of the file, in preparation for adding new material. Read/Append mode differs from Write mode in two fundamental ways:

1) In Write mode, the opened file is created anew, rather than merely accessed, erasing any existing file of the same name. One way a call to OPEN in Write mode can fail is if the file's 'delete' attribute has been cleared with the AmigaDOS Protect command. Read/Append mode requires an existing file of the given name to be available, and an OPEN in this mode will fail if it is not.

2) Files opened in Write mode guarantee the caller exclusive access to the file until the file is closed. Meanwhile, attempts to open the same file, both from within ARexx and anywhere else, will fail (see the second call to OPEN in the examples above). In Read/Append mode, on the other hand, the file may be open to multiple accessors simultaneously (see the third and fourth OPEN calls in the examples).

The boolean value returned by OPEN reflects whether the file open was successful.

See also: close

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	OPENPORT
Type:	support function
Format:	bool = OPENPORT(portname)
Description:	open a public message port with the given name

***Dialog* examples:**

```
->openport('bookport')
1
->openport('bookport')
0
->show('i') /* list 'internal' message ports */
bookport
->closeport('bookport')
1
```

Discussion: Message ports are the rendezvous points for passing information (in the form of 'messages') between tasks in the Amiga system. Some message ports, termed 'public' ports, are named, and may be looked up in the system message ports list.

Message ports are of pivotal importance to ARexx. Much of what gives the language its special role on the Amiga is its ability to mediate communication among tasks. The names of function hosts and command hosts are actually the names of message ports they have opened in order to receive messages from ARexx.

OPENPORT tries to create a message port using the given name, and returns a boolean value reflecting its success. The only common reason for failure is that the port already exists, as in the second example above.

Message ports are a managed resource for ARexx scripts. Though it is preferable that each port opened by a script is also explicitly closed by it, ARexx will in any case close the port when the script has finished running. All named message ports in the system, regardless of who created them or why, are linked into the system message ports list; you can see the names on the list with:

```
->show('p',,'0a'x)
```

(The third argument in this call inserts a linefeed between each entry; it helps eliminate any confusion arising from port names that contain embedded spaces.)

Message ports created by a particular ARexx script are known as 'internal' ports. You can see a list of them with:

```
->show('i',,'0a'x)
```

...but unlike the other list, this one is usually empty.

See also: closeport, getarg, getpkt, reply, typepkt, waitpkt

The entry for the GETARG function in this Reference Section contains a programming example that creates a very simple command host. Refer to it for further insight as to how and why message ports are used.

Name:	OPTIONS
Type:	instruction
Format:	OPTIONS [NO] RESULTS
	OPTIONS [NO] CACHE
	OPTIONS PROMPT [expr]
	OPTIONS FAILAT expr
	OPTIONS
Description:	set script options

Script example:

```
/* OPTIONS example - uses PARSE SOURCE to find
   its own path name, and invokes itself.
*/
parse source src rr call resolved ext host
arg mode

if rr then
   return "Here's a result!"
else if mode = 'CHILD' then
   return 20

options results
address rexx resolved
say "Result string returned: '"result"'"

options no results
options failat 30
address rexx resolved 'CHILD'
say "RC is" rc"; failat is set to 30."

options failat 15
address rexx resolved 'CHILD'
say "RC is" rc"; failat is set to 15."
```

Example output:

```
Result string returned: 'Here's a result!'
RC is 20; failat is set to 30.
 22 *-* address rexx resolved 'CHILD';
+++ Command returned 20
RC is 20; failat is set to 15.
```

Discussion: The effect of OPTIONS with no sub-keyword is to restore all OPTIONS settings to their default states. The defaults are:

Option	Default value
RESULTS	off
CACHE	on
PROMPT	"" (empty string)
FAILAT	caller's FAILAT level, generally 10

OPTIONS RESULTS determines whether commands invoked by the script will be asked by ARexx to return a result string. By default this option is off, so the line:

```
OPTIONS RESULTS
```

must be given before invoking a command host from which information is required. Command hosts embedded in application programs are able to return either a result (if requested) or an error code, in such a way that the ARexx interpreter can distinguish between them, assigning results to the RESULT variable and errors to the RC variable.

When an ARexx script is invoked as a command (like the example script above, which invokes itself), the situation is a bit different: it cannot specify to ARexx that the value it returns (with RETURN or EXIT) will be used in one way or the other. That is determined only by the calling script. A returned value will be placed in its RESULT variable if OPTIONS RESULTS has been used, but otherwise becomes the value of RC, and is treated as an error. If OPTIONS RESULTS was not used, and the called script nonetheless attempts to return a non-numeric string, that in itself counts as an error with severity level 10.

The invoked script can, however, use PARSE SOURCE to find out whether a result was requested. If not, the script should not try to return a result. If a result was requested, the invoked script has no way to return an error code.

To switch off OPTIONS RESULTS after it has been switched on, the NO keyword is used:

```
OPTIONS NO RESULTS
```

OPTIONS CACHE is 'on' by default. Its action is to activate what ARexx documentation describes as 'an internal statement-caching scheme'. Although it is possible to turn off this option with:

```
OPTIONS NO CACHE
```

we know of no reason for doing so. When the cache is active, loops in ARexx scripts run considerably faster than when it is not active, and there are apparently no adverse consequences in other contexts.

OPTIONS PROMPT lets you set the prompt string to be used in the PULL, PARSE PULL and PARSE EXTERNAL instructions. If you don't specify a string, the prompt is reset to the empty string, which is the default.

OPTIONS FAILAT must be followed by an expression specifying a 'failure level' for the reporting of errors returned by commands called from the script. If the return code (reflected in the RC variable) from a command is greater than or equal to the current value of the FAILAT variable, either or both of the following may happen:

1) If the 'normal' mode of error tracing is active (as it is unless you have specifically turned tracing off), the error will be reported in the trace output (from RX, this is usually the CLI window) in a form resembling:

```
  23 *-* 'ram:test'
+++ Command returned 20
```

2) If you are trapping command failures with the SIGNAL FAILURE trap, control within your script will be transferred to the 'FAILURE:' label.

If you have turned tracing off, and do not have a trap set, you will not be informed of command errors.

The default failure level for a script originates with the AmigaDOS failure level set for the parent 'process', which is normally 10. If you give the AmigaDOS command:

```
Shell> Failat 20
```

then invoke a script with RX, that script's default failure level will be 20.

Name:	OTHERWISE
Type:	instruction
Format:	OTHERWISE [;] [instruction(s)]
Description:	introduce default case for SELECT control structure

Script example:

```
/* OTHERWISE example - analyze.rexx
   Usage: rx analyze <file name>
*/
if open('af',arg(1)) then do
   h = readch('af',100)
   call close('af')

   lw0 = left(h,4)     /* first 'long word' in file */
   lw2 = substr(h,9,4)/* third 'long word' in file */

   /* all 'printable' characters and linefeed */
   asc = xrange(' ','~')xrange('a0'x,'ff'x)'0a'x
   select
      when lw0 = 'FORM'          then ft='IFF ('lw2')'
      when lw0 = '000003f3'x     then ft='Executable program'
      when substr(h,3,5)='-lh1-' then ft='LHARC archive'
      when length(compress(h,asc))<10 then ft='ASCII text'
      otherwise
         ft = "Unknown"
      end
   say "Probable file type:" ft
   end
else
   say "Unable to open file '"arg(1)"'."
```

Discussion:

OTHERWISE provides a default case for the SELECT instruction, when the conditionals in all the WHEN instructions governed by the SELECT have been evaluated and rejected. It is similar to the final ELSE at the end of an IF-ELSE chain, the one that handles the situations not picked up by previous tests.

Sometimes a SELECT will be written such that the WHEN conditions collectively exhaust all alternatives; in that case, no default is possible. When a default *is* wanted, though, it is compulsory: an OTHERWISE *must* be provided. The lack will be detected only when and if all the WHEN cases evaluate to False, which in extreme instances might not happen till long after a script is written and apparently well-tested. It is therefore a good idea to get into the habit of automatically writing in an OTHERWISE for every SELECT you code.

Any number of instructions, including zero, may be put under an OTHERWISE without a NOP or a DO-END block being required.

See also: select, when

Name:	OVERLAY
Type:	built-in function
Format:	s = OVERLAY(new, old, [start], [len], [pad])
Description:	overlay *new* on *old*, from *start*

***Dialog* examples:**

```
->overlay("White ","Real swine")
White wine
->overlay("ad","Sally dressing",4)
Salad dressing
->overlay("","Give me my walking cane!",12,4,"*")
Give me my ****ing cane!
```

Discussion: The first argument string is overlaid upon the second. By default, the starting point is the beginning of the second string, but the point can be specified as the *start* argument, the overlay beginning at the given character position. Giving 1 as the position produces the default behavior; giving a value greater than the length of *old* as the overlay position causes *old* to be padded as necessary with spaces (the default) or with the given *pad* character. The *new* string will be truncated or padded to the number of characters given as *len*, the default being the length of the overlay string itself.

One use for OVERLAY might be to update information in a database record containing fixed length fields. For example:

```
rec = "1/4-inch washers (1000)     $  3.95 VK-10372"
say overlay(right("4.15",6), rec, 30)
```

which changes only the price information in this inventory record, and leaves the description and part number undisturbed:

```
1/4-inch washers (1000)     $  4.15 VK-10372
```

See also: delstr, insert

Name:	PARSE
Type:	instruction
Format:	PARSE [UPPER] source [template] [,template]

Source name	Source string contents
ARG	Arguments to script or function
EXTERNAL	Input via STDERR file
NUMERIC	NUMERIC options: *digits fuzz form*
PULL	Input via STDIN file
SOURCE	*type result called resolved ext host*
VALUE expr WITH	Result of expression
VAR varname	Contents of varname
VERSION	*version cpu mpu video freq*

Description: split input string(s) into substrings

Script example:

```
/* PARSE example - parse.rexx */
arg src
upper = ''

if word(src,1)='UPPER' then do
   upper = 'UPPER'
   src   = subword(src,2)
   end

if word(src,1)='EXTERNAL' then
   open('STDERR','CON:0/10/500/50/STDERR - for PARSE EXTERNAL')
else if word(src,1)='ARG' then
   src = 'ARG'

testvar ="This is a sample variable to examine with PARSE VAR."

options prompt ">"
interpret "parse" upper src text

do i=1 to words(text)
   say i':' word(text,i)
   end
```

Example input (output is too lengthy to list):

```
Shell> rx parse arg A number of arguments
Shell> rx parse external    /* then type something in window */
Shell> rx parse numeric
Shell> rx parse pull        /* then type something in console */
Shell> rx parse source
Shell> rx parse value .61803399 * 1.61803399 with
Shell> rx parse var testvar
Shell> rx parse version
```

Also try using 'upper' as the first argument to each of these commands, as in:

```
Shell> rx parse upper external
```

Discussion: The example script lets you test the various 'source' options to PARSE. Each source option supplies a different type of 'parse string' to be parsed according to the given template. Because templates are discussed in detail in Chapter 10 (Parsing and String Handling), we describe them only briefly here, then discuss each of the available source options in turn.

Templates A PARSE template is an arbitrary mixture of 'targets' and 'markers'. A 'target' is normally the name of a variable in which a portion of the parse string should be stored. A 'marker' is a specification, usually in the form of a number or string, of where the parse string should be split.

During parsing, an internal value referred to as the 'scan position', which begins with 1 at the left end of the string, is updated as markers are encountered. The part of the parse string assigned to a target is that which is bounded by the scan positions determined by the markers before and after the target in the template.

A target with another target neighboring on the right and only a space between is parsed by tokenization', being assigned the next available word in the parse string. The final target in a template is assigned the balance of the parse string, including perhaps a leading space if the previous target was parsed by tokenization. Unused targets in a template (unused because the parse string was exhausted by earlier targets) are assigned the null string.

A period may be used anywhere in the template as a special target called a 'place-holder'. This acts like a normal target in its effect on the scanning of the parse string, but the portion of the parse string that it matches is not stored. Using the place-holder as the last target of a template forces the previous target (if any) to be parsed by tokenization, which otherwise would not happen because it would have no other target neighboring on the right.

Apart from the implied markers of tokenization, 3 types of marker are used in PARSE templates:

1) A literal string (in quotes), or the value of a variable whose name is given in parentheses, is a 'pattern marker'. The effect of a pattern marker is to set the scan position to the position where the given pattern next occurs in the parse string (or to the end of the string if it does not occur at all). Other types of markers do not modify the parse string, but pattern markers remove the text they match from the parse string, thereby changing its length and possibly affecting the parsing of the string in the remainder of the template.

2) A literal number, or the numeric value of a variable whose name is preceded by an equals sign, directly specifies a new value for the scan position.

3) A number preceded by a plus sign or a minus sign is a 'relative marker', which adjusts the scan position to the right (positive) or the left (negative). Either a literal number or a numeric variable can be used.

Multiple templates, separated by commas, can be given for the same parse string. Unless otherwise stated in the discussions of source types below, the parse string for each template will be the same. In that case, the only point of using multiple templates would be to restore the parse string to its original condition after having hacked it about with pattern markers.

Sources

Once extracted from the source, all parse strings are on the same footing, though their origins are highly various. A supplementary operation that can be applied to all sources is conversion to upper case before parsing. This is achieved by using the keyword 'UPPER' before the keyword specifying the source. We now consider each of the sources in turn:

ARG

The arguments to the present script or internal function are presented in sequence to the templates provided. In the case of scripts invoked as commands (the usual method), at most one argument will be available, so only one template will be required. Instructions like this are therefore very common at the start of scripts:

```
parse arg filename
```

which is exactly equivalent to:

```
filename = arg(1)
```

Multiple arguments to a function (including scripts invoked as functions) require multiple templates, but these are often just variables to which the whole of each argument is assigned, as in:

```
parse arg mass, temperature, volume
```

which is exactly equivalent to:

```
mass        = arg(1)
temperature = arg(2)
volume      = arg(3)
```

Although it is very common to assign the whole of each argument to its own variable, the templates used with PARSE ARG are of course just the same as with any other source, and more complex parsing can be used in any or all templates:

```
parse arg firstname lastname .,street,city,state zip
```

When conversion of the arguments to upper case is wanted, the UPPER sub-keyword can be applied. But generally the ARG instruction, an exact synonym for PARSE UPPER ARG, is used instead. This is particularly appropriate when the arguments are numeric:

```
arg mass, temperature volume
```

EXTERNAL If a file with the identifier 'STDERR' is currently open, one line is read from it to be used as the parse string. If 'STDERR' is not open, the parse string is null. The usual reason for opening a file of this name is that it provides an alternative channel for input and output during tracing, thus keeping the trace information separate from the script's other console i/o (through 'STDOUT'). PARSE EXTERNAL might well be used in scripts whose standard input and/or output are meant to be redirected, or which—because they are launched asynchronously from a CLI with a command like:

```
run rx <scriptname>
```

do not have a usable standard input at all. All that is required to open a console for these situations is:

```
call open('STDERR','con:0/0/640/100/Alternate console')
```

and to read a line from that console:

```
parse external line
```

The input prompt is the same as that used for the PULL source, set with OPTIONS PROMPT.

If PARSE EXTERNAL is used with multiple templates, a new line is read from 'STDERR' for each template.

NUMERIC The parse string contains the current NUMERIC options, in the order *digits fuzz form*. The string obtained is equivalent to the string:

```
digits() fuzz() form()
```

By default, its contents will be:

```
9 0 SCIENTIFIC
```

PULL A line is read from the file with the identifier 'STDIN' if one is available. Normally this will be the console from which the script was launched, unless input has been redirected. In some circumstances, such as when a script is launched with the AmigaDOS Run command, no 'STDIN' is available unless special arrangements are made by the script (see the entry for the PRAGMA function in this Reference Section). Input via PULL is prompted with the string set with OPTIONS PROMPT. Initially, the prompt string is null.

If PARSE PULL is used with multiple templates, a new line is read from 'STDIN' for each template.

When conversion of the input line(s) to upper case is desired, the UPPER sub-keyword can be applied. But generally the PULL instruction, an exact synonym for PARSE UPPER PULL, is used instead.

SOURCE The parse string contains information about the way the current script was called. The string has the form:

```
type result called resolved ext host
```

The meanings of the individual fields are:

type	How the script was invoked, either COMMAND or FUNCTION
result	Whether a result was requested (1) or not (0)
called	The name under which the script was invoked
resolved	The full name, including path and extension, under which the script was located
ext	The current default file extension
host	The initial host address

The following script was saved as "rexx:Test.rexx":

```
/* PARSE SOURCE */
parse source src
say src
```

and invoked with:

```
Shell> rx test
```

It produced this output:

```
COMMAND 0 test Work:Scripts/rexx/Test.rexx REXX REXX
```

VALUE An expression (delimited on the right with the sub-keyword WITH) is evaluated, and the result is used as the parse string. If the variable 'v' has the value 3, then after this:

```
parse value v "*" v "=#" v*v with s "#" v .
```

its new value would be 9, while that of the string 's' would be '3 * 3 ='. The expression is not re-evaluated if multiple templates are used. For example, suppose we give the template above twice:

```
parse value v "*" v "=#" v*v with s "#" v ., s "#" v
```

The final values of 's' and 'v' are the same as before, whereas if the expression were re-evaluated between templates we would expect 'v' to contain 81, and 's' to contain '9 * 9 ='.

VAR The contents of the named variable are used as the parse string:

```
parse var coord x1 ',' y1
```

Unlike PARSE VALUE, the parse string can change between templates if it is used as a target. Consider this parsing of a variable 'f4' that contains the string 'fe-fi-fo-fum':

```
parse var f4 fa'-'f4, fb'-'f4, fc'-'f4, fd'-'f4
```

Try this, and afterwards you'll discover that 'fa' and its fellows each contain one syllable of the original chant, while 'f4' is now empty. Which is not to say that the following wouldn't have been better if you really wanted to achieve that effect:

```
parse var f4 fa'-'fb'-'fc'-'fd f4
```

VERSION The parse string contains information about the versions of ARexx and the machine configuration on which it is being run, in the form:

```
ARexx version cpu mpu video freq
```

The meanings of the individual fields are:

version	The release version of ARexx (e.g. V1.15)
cpu	The microprocessor (e.g. 68000)
mpu	The math coprocessor (e.g. 68881) or 'NONE'.
video	The type of video, either 'NTSC' or 'PAL'.
freq	The line frequency, either '60HZ' or '50HZ'.

The present release of ARexx does not distinguish between the 68881 and 68882 math co-processor chips, reporting both as the 68881.

See also: arg, pull

Chapter 10 (Parsing and String Handling) covers parsing and parse templates in detail, with numerous examples.

Name:	PERMIT
Type:	support function
Format:	count = PERMIT()
Description:	re-enable multitasking after FORBID

***Dialog* example:**

```
->forbid()
0
->forbid()
1
->permit()
0
->permit()
-1
```

Discussion: Every call to the FORBID function, which turns off multitasking, must be balanced by a call to PERMIT. Every FORBID increments a 'nesting count'; every PERMIT decrements it. Both functions return the new value of the nesting count to the caller. When the count reaches -1, multitasking is enabled once again.

See also: A detailed explanation of the usage of FORBID and PERMIT is given under the entry for FORBID. For an example of a script that requires the protection of these functions, see the Reference Section entry for NEXT.

Name:	POS
Type:	built-in function
Format:	n = POS(pat,str,[start])
Description:	return start position of pattern *pat* in string *str*

***Dialog* examples:**

```
->index("rubadubadub","ub")
2
->pos("ub","rubadubadub")
2
->index("rubadubadub","ub",3)
6
->pos("ub","rubadubadub",3)
6
```

Discussion: POS is identical operation to INDEX except in one detail: the order of the two string arguments is reversed. POS is actually the standard form of the function, as implemented in ARexx's parent language REXX. The ordering of the arguments in INDEX reflects that of the 'index' function provided in some C libraries. Which you use is a matter of custom and taste.

See also: index, lastpos, find

Name:	PRAGMA	
Type:	built-in function	
Format:	oldcd = PRAGMA('d', [newcd])	/* Current directory */
	oldpri = PRAGMA('p', newpri)	/* Task priority */
	oldsize = PRAGMA('s', size)	/* Stack size */
	1 = PRAGMA('w', [mode])	/* DOS requesters */
	id = PRAGMA('i')	/* Task ID (address) */
	bool = PRAGMA('*', [file])	/* Console handler */
Description:	A grouping of system-specific facilities as one function	

***Dialog* examples:**

```
->pragma('d')
Work:text              /* Current directory   */
->pragma('d','ram:')
Work:text              /* Returns previous CD */
->pragma('d','Work:text')
Ram Disk:

->pragma('p',1)        /* Set my priority to 1      */
0                      /* Returns previous priority */
->pragma('p',0)
```

```
1

->pragma('s',15000)  /* Set stack to 15000 */
10000                /* Old stack size     */

->pragma('w','n')    /* Turn off DOS requesters */
1                    /* Always returns 1        */
->exists('UFO:')     /* Is there a volume UFO:? */
0                    /* No - but no requester   */
->pragma('w')        /* Turn on DOS requesters  */
1

->pragma('i')
07EB5380             /* Address of my task */
```

Discussion: Though it is implemented as a single function, PRAGMA is actually a grab-bag of necessary house-keeping facilities relating more to the Amiga system than to the ARexx language itself. There are currently 6 PRAGMA commands. Since they have little in common, we'll take them one by one.

Get/Set Current Directory: oldcd = pragma('d',[newcd])

Every AmigaDOS process has a current directory (CD) on start-up. The CD is the process's home base in the filing system. Path names that do not begin with device or volume names are taken by AmigaDOS as being relative to the CD. Every ARexx script inherits the CD of the process (often a CLI) that created it, but can modify it if desired with this facility. Any new process launched by the script itself (such as an AmigaDOS command or another script invoked as a function) will in turn inherit the modified rather than the original CD. The current CD can be determined without modifying it by omitting the second argument.

Modify Task Priority: oldpri = pragma('p',newpri)

Another property that a script inherits from its parent process is its task priority, which helps determine its share of CPU time relative to other tasks. Programs launched from within a script inherit its priority in turn, and it is for their benefit that this variety of PRAGMA would ordinarily be used. Supposing your script is launched with a priority of 0 (the usual case), but that it will in turn launch (with ADDRESS COMMAND) a terminal program, say, which you would prefer to run at priority 1. You could achieve this with instructions like:

```
oldpri = pragma('p',1)          /* change priority temporarily */
address command 'run myterm'    /* launch terminal at new pri  */
call pragma('p', oldpri)        /* restore original priority   */
```

Set Stack Size: oldsize = pragma('s',newsize)

The AmigaDOS Stack command allows you to set, for programs launched thereafter, the size of the special data area known as the stack. ARexx scripts inherit this stack size, like other properties, from the process that launched them, and pass it on in turn to other processes that launch themselves. If some program you wish to launch from a script (with ADDRESS COMMAND) has an unusually high stack requirement, you can increase your stack size in preparation for calling it. The form is very similar to that used for modifying the task priority:

```
oldsize = pragma('s',15000)    /* set stack size temporarily  */
address command 'run myprog'   /* launch stack-hungry program */
call pragma('s',oldsize)       /* restore original stack size */
```

Disable/Enable Requesters: 1 = pragma('w',[mode])

When AmigaDOS encounters an error condition that might be correctable if the user intervenes, its usual action is to post a requester on the Workbench screen telling the user what action to take. This most often happens when an attempt is made to access a file on a disk volume that is not currently mounted. For example, if a script calls EXISTS or OPEN on a file named 'mydisk:myfile', and no volume named 'mydisk:' is currently mounted, the user will ordinarily be presented with a requester bearing the message: 'Please insert volume mydisk: in any drive'. Only if the user cancels this requester does AmigaDOS give up on locating the file, causing the ARexx function call to fail.

Occasionally it is handy to be able to test for the existence of a file (or directory) without the danger of requesters popping up to bother the user. PRAGMA's 'w' option makes this possible. If the second argument with this option is the mode 'n', AmigaDOS will no longer post warning requesters for operations originating with the calling script's process. If the second argument is 'w', or is not given, requesters will be posted to the Workbench screen as usual. Normally it is best to leave requesters enabled as much as possible.

The return value from this variety of PRAGMA is always 1 (or True).

Get Task ID: id = pragma('i')

Sometimes it is useful to be able to create an identifier string that you know is unique to one particular invocation of a script. A good source for such a string is the address of the script's Task information among the operating system's data structures, which is obviously a value that

other invocations of the script will not share. This string can be used to create unique names for objects, like Message Ports, that are visible throughout the system:

```
portname = 'Port_'pragma('i')
```

Set Default Console Handler: bool = pragma('*', [file])

The 'console handler' is the interactive file associated with the special console file name '*'. Normally one interactive file, operating through the CLI window, is used for both the standard input and the standard output files (STDIN and STDOUT). Sometimes it is better to operate interactive I/O through a customized console window (opened as a file whose name begins with 'con:'). If a script was invoked from the CLI with the AmigaDOS Run command, keyboard input would not be possible at all through the CLI window.

The second argument to this variety of PRAGMA is the ARexx file identifier of the interactive file which is henceforth to be used as the console handler. If this argument is not given, the original console handler for the process (if any) is re-installed.

The following function, DIVERT, accepts a console window specification as its sole argument, and attempts to reroute standard input and output to that window. If it is unable to do so (owing, probably, to a faulty specification), it re-installs the default console handler.

```
/* divert

   Divert standard input and output streams to a
   (console) file opened according to the passed-in
   spec. If this fails (probably because of a bad
   spec), the default i/o streams are reopened. e.g.:
   con_opened = divert('con:0/0/640/200/My console')
*/

divert: procedure

call close("STDOUT")
call close("STDIN")

success = 0

if open("STDOUT",arg(1),"w") then do
   call pragma("*","STDOUT")

   if open("STDIN","*","r") then
      success = 1
   else
      call close("STDOUT")
```

```
   end

if ~success then do
   call pragma("**")
   call open("STDOUT","**","w")
   call open("STDIN", "**","r")
   end

return success
```

Name:	PROCEDURE
Type:	instruction
Format:	PROCEDURE [EXPOSE var [var ...]]
Description:	protect function caller's variables from name collisions

Script example:

```
/* PROCEDURE */
a = 'aardvark'
z = 'zymosis'

call abc()

say "From '"a"' to '"z"'."
exit

abc: procedure expose z
   a = 'aardwolf'
   z = 'zymotic'
   return
```

Discussion:

After an internal function has used the PROCEDURE instruction, any variables it references are private, and do not interfere in any way with the variables used by the function's caller, even though the variable names may be the same. They are termed 'local' variables. Local variables cease to exist upon return from the function.

PROCEDURE thus provides useful protection for the caller of a function. Sometimes, though, the protection is a little too much: the called function may need to access or modify certain of the caller's variables even while cutting itself off from the rest. Those exceptions can be handled with the EXPOSE sub-keyword. Any variables listed after EXPOSE are shared between the function and its immediate caller. A stem (a name ending with a period) in the EXPOSE list exposes all compound variables formed on that stem. The sharing of variables does not extend back over multiple generations (to the caller's caller, and so on) automatically: variables must be 're-exposed' at each new level.

A typical place to put the PROCEDURE instruction is on the same line as the function's label, as in the example. The instruction is legal anywhere within the function, however (though only once), and applies to all succeeding variable references. PROCEDURE is not legal other than in an internal function.

See also: call

Name:	PULL
Type:	instruction
Format:	PULL [template] [,template ...]
Description:	shorthand form of PARSE UPPER PULL

Script example:

```
/* PULL */
LF  = '0a'x       /* 'linefeed'                          */
RI  = '1b'x"M"    /* 'reverse index' (cursor up)         */
EL  = '1b'x"[1K"  /* 'erase in line' (clear to eol)      */

options prompt copies(RI,2)"RETURN on empty line to quit: "EL
line = copies('-',70)
say line || copies(LF,3)line || RI

do until line=''
   pull line
   say line || EL
   end

say
```

Discussion: Since PULL means exactly the same thing as PARSE UPPER PULL, you will find a detailed treatment of it in the entry for PARSE. The advantage of PULL is simply its conciseness.

You may or may not find it desirable in a given situation that PULL translates the input line to upper case. If it would be better to leave it in the original case—for instance if the input were to be redisplayed or saved to a file for possible future reference by the user—use PARSE PULL instead.

See also: parse, arg

Chapter 10 discusses PARSE extensively, and should be read by those who are not yet familiar with its many features.

Name:	PUSH
Type:	instruction
Format:	PUSH expr
Description:	pre-load the standard input in 'last-in, first-out' order

Script example:

```
/* PUSH example - sort files in descending size order
   This only works with AmigaDOS 2.0 or ConMan
*/
address command "list >t:aaa.a." arg(1) "nohead files"
address command "sort t:aaa.a. to t:aaa.b. colstart=25"
count = 0

if open('sf', 't:aaa.b.') then do
   line = readln('sf')
   do while ~eof('sf')
      if ~abbrev(line, ':') then do
                              /* eliminate filenotes */
         push line
         count = count + 1
         end
      line = readln('sf')
      end
   call close('sf')
   call delete('t:aaa.a.')     /* kill temp files */
   call delete('t:aaa.b.')

   do count
      parse pull line
      say line
      end
   end
```

Discussion: Normally one reads from the standard input file 'STDIN', and does not write to it. After all, it's normally the console input file—the keyboard—so what can writing to it even mean? Actually, the words 'input' and 'output' can be a bit deceptive. Why does a mirror flip your image horizontally but not vertically? It's all a matter of viewpoint. The standard input file provides input to your script, but accepts output from the keyboard. The standard output file accepts output from your script, but provides input to the screen.

With PUSH (and its brother QUEUE), you are putting yourself in the keyboard's shoes (as it were), jumping two links back in the input chain. Now you can output to your own input file—which in general is also the input file of your parent CLI.

If you use the Shell much, you know that it lets you 'type ahead' of it. Type:

```
Shell> List c:
```

then before the listing is complete, add:

```
Shell> List s:
```

The second command isn't acted upon immediately—the first one is still executing. But when it's done, we find that the second command wasn't ignored—it was simply set aside till it could be acted upon. Thus we learn that there is a waiting list at the standard input... all PUSH has to do is add new lines to that list.

The lines you type ahead at the console are added to the end of the waiting list in true democratic style: it's first come, first served, or, as computerists say, first in, first out, or FIFO. True to its name, however, PUSH bulls its way in at the head of the line: its idea of lining up is last in, first out, or LIFO. If we wanted to simulate the type-ahead command above, for instance, we would use:

```
Shell> rx "push 'list s:'; push 'list c:'
```

The RX command would do nothing itself except push the given lines back onto the standard input. The next time the Shell goes for a line, however, the first thing it will find is 'list c:', and that is the command it will execute. Next it will find 'list s:', and execute that, and only then will it start receiving lines from the keyboard again.

With PUSH you can also use 'STDIN' as a scratchpad—a private 'stack' for string storage. The example script takes advantage of the LIFO character of PUSH to reverse the strings in a sorted file, and so achieve a reverse sort. It doesn't really care that PUSH interacts with the standard input file—any other stacking facility would do just as well. But PUSH does the job.

There's only one serious drawback to using PUSH: it doesn't work under AmigaDOS 1.3 unless you're running William Hawes' shareware 'ConMan' program. Hawes is the author not only of 'ConMan' but also of two commercial products: a replacement command shell called 'WShell', and ARexx itself. All three products were designed with each other in mind, so it isn't surprising that an ARexx facility like PUSH would require a capability that—until AmigaDOS 2.0—was found only in ConMan. But that's the way it is: if you want to take advantage of PUSH (and QUEUE), you need either AmigaDOS 2.0, or ConMan. Otherwise PUSH will have no effect.

See also: queue, lines, pull

Name:	QUEUE
Type:	instruction
Format:	QUEUE expr
Description:	pre-load the standard input in 'first-in, first-out' order

Script example (2 scripts):

```
/* QUEUE - queue1.rexx */

queue 'rx queue2'
queue 'Y'
queue 'Y'

/* QUEUE - queue2.rexx */
options prompt "Are you sure you want me to calculate 12**5? "
pull yn

if yn='Y' then do
   say "Many have gone mad with this knowledge. I ask again:"
   pull yn

   if yn='Y' then
      say 12**5
   end
```

Discussion: Like PUSH, the QUEUE instruction sends lines to the standard input file, from where they will be read in the normal way as though they had been typed at the keyboard. The only difference between PUSH and QUEUE is that the latter loads 'STDIN' in first-in, first-out (FIFO) order. This makes it more natural for pre-entry of command dialogues, as shown in the example. The second script, 'queue2.rexx', can be run by itself, but requires two lines of keyboard input before delivering its real message. The first script, 'queue1.rexx', has as its whole purpose subverting 'queue2.rexx' so that manual input to the latter will no longer be required.

The point is that 'queue2.rexx' represents any program—including commercial programs written in languages like C and assembler—requiring input through 'STDIN', while 'queue1.rexx' represents the script you write to deal with such programs automatically. With QUEUE you can pre-load the program's input stream with as many commands as required, and never actually type a single one.

See also: push, lines, pull

Name:	RANDOM
Type:	built-in function
Format:	n = RANDOM([low],[high],[seed])
Description:	return 'random' integer between *low* and *high* inclusive

Dialog examples:

```
->random()      /* same as random(0,999) */
546
->random(0,1)
1
->random(0,1)
0
->random(3000,3149)
3085
```

Discussion: Games and simulations typically need random numbers for variability from one run to the next. On a computer, 'random' nearly always means 'pseudo-random', which is to say that the numbers are generated according to a mathematical procedure designed so that the generated numbers will appear truly random, both subjectively and to statistical tests. No pseudo-random generator completely succeeds at emulating true randomness—whatever 'true randomness' may be—but the departure from the ideal is negligible for most purposes.

RANDOM takes three arguments, all of which are optional. The first two specify the lower and upper bounds of the range from which the generated random number will be drawn. The gap between the lower and upper values must be no greater than 1000. The default values for these arguments are 0 and 999 respectively. If you wish to generate random numbers over a wider range, use RANDU and scale the result as desired.

The random number generator used by ARexx works from a 'seed' value which is the same at the outset of every script run, resulting in the same sequence of 'random' numbers on each run. This is sometimes useful (for controlled testing of the script while it is being debugged), but in general a different sequence is wanted on each run. The solution is to supply a truly random seed value for RANDOM to work from. The seed is the optional third argument to RANDOM (if it is absent, the random number generator is not reseeded); it must be a non-negative integer. Perhaps the best readily-available source for a seed is the 'seconds' value from the TIME function. Just put the following line near the beginning of any script that uses RANDOM:

```
call random(,,time('s'))
```

or:

```
call randu(time('s'))
```

Particular sequences of 'random' numbers can be duplicated by using the same seed on the first call to RANDOM. The seed given to RANDOM also affects the RANDU function.

See also: randu

Name:	RANDU
Type:	built-in function
Format:	n = RANDU([seed])
Description:	return pseudo-random number between 0 and 1

***Dialog* examples:**

```
->randu()
0.581444375
->randu(100)
0.773375754
->trunc(-500 + randu()*(2000 - -500 + 1))/* -500 <= n <= 200 */
14
```

Discussion: RANDU returns a positive number less than 1. The result can be scaled to any range desired, as in the third example. In general, if you need a random integer between 'lo' and 'hi' inclusive, the formula is:

```
r = trunc(lo + randu() * (hi - lo + 1))
```

The optional argument to RANDU is a seed value, which must be a non-negative integer. The seed given to RANDU also affects the RANDOM function.

See also: random

Name:	READCH
Type:	built-in function
Format:	s = READCH(file, [count])
Description:	return *count* characters from *file*

***Dialog* examples:**

```
->open('w_file','ram:testfile','w')
1
->writeln('w_file','abcdefghijklmnopqrstuvwxyz')
27
->seek('w_file',0,'b')
0
->readch('w_file')
a
```

```
->readch('w_file')
b
->readch('w_file',6)
cdefgh
->readch('w_file',4)
ijkl
->readln('w_file')
mnopqrstuvwxyz
->close('w_file')
1
```

Discussion: READCH reads a given number of characters (the default is 1) from an open file. The characters need not be printable letters, numbers and punctuation, or even recognized ASCII control characters like tab, backspace and linefeed. They can have any value from 0 through 255. If fewer than the requested number of characters are available in the file, READCH will return as many as it can. If the number of characters returned is less than the requested length, either a read error has occurred, or—far more usually—the end of file has been reached. In the latter case, a call to the EOF function will return 1.

The Dialog examples above show the difference between READCH and READLN. The first three function calls (OPEN, WRITE and SEEK) create a test file containing the letters of a lower-case alphabet, ending with a linefeed (added automatically by WRITELN), then seek back to the start of the file. The first two calls to READCH fetch 1 character each, making use of the default value for the second argument. Then 6 characters are read, then another 4. Finally, READLN brings in not a given number of characters but the rest of the line, up to the linefeed.

See also: readln, writech, writeln, eof

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	READLN
Type:	built-in function
Format:	s = READLN(file)
Description:	return a line from the file as a string

Example:

```
/* MLT - given a text file name as a command line
   argument, determine and list the  byte offsets of
   each line in the file.
   Usage: rx mlt <filename>
*/
nlines = MakeLineTable(arg(1))
```

```
if nlines < 0 then
   say "Unable to open file '"arg(1)"'"
else do
   say "Line Offset"
   do i=1 to nlines
      say right(i,4) right(linetab.i,6)
      end
   end

exit

/* MakeLineTable(filename)

   Build table of line offsets for file, return line
   count.
*/
MakeLineTable: procedure expose linetab.
   nlines   = 0
   offset   = 0
   linetab. = -1
    /* Invalid default offset for all lines */

   if open('mlt_file',arg(1)) then do
      line = readln('mlt_file')

      do while ~eof('mlt_file')
         nlines          = nlines + 1
         linetab.nlines = offset
         offset          = offset + length(line) + 1
         line            = readln('mlt_file')
         end

      call close('mlt_file')
      end
   else
      nlines = -1
        /* Return negative if file won't open */

   return nlines
```

Discussion: READLN returns as a string all the text between the current file position and the next linefeed character or the end of the file, whichever comes first. If there is a linefeed, it is read in (advancing the file position beyond it), but it is not added to the returned string. If the end of file is reached before a linefeed is found, the end-of-file condition is set, and the partial line is returned as usual. Only an actual linefeed (ASCII 10) counts as a line separator; the carriage return (ASCII 13) has no special status.

See also: readch, writech, writeln, eof

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	REMLIB
Type:	built-in function
Format:	bool = REMLIB(name)
Description:	remove an entry from the Library List

***Dialog* examples:**

```
->addlib('ersatz.library',0)
1
->show('l')
rexxsupport.library ersatz.library REXX
->remlib('ERSATZ.library')
0
->remlib('ersatz.library')
1
```

Discussion: REMLIB searches the Library List (ARexx's list of available function hosts and function libraries) for an entry whose name matches its argument, and removes the host or library of that name. The boolean return value reflects whether the name could be matched (1) or not (0). As the examples show, the search for a matching name is case-sensitive.

The two situations in which you may wish to call REMLIB are:

1) when memory is getting tight and you have loaded some function library that is no longer needed;

2) when you have added a name to the Library List in error.

See also: addlib, show

Name:	RENAME
Type:	support function
Format:	bool = RENAME(oldname, newname)
Description:	rename a file or directory

***Dialog* example:**

```
Shell> echo >ram:Canaveral "Blast-off!"
Shell> rx "say rename('ram:Canaveral','ram:Kennedy')
1
Shell> makedir ram:Cape
Shell> rx "say rename('ram:Kennedy','ram:Cape/Kennedy')
1
Shell> rx "say rename('ram:Cape','ram:George)
1
Shell> rx "say rename('ram:George/Kennedy','df0:Canaveral')
0
```

Discussion: Like the AmigaDOS command of the same name, ARexx's RENAME function lets you change the name of a file ('Canaveral' to 'Kennedy', in the example), or of a directory ('ram:Cape' to 'ram:George'), or move a file or directory into another directory on the same device ('Kennedy' to 'Cape/Kennedy'). The boolean return value indicates whether the rename was successful.

Again like RENAME, you cannot move a file or directory onto a different device ('ram:' to 'df0:').

Some of the other conditions that can cause RENAME to fail are:

- The file or directory given as *oldname* does not exist
- The file or directory given as *newname* already exists
- Either of the pathnames is invalid
- The volume is write-protected

See also: delete, makedir

Name:	REPLY
Type:	support function
Format:	1 = REPLY(pkt, [result], [result2])
Description:	return a message packet to its sender

Discussion: Most ARexx scripts execute in an environment where the air is thick with message packets darting frenetically in all directions around them, yet remain peacefully oblivious even to the existence of messages and of the message ports to which messages are posted. When a script opens a public port with OPENPORT, and starts receiving message packets of its own from the outside world, it acquires new responsibilities.

Every packet a script receives belongs to some other task, and that task is depending on the recipient of its message to reply, generally as soon as possible. Once its port has been opened, a message-driven script (such as the tiny command host given with the reference entry for GETARG) works like this:

```
Until it is time to quit, do the following loop:
   Wait at the message port for a message packet
   Get a message packet from the port
   Check that the packet is real - is its address null?
   If the packet is real, examine it with GETARG and/or TYPEPKT
   Do whatever should be done about the packet
   REPLY to the packet giving appropriate results
   Go back to the top of the loop
```

If we assume that your incoming messages are arriving from other ARexx scripts who are using your script as a command host, the replies to the messages will include a return code to signal any errors and, optionally, a result value. The return code will be copied into the calling script's RC variable; it should be 0 if no error occurred, a higher number otherwise. The number chosen should follow the usual ARexx practice of using small error numbers (5-9, say) for small errors that the calling script may be able to live with, larger numbers (10-19) for more serious errors, and yet larger numbers (20 or more) for panics, tragedies, catastrophes and fiascos. The result value will be copied into the calling script's RESULT variable in the normal way, provided that it has requested results by setting OPTIONS RESULTS, and that the error code you returned was 0.

The *pkt* argument to REPLY is the address of the message packet to which you are replying. The *result* argument is the numeric error code, and *result2* is the result value. The error code defaults to 0, and the result defaults to the null string.

Any messages to which your script has not replied when it exits will be replied by ARexx automatically. However, you should take care of replying them yourself, and promptly: if you do not, another task, depending on the reply, may be paralyzed meanwhile.

See also: closeport, openport, getarg, getpkt, typepkt, waitpkt

For a programming example including REPLY, and a further explanation of message packets and their role in some ARexx scripts, see the entry for GETARG.

Name:	RETURN
Type:	instruction
Format:	RETURN [expr]
Description:	return control (and optionally a value) to caller

Example:

```
/* RETURN - return.rexx */
say getvalue(getvalue(arg(1) arg(1)))
exit

getvalue: procedure
   return arg(1) arg(1)
```

Example session:

```
Shell> rx return Ho Ho
Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho Ho
```

Discussion: If used within an internal function, RETURN causes control to pass back to the caller of the function, which may be another function, or may be the main level of the script. If used from the main level, RETURN is identical in effect to EXIT, returning control to the invoker of the script.

In either case, RETURN may be accompanied by an expression, whose value is passed back to the caller as the value of the internal function or script. Whether the caller actually wanted an expression returned, or will make use of it, cannot ordinarily be known by the returning function. In many cases, though, it may be appropriate for a script to check (with PARSE SOURCE) whether a result was requested, and act accordingly.

See also: call, exit

Name:	REVERSE
Type:	built-in function
Format:	s = REVERSE(str)
Description:	reverse a string

***Dialog* examples:**

```
->reverse('desserts repaid')
diaper stressed
->reverse('STOP ====>')
<==== POTS
```

Discussion: The argument string is returned with the characters in reverse sequence.

Name:	RIGHT
Type:	built-in function
Format:	s = RIGHT(str,width,[pad])
Description:	extract rightmost *width* characters of *str*

***Dialog* examples:**

```
->right("panache",4)
ache
->right(3,8,0)
00000003
->right(4926872,8,0)
04926872
```

Discussion: RIGHT extracts and returns the given number of characters from the right of the argument string. If necessary to make up the requested count, the argument string is padded on the left with spaces (the default) or the *pad* character. A common use of RIGHT, as demonstrated in the examples, is the right-alignment of a column of numbers, with or without leading zeros.

See also: center, left

Name:	SAY
Type:	instruction
Format:	SAY [expr]
Description:	send expression result to standard output (usually screen)

Discussion: SAY, which is undoubtedly the most used ARexx instruction (possibly excepting the assignment 'instruction', which is in a somewhat different category), is generally equivalent to calling the WRITELN function with 'STDOUT' as the file identifier:

```
call writeln('stdout', <expr>)  /* like SAY <expr> */
```

When you run a script with RX from a CLI, unless you use the shell output redirection operation, the 'STDOUT' identifier is originally associated with the CLI console window, so that is where your output will go. But there is nothing magic about 'STDOUT'—you can close it, and open it on another file—such as the 'SPEAK:' device—if you like. With the aid of the '*' option to the PRAGMA function, you can attach both 'STDIN' and 'STDOUT' to a console window of your own.

See also: writeln, pragma, pull

Name:	SEEK
Type:	built-in function
Format:	n = SEEK(file,offset,[mode])
Description:	move *file* position to *offset* according to *mode*

***Dialog* examples:**

```
->open('w_file','ram:testfile','w')
1
->writeln('w_file','abcdefghijklmnopqrstuvwxyz')
27
->seek('w_file',0)
27
->seek('w_file',10,'b')
```

```
10
->readch('w_file')
k
->seek('w_file',0)
11
->seek('w_file',-10,'e')
17
->seek('w_file',100,'e')
17
->close('w_file')
1
```

Discussion: SEEK lets you determine or modify the current file position within an open file. The first argument to SEEK is the identifier of the file, and the second is a numeric offset specifying a new desired file position according to one of three modes, given as the optional third argument. The return value is the new file position, expressed in bytes from the beginning of the file.

In 'c' ('Current') mode, the default, the new position is the current position plus the offset. An offset of 0 can be used to determine the current file position without modifying it. The offset argument can be either negative (towards the beginning of the file) or positive (towards the end of the file).

In 'b' ('Beginning') mode, the new position is at the specified offset from the start of the file. If all goes well, the offset argument will equal the return value. The offset must be positive (including 0) for a seek in this mode to succeed.

In 'e' ('End') mode, the new position is at the specified offset from the end of the file. The offset argument must be negative or 0 for a seek in this mode to succeed.

If the absolute file position calculated from the combination of the given mode and offset is negative, or past the end of the file, the attempt to seek fails, and the file position does not change.

See also: readch, writech

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	SELECT
Type:	instruction
Format:	SELECT; WHEN ... [OTHERWISE [;] [instructions]] END (WHEN ... in the template stands for one or more instances of: WHEN expr [;] THEN [;] instruction The instruction can be a compound instruction such as a DO-block, an IF-THEN-ELSE, or even a SELECT instruction.)
Description:	branch to first case whose controlling condition is met

Script example:

```
/* SELECT example: cheerup.rexx */
options prompt "Your worst problem in one word: "
pull problem

select
  when problem = 'HEALTH' then
     say "But while there's life there's hope!"
  when problem = 'MONEY' then
     say "At least you've got your health!"
  when find('WOMEN MEN GIRLS BOYS LOVE', problem) > 0 then
     say "At least you're financially comfortable!"
  when problem = 'JOB' | problem = 'BOSS' then do
     say "Well, things are bound to get better..."
     say "And lots of people have it worse than you!"
     end
  otherwise
     say "I know what you mean! That's my problem too!"
     say "Let's get together and form a support group!"
  end
```

Discussion: In most respects, SELECT is just a slightly more formal alternative to a chain of IF-THEN-ELSE statements. Whereas the indentation of an IF chain is conventionally fudged so that the alternative cases are at the same indentation level (logically speaking the whole chain should slant gradually away to the right), with SELECT the fudged indentation is institutionalized. That doesn't change the fact that the earlier cases have priority, however. If more than one of the WHEN test conditions is satisfied, it is the dependent instruction of the earlier one that will be executed, while the later one will be ignored.

If none of the WHEN conditions is True, ARexx insists on being able to default to an OTHERWISE, though—unlike all other keywords that govern dependent instructions—it is not required that any instructions actually be placed between the OTHERWISE and the END instruction that terminates the SELECT. Another special syntax feature of OTHERWISE is that it can govern multiple instructions without requiring a DO-END block.

As with IF-ELSE, it is sometimes convenient to use the NOP instruction with a WHEN, as in this model of a partially-implemented command system. Here, any alphabetic character is accepted as a command by the system, but only a few are actually hooked up:

```
select
   when cmd='A' then call archive()
   when cmd='E' then call erase()
   when cmd='Q' then call quit()
   when cmd='V' then call verify()
   when datatype(cmd, 'u') then nop   /* all letters allowed */
   otherwise
      say "Invalid command!"
   end
```

Whether you use IF-ELSE or SELECT-WHEN-OTHERWISE in a particular programming situation is mostly a matter of taste. Rough tests indicate that SELECT is usually more efficient, more markedly so when the number of alternatives is large, but that is usually a consideration only in 'tight' loops with a large number of repetitions.

See also: when, then, otherwise, if, then, else, do, end

Name:	SETCLIP
Type:	built-in function
Format:	bool = SETCLIP(name, [value])
Description:	set string *value* for clip *name*

***Dialog* examples:**

```
->setclip('herman','melville')
1
->'['getclip('herman')']'
[melville]
->setclip('herman')
1
->'['getclip('herman')']'
[]
->setclip('herman')
0
```

Discussion: ARexx maintains a list of strings called 'clips', which is globally available to all scripts. Each entry on the Clip List pairs a unique name with a value string; SETCLIP can either create or delete these entries. The entries may be retrieved with GETCLIP.

SETCLIP's first argument is the name of the clip, and the optional second argument is the value to be associated with the name. If a clip of the given name already exists, the new value string replaces the

previous one. If the second argument is not given, the clip of the given name is deleted.

The boolean return value indicates whether the SETCLIP call succeeded.

See also: getclip

Name:	SHELL
Type:	instruction
Format:	SHELL
	SHELL name
	SHELL VALUE name-expression
	SHELL name command-expression
Description:	modify host address; send command to a host

Discussion: SHELL is an exact synonym for the ADDRESS instruction; consult the entry for ADDRESS in this Reference Section for examples and discussion.

Name:	SHOW
Type:	built-in function
Format:	s = SHOW(mode,,[pad])
	bool = SHOW(mode,name)
Description:	return information about a resource

***Dialog* examples:**

```
->show('L')
rexxsupport.library HB_REXX REXX
->show('L',,'0a'x)
rexxsupport.library
HB_REXX
REXX
->show('L','rexxsupport.library')
1
->show('L','ersatz.library')
0
```

Discussion: SHOW yields information about various types of named objects that are organized in lists. Each SHOW mode (as expressed in the first argument) corresponds to a different type of object:

'C' ('Clips')	Clips, by clip name (see GETCLIP, SETCLIP)
'F' ('Files')	Open files, by file identifier (see OPEN)
'I' ('Internal')	Internal message ports (see OPENPORT)
'L' ('Libraries')	Function libraries and hosts (see ADDLIB)
'P' ('Ports')	System message ports

In the first form of SHOW (no second argument), all the names on the list for the given resource are returned as a string, separated by the *pad* character, or by spaces in its absence. In the second example, the use of a linefeed ('0a'x) as the *pad* character causes the names to appear on separate lines.

The second form of SHOW returns 1 if the given *name* occurs on the list specified by *mode*, and 0 otherwise.

See also: showlist

Name:	SHOWDIR
Type:	support function
Format:	filelist = SHOWDIR(dir, [mode], [pad])
Description:	list the file and/or directory names in a directory

Dialog examples

```
/* For 'mydir' in these examples substitute the name
   of any directory in your system with both files
   and subdirectories.
*/
->showdir('mydir')
Railroads Utilities Vacuum.anim Silence.snd Lamprey.pic
->showdir('mydir','f')
Vacuum.anim Silence.snd Lamprey.pic
->showdir('mydir','d')
Railroads Utilities
->showdir('mydir',,',')
Railroads,Utilities,Vacuum.anim,Silence.snd,Lamprey.pic
->showdir('mydir','f','0a'x)
Vacuum.anim
Silence.snd
Lamprey.pic
```

Discussion: By default, SHOWDIR returns a list containing the names of both the files and the subdirectories in the directory whose name is given as its first argument. The second argument is a specifier for one of three modes:

Mode	Lists...
'A' (All)	Both files and subdirectories; the default
'F' (Files)	Files only
'D' (Directories)	Subdirectories only

The *pad* argument is the character used to separate entries in the returned list. As usual, the default pad character is the space. Other useful possibilities include the comma and the linefeed ('0a'x), both illustrated in the examples.

See also: exists, statef

Name:	SHOWLIST
Type:	support function
Format:	list = SHOWLIST(mode,,[pad])
	bool = SHOWLIST(mode, name)
	addr = SHOWLIST(mode, name,, 'a')
Description:	return information about a shared system list

***Dialog* examples:**

```
->showlist('V')
AREXX_DISK RAM DISK WORK SYSTEM2.0
->showlist('V',,'0a'x)
AREXX_DISK
RD
WORK
SYSTEM2.0
->showlist('V','AREXX_DISK')
1
->showlist('V','DF0')
0
->showlist('H','DF0')
1
->c2x(showlist('L','intuition.library',,'a'))
07E096FC
```

Discussion: As discussed under other topics in the Reference Section (e.g. NEXT and FORBID), system information that is shared between tasks is stored in a variety of 'linked lists', each one devoted to a particular type of data structure. (This is technically a slight simplification: on some lists, such as the one for handlers, volumes and assigns, related structures share a single list.)

Locating and scanning a list correctly is not only a bit of a chore, as the programming example for NEXT shows, but also requires a more

detailed knowledge of the operating system than most people need or want. Fortunately, most lists of interest are available very straightforwardly with the SHOWLIST function, which can furnish information about 12 kinds of data. The *mode* argument selects which list will be scanned, according to this table (as usual, the mode may be specified in the function call by its first letter only, if preferred):

Mode	Description
Assigns	A list of logical (assigned) directories, in upper case, including standards like 'C', 'LIBS' and 'DEVS', plus any assigns you may have made for your own purposes.
Devices	These are software 'devices', with names like 'timer.device' and 'console.device'; their purpose is to provide a consistent way for programs to deal with hardware.
Handlers	These are AmigaDOS handlers (a higher-level interface to devices) listed in upper case: 'PIPE', 'CON', 'DF0'.
Interrupts	The names of nodes on the 'IntrList' list of Interrupts in the Exec Library structure. It is hard to conceive of an application that would require this mode.
Libraries	The shared libraries in the system, including 'normal' libraries like 'intuition.library', and ARexx function libraries like 'rexxsupport.library'.
Memory	The names of nodes on the 'MemList' list of memory blocks in the Exec Library structure, e.g. 'chip memory'.
Ports	The names of nodes on the 'PortList' list of public message ports in the Exec Library structure. This is the same list you see with the 'p' mode of the built-in function SHOW.
Resources	'Resources' are pieces of Amiga system software, on a par with libraries and devices, that have the special job of governing access to hardware, which by its nature can't be freely used by multiple tasks simultaneously. SHOWLIST('r') will return names like 'battclock.resource', which handles the battery

backed-up clock, 'disk.resource', and 'potgo.resource', which manages the joystick/mouse ports.

Semaphore — 'Semaphores' are another mechanism for avoiding conflicts between tasks trying to access resources simultaneously, and are used by some programs as a less drastic alternative to turning off multitasking. Access to Exec's semaphore list is of dubious benefit to ARexx scripts.

Task ready — Tasks that are not currently active but are prepared to go to work when the task scheduler gives them a chance are termed 'ready'. By now you will hardly be surprised to learn that Exec maintains a list of ready tasks. At most times on most systems the list will be empty or nearly empty.

Volumes — A list of volume (disk) names in your system, in upper case.

Waiting — Most Amiga tasks are idle most of the time, waiting for one or more pre-arranged wake-up calls: a keypress, the completion of disk activity, a time-out, a mouse-click, and many more. Such a task is said to be waiting. The instruction SHOWLIST('w') should return a quite surprisingly long list of tasks in this state.

If SHOWLIST is called with only the *mode* argument, the result is a list of names separated by spaces. To get the same list with the pad character of your choice, leave the *name* argument null and supply the *pad* character.

The second type of SHOWLIST call specifies both a mode and a name that is likely to be on the list in question. The boolean return indicates whether the name is indeed on the list. The third, fourth and fifth Dialog examples above are all queries of this type.

The third type of SHOWLIST call actually returns the address, as a 4-byte string, of a member of the given list, or the null address string if the name is invalid.

See also: next, null, offset

An example of a script that uses SHOWLIST extensively, but is unfortunately slightly too long to be listed here, may be found on the accompanying disk as 'asnvol.rexx'.

Name:	SIGN
Type:	built-in function
Format:	n = SIGN(num)
Description:	determine the sign of a number

***Dialog* examples:**

```
->sign(11)
1
->sign(0)
0
->sign(-33.987e11)
-1
->sign(33.987e-11)
1
```

Discussion: SIGN returns -1, 0 or 1 for a negative, zero or positive argument respectively.

See also: abs

Name:	SIGNAL
Type:	instruction
Format:	SIGNAL ON type
	SIGNAL OFF type
	SIGNAL [VALUE] label-expression
Description:	Turn interrupt type on/off; transfer control to label

Script example:

```
/* SIGNAL - springing a variety of traps */
signal on syntax
signal on error
signal on failure
signal on halt
signal on ioerr
signal on novalue

syntax_test: test = 'error'
   say center('This will fail!') /* Wrong # of args */
error_test: test = 'failure'
   'does_not_exist'          /* Non-existent command */
failure_test: test = 'novalue'
   address command "List ')('" /* Bad List template */
novalue_test: test = 'ioerr'
   say trapezium                 /* Not initialized */
ioerr_test: test = 'halt'
   call open('pf','prt:')
   say readln('pf')           /* Can't read printer! */
halt_test:
```

```
   address command hi              /* Halt ourselves? */
shutdown:
   say "Exiting gracefully after all the chaos..."
exit

syntax : call report(sigl, "SYNTAX ("errortext(rc)")");signal
next
error  : call report(sigl, "ERROR")  ; signal next
failure: call report(sigl, "FAILURE"); signal next
novalue: call report(sigl, "NOVALUE"); signal next
ioerr  : call report(sigl, "IOERR")  ; signal next
halt   : call report(sigl, "HALT")   ; signal shutdown

next:
   signal value test'_test'

report: procedure expose rc
   parse arg sl, text
   pen1 = '1b'x"[31m"; pen2 = '1b'x"[32m"
   say pen2 || text || pen1 "trap (rc =" rc", sigl =" sl")."
   return
```

Discussion: Although ARexx excels as a casual scripting language for one-time needs, or for throwing together personalized utilities that don't require such niceties as protected input or detailed error-checking, it is also suitable for the large applications in which those 'niceties' suddenly become 'essentials'.

SIGNAL provides an error-trapping facility that allows a script to maintain control when something goes wrong. The 'something' might be an error arising within the script itself, such as a syntax error or a memory allocation failure; it might be a 'failure' return code from an external command; or it might be the user hitting CTRL-C in an attempt to interrupt whatever the script is doing.

These conditions and several others can be trapped individually by setting the appropriate signal to 'on'. Once a particular signal has been enabled, if the corresponding condition occurs, the normal execution of the script is interrupted, and control is transferred to a label named for that type of signal. For instance, a trap for syntax errors can be set with:

```
signal on syntax
```

The trap is sprung the next time a syntax error occurs. When that happens, control is transferred to this label, somewhere in the same script:

```
syntax:
```

Of course, if there isn't such a label, that in itself is a syntax error! So what happens? Nothing drastic, actually. To avoid the disastrous kind of circularity that could arise in such situations, the activation of a trap simultaneously disables the corresponding signal. Further errors of that type will not be trapped until the appropriate SIGNAL ON instruction has been given again.

SIGNAL OFF, of course, is simply a way of disabling the signal 'manually'. It is much less often used than SIGNAL ON.

The break signals

Among the conditions that can be trapped are the 'break signals' recognized by AmigaDOS. There are four of these, known as BREAK_C, BREAK_D, BREAK_E and BREAK_F. They are activated either with the AmigaDOS Break command, or by typing one of the letters C, D, E or F (either upper or lower case) while holding down the 'Ctrl' key. The most often used, BREAK_C, is recognized by most AmigaDOS commands as a request to halt, while BREAK_D is used for interrupting the operation of an AmigaDOS script.

The default behavior of BREAK_C in an ARexx script is to cause a halt, and return an error code of 10. The other break signals have no default effect. By setting SIGNAL ON for any of these, however, they can be trapped individually and used to generate any appropriate response 'asynchronously'—that is, regardless of the other activities the script may be involved in at the time.

There is a caveat with respect to the break signals: they do not work under AmigaDOS 2.0 for ARexx scripts getting their standard input and output from a CLI window, though the BREAK_C interrupt does work for AmigaDOS commands and for other programs that normally support it. BREAK_C (and the other break signals) also work for ARexx scripts under AmigaDOS 1.3. But under 2.0, they work only if you reroute your standard input to a custom console window, either with something like:

```
call close('STDIN')
call open('STDIN', 'con:0/0/600/180/My input')
```

or with a routine like Divert, which is listed under the entry for PRAGMA in this Reference Section. (NB: this information with respect to AmigaDOS 2.0 was determined using a late pre-release version of the operating system. It is possible that the problem will have been corrected in the final version.)

RC and SIGL

When a SYNTAX, ERROR or FAILURE interrupt is received, the special variable RC can be examined to learn more about the problem. As the example script demonstrates, text descriptions of syntax errors are available using the ERRORTEXT function. Return codes from commands are also stored in RC, but are of an altogether different type: their significance is conventional only, and is determined ultimately by the command itself. The interpretation of RC in an ERROR or FAILURE trap is therefore dependent on the context of the particular script.

With an IOERR interrupt, the RC variable generally reflects documented AmigaDOS error number (such as 218 for 'Device not mounted') for the operation that has failed, and its meaning can therefore be investigated with the Fault command from the Shell. Not universally however: the foredoomed attempt to read from 'PRT:' in the example script returns error 253, which is not documented.

When an interrupt of any type is received, the special variable SIGL is automatically assigned the number of the script line that was executing when the signal was detected. This can be used to tailor the response taken by the trap routine to the context in which the signal was trapped.

Here is a complete list of trap types, and the conditions they trap:

Trap	Description
BREAK_C	A BREAK_C interrupt has been received, usually because the user has entered CTRL-C at the keyboard, or given the AmigaDOS Break command. The interrupt will not be sensed, whether or not the trap is set, under AmigaDOS 2.0 (see the main discussion for more on this).
BREAK_D BREAK_E BREAK_F	These work in the same way as BREAK_C. The only difference is that they do not have the default behavior of halting the script when the break trap is not set.
ERROR	If the FAILURE trap is set, this will trap errors resulting from external commands up to but not including the FAILAT threshold. If FAILURE is not set, this will trap all errors from external commands. The exact return code is stored in the RC variable as usual.

FAILURE — This will trap those errors from external commands that equal or exceed the FAILAT threshold (as set with OPTIONS FAILAT). The exact return code is stored in the RC variable as usual.

HALT — When the ARexx process under which the script is running receives a 'halt' request, generally meaning that the HI support program has been executed, it is normally interrupted on the spot. But if this trap is set, the halt request instead simply causes a branch to the corresponding label. The point of this is not to allow scripts to get around the external halt mechanism, but to give them a chance to exit cleanly: releasing resources like memory or message ports, leaving data files in a fit state for later use, and so on.

IOERR — This traps some input/output failures, though not all. IOERR does not, for example, catch failed file open or seek attempts, but it will catch other errors, such as an attempt to read from a write-only device (as in the example script), or an attempt to access a file open on an unmounted volume. Before depending on IOERR in a particular situation, make sure it can actually trap the errors you hope it will.

NOVALUE — In many languages, it is regarded as an error to use an uninitialized variable in an expression. ARexx is more relaxed, and simply takes the variable as having its own name as its value. If you would prefer the more stringent approach, though, you can have it: just put add these two lines to your script, the first near the beginning, and the second way down at the bottom below an EXIT so that it won't be executed in the normal sequence:

1) signal on novalue
2) novalue: "Uninitialized var, line" sigl; exit

SYNTAX — The range of errors trapped with SYNTAX is quite wide, and includes some (such as memory allocation failures) that aren't really syntax errors at all. A few, such as unbalanced quotes and parentheses, that really are syntax errors, are caught on the initial scan of a script before it begins to execute, and so before any traps are in place. Those

simply cause the script to abort. To see a list of all the 'syntax' errors, run this little script:

```
/* Display all syntax error messages */
do i=1 to 48; say i':' errortext(i); end
```

A few error numbers in the given range are not assigned. These display as 'Undiagnosed internal errors'.

Using SIGNAL 'manually'

Though the primary use of SIGNAL is to interact with the ARexx interrupt system, it is also possible to use SIGNAL to transfer control directly to the label of your choice, using the syntax:

```
signal <expression>
```

The expression is evaluated and taken to be the name of a label somewhere in the current script. Control is immediately transferred to that label. For instance:

```
/* SIGNAL <expr> */
options prompt "> "
say "Enter lines to be 'interpreted' (EXIT to quit)."

restart:
   signal on syntax
   signal on error

   do forever
      parse pull line
      interpret line
      end

syntax: say "Syntax error:" errortext(rc); signal 'restart'
error : say "Command error" rc; signal 'restart'
```

Although in this example the 'expression' consists of the literal string 'restart', any expression resulting in a valid label name would have served.

See also: errortext

Name:	SOURCELINE
Type:	built-in function
Format:	n = SOURCELINE()
	s = SOURCELINE(num)
Description:	read source of currently-executing script

Script example:

```
/* SOURCELINE demo */
say "This script has" sourceline() "lines."
say "Line 3 reads thus:"sourceline(3)
```

Example output:

```
This script has 3 lines.
Line 3 reads thus:   say "Line 3 reads thus:"sourceline(3)
```

Discussion: SOURCELINE with no arguments returns the number of lines in the current script (i.e. the one containing the call to SOURCELINE). With a numeric argument, which must lie between 1 and the number of lines in the script, inclusive, SOURCELINE returns the text of the indicated line. A typical use of this function is to display help information embedded—perhaps as comments—within the script. Such a use might look like this:

```
if upper(cmd) = 'HELP' then
   do i=3 to 8
      say sourceline(i)
      end
```

Name:	SPACE
Type:	built-in function
Format:	s = SPACE(str, [len], [pad])
Description:	set length of spaces between words

***Dialog* examples:**

```
->"["space("gene a  lo   gist")"]"
[genealogist]
->space("7 year old",1,"-")
7-year-old
->space('> + + + <',8,'-')
>--------+--------+--------+--------<
```

Discussion: SPACE always removes any leading or trailing space from its string argument. Embedded spaces are replaced by a fixed-length run of the optional *pad* character, which defaults to a space. The length of the run is given by the *len* argument, which defaults to 0 (eliminating all spaces from the string as in the first example).

See also: compress, strip, translate, trim

Name:	STATEF
Type:	support function
Format:	filestring = STATEF(pathname)
Description:	obtain information about a file or directory

Dialog examples:

```
->statef('s:')
DIR 0 0 ----RWED 4966 307 2987
->statef('s:startup-sequence')
FILE 1113 3 ----RWED 4962 98 559
->'['statef('ram:DoesNotExist')']'
[]
```

Discussion: The EXISTS function in the built-in library can confirm for you that a given file or directory exists, but for more detailed information about a filing system 'object', you need STATEF, which takes an AmigaDOS pathname as its single argument, and returns a string with this structure:

```
type size blk bits day min tick com
```

The meaning of these names is:

type	Type of the object, either 'FILE' or 'DIR'
size	The size of the object in bytes (always 0 for a directory)
blk	The size of the object in disk blocks
bits	The protection bits for the file or directory
day	The creation date, in days since January 1, 1988
min	The creation time, in minutes from midnight
tick	The creation time, in ticks (1/50 seconds) within the minute
com	The file note, if any, for the file or directory

Essentially this information is the same as that returned by the AmigaDOS List command, but without tidy formatting, and with the date still in the 'raw' form used internally by AmigaDOS. This may be fine: if you just want some of the information you can split it up easily like this:

```
parse value statef(name) with type size blk bits day min tick com
```

where 'name' is the file or directory of interest. If you wanted to transform the STATEF information into a package as tidy as that returned by List, though, you could continue after the PARSE instruction like this:

```
  if type='DIR' then
     size = 'Dir'
/* the following line should be entered on one line */
  say left(name,30) right(size,7) bitor(bits)
timestamp(day,min,tick)

  if com ~= '' then
     say ':' com
  exit

  /* timestamp - turn days/minutes/ticks returned by STATEF into
     the normal human-readable date format (dd-mmm-yy hh:mm:ss).
  */
  timestamp: procedure
     parse arg d, m, t
     dt = space(date('n',d,'i'),1,'-')
     tm = right(m%60,2,0)':'right(m//60,2,0)':'right(t%50,2,0)
     return dt tm
```

See also: exists, showdir

Name:	STORAGE
Type:	built-in function
Format:	n = STORAGE()
	s = STORAGE(addr,[str],[len],[pad])
Description:	copy a string to a memory area

Script example:

```
/* Demo script using STORAGE */
say "Available memory:" storage() "bytes."

/* Allocate 48-byte buffer */
buf = getspace(48)

/* Copy string to buffer, ignore old contents */
call storage(buf, copies("Repetition! ", 4))

/* Copy string to buffer, echo old contents */
say storage(buf, "Star-padded string", 48, "*");

/* Free buffer (not strictly needed - see FREESPACE) */
call freespace(buf, 100)
```

Discussion: Like EXPORT, STORAGE provides a method of writing data into any memory location from within an ARexx script. STORAGE in fact behaves identically to EXPORT except for two small points:

1) When called with no arguments, STORAGE returns the amount of free memory in the system, in bytes (as in the example). EXPORT does not support this feature.

2) Whereas EXPORT returns the number of bytes copied to the destination address, STORAGE returns the previous contents of those same bytes as a string.

See also: export, import, getspace, freespace, allocmem, freemem

Name:	STRIP
Type:	built-in function
Format:	s = STRIP(str, [mode], [list])
Description:	strip leading and/or trailing characters from string

***Dialog* examples:**

```
->'['strip('   repaper   ')']'
[repaper]
->'['strip('   repaper   ',1')']'
[repaper   ]
->'['strip('+++repaper+++','t','+')']'
[+++repaper]
->'['strip('   repaper   ',,' r')']'
[epape]
->'['strip('   repaper   ',,' re')']'
[pap]
->'['strip('   repaper   ',,' rep')']'
[a]
->'['strip('   repaper   ',,' repa')']'
[]
```

Discussion: The primary purpose of STRIP is to remove leading and trailing blanks from a string. This is reflected in the default behavior shown in the first example, in which only the first argument—the string itself—is given. The *mode* argument is one of:

'B' ('Both')	default: both leading and trailing spaces
'L' ('Leading')	leading spaces only
'T' ('Trailing')	trailing spaces only

The *list* argument allows characters other than spaces to be stripped from the ends of the string. This argument should be regarded as a list rather than an ordered string: any characters in it, regardless of the order in which they occur, are stripped from the ends of the main argument string.

See also: compress, space, translate, trim

Name:	SUBSTR
Type:	built-in function
Format:	s = SUBSTR(str, start, [len],[pad])
Description:	extract substring from string

***Dialog* examples:**

```
->substr('sparrowhawk',8)
hawk
->substr('sparrowhawk',3,5)
arrow
->substr('sparrowhawk',9,5,'!')
awk!!
```

Discussion: SUBSTR's first argument is the string from which a substring will be extracted, and the second is the character position, counting from 1, at which the substring begins. The optional *len* argument is the length of the substring; it defaults to the remainder of the string, as in the first example. If the argument string is too short to allow a substring of the requested length, the substring is padded on the right with spaces (the default) or the *pad* character.

Name:	SUBWORD
Type:	built-in function
Format:	s = SUBWORD(str, start, [count])
Description:	extract words from string

***Dialog* examples:**

```
->subword('Never have I lied to you, my friends!',3)
I lied to you, my friends!
->subword('Why would I lie when I want your vote?',3,5)
I lie when I want
```

Discussion: SUBWORD's first argument is the string from which a substring will be extracted, and the second is the word, counting from 1, at which the substring begins. The optional *count* argument is the number of words in the substring; the default is the remainder of the string, as in the first example. The string returned by SUBWORD never has leading or trailing spaces.

See also: delword, find, word, wordindex, wordlength, words

Name:	SYMBOL
Type:	built-in function
Format:	s = SUBWORD(str)
Description:	determine if a string is a valid ARexx symbol

Script example:

```
/* The SYMBOL function */
say symbol('HORSE')   /* output: LIT */
say symbol('COW')     /* output: LIT */
say symbol('Z.DOG')   /* output: LIT */
say symbol('No-no')   /* output: BAD */

horse = 'COW'
cow   = 12
z.dog = 'Good old rover'

say symbol('HORSE')   /* output: VAR */
say symbol('horse')   /* output: VAR */
say symbol('HORSE.') /* output: LIT */
say symbol('COW')     /* output: VAR */
say symbol('Z.DOG')   /* output: VAR */
say symbol('Z.')      /* output: LIT */

say symbol(horse)     /* output: VAR */
say symbol(cow)       /* output: LIT */
say symbol(z.dog)     /* output: BAD */
```

Discussion: SYMBOL takes a single string argument, and returns one of three result strings:

'LIT' The given string is either a 'fixed' symbol (a literal number) or a valid but unused variable name.

'VAR' The string is a variable name to which a value has been assigned.

'BAD' The string is neither a literal number nor a valid variable name.

The symbol 'No-no' in the example script could not be a variable name because the hyphen is not allowed in identifiers. The compound variable Z.DOG, has the value 'Good old rover'; that cannot be a variable name because of the embedded spaces.

Notice from the example script that an initialized compound variable will rank as 'VAR' even though the corresponding stem is a 'LIT'. The reverse is not true, since if the stem has been initialized, all compounds from that stem are initialized by inheritance.

See also: value

Name:	THEN
Type:	instruction
Format:	THEN [;] instruction
Description:	execute dependent instruction if preceding expression was True

Discussion: THEN is valid in two contexts only:

```
IF <expression> THEN
```

and

```
WHEN <expression> THEN
```

and the latter can itself only occur within a SELECT instruction. In either case, the instruction dependent on THEN can be either a simple instruction consisting of a single line of code, or a compound instruction such as an IF-THEN-ELSE, a SELECT-WHEN-OTHERWISE, or a DO-END block or loop.

See also: if, else, select, when, otherwise, do, end

Name:	TIME
Type:	built-in function
Format:	t = TIME([mode])
Description:	find current or elapsed time

***Dialog* example:**

```
->time()
15:16:08
->time('n')
15:16:08
->time('c')
3:16PM
->time('h')
15
->time('m')
916
->time('s')
54968
->time('e')
0.00
->time('e')          /* Almost a minute having passed. */
56.46
->time('r') time('e')
                     /* About half a minute later...   */
87.36 0.00
```

Discussion: ARexx's TIME function deals in two kinds of time: the time of day, and elapsed time. The two mode arguments 'e' and 'r' concern elapsed time; the others concern the time of day.

The details of the modes, which may be abbreviated to a single letter as usual, are:

Mode	Time information returned...
Civil	The time in the form *h:mmAM* or *h:mmPM*. The hour part of the time is a number between 1 and 12, and is not padded with a leading zero.
Elapsed	The elapsed time in the form *s.cc*, where *s* is an unformatted number giving elapsed seconds, and *cc* is hundredths of seconds in two digits. The elapsed time clock does not start running until the first time either this mode or the 'r' mode is used within a script. On that initial call, it returns '0.00'. The clock runs continuously thereafter, and may be read with further 'e' calls, or read and simultaneously reset to zero with the 'r' mode.
Hours	The number of completed hours since midnight of the current day as an unformatted numeric string from 0 through 23.
Minutes	The number of completed minutes since midnight of the current day as an unformatted numeric string from 0 through 1439.
Normal	The current time in 24-hour *hh:mm:ss* format. All three fields will always contain 2 digits, padded with a leading zero if necessary. This is the default format (see the first example).
Reset	The elapsed time in form *s.cc* (see 'Elapsed' mode above). The elapsed time clock is simultaneously reset to '0.00'.
Seconds	The number of completed seconds since midnight of the current day as an unformatted numeric string from 0 through 85399.

The TIME function has, along with DATE, the special property that multiple calls within a single instruction will be mutually consistent: neither the calendar nor the clock will (be observed to) advance between calls.

See also: date

Name:	TRACE
Type:	instruction
Format:	TRACE mode
	TRACE [VALUE] expr
	TRACE num
Description:	set tracing mode

Discussion: The TRACE facility is the most powerful debugging tool available to the ARexx programmer. It can be accessed in 3 different ways:

1) the TRACE function (covered next in this Reference Section)
2) the TS and TE support programs (covered in an appendix)
3) the TRACE instruction

Tracing means the display of information about the state of a running program, such as the number of the line currently being executed, the values of variables and expressions, and the results returned by external commands. There are three options for the display of trace information:

1) the standard output (intermixed with other script output)
2) the file identified as 'STDERR' in the script, if any
3) the 'global tracing console' (see the TCO and TCC support programs)

Sometimes it is sufficient simply to passively view a tracing display as a script executes. That is one of the two tracing styles supported by ARexx. The other, more painstaking and more powerful, is 'interactive tracing', in which script execution may be suspended after (almost) every instruction so that the programmer can not only study the trace output at leisure, but also execute ARexx instructions entered from the keyboard. This makes it possible to do things like examine and even modify variables being used in the script, or to switch trace options on the fly.

It probably doesn't occur to most ARexx programmers that their programs are traced by default. But consider this Shell session:

```
Shell> rx "options failat 5; zot; say 'Okay!'
  1 *-* zot;
+++ Command returned 5
Okay!
```

and now this one:

```
Shell> rx "trace 'Off'; options failat 5; zot; say 'Okay!'
Okay!
```

If you didn't initially recognize the two extra lines in the first session as trace output, that's understandable: an ARexx programmer sees lines like that so often that they seem to be coming from the language core, but really they do not. It is simply that the 'Normal' trace option is set by default, showing the line number and return code for commands whose return codes exceed the failure threshold.

From the initial setting of 'Normal', the trace mode may be set to any of the following (as usual, only the first letter, in upper or lower case, is needed to select a particular mode):

Mode	Meaning

All	List all clauses as they execute. Often 'clause' is synonymous with 'line', but not always. For instance, the line:
	`if a=3 then say "Hello"`
	contains two clauses that will appear separately in the trace output:
	`1) if a=3 then`
	`2) say "Hello"`

Background	This mode, designed for fully tested scripts (or portions of scripts), is the same as 'Off' mode, except that it cannot be overridden by the TS support program—which puts every ARexx script into interactive trace mode.
Commands	Any clause that sends a command to either the primary or alternate host is displayed, as is the command actually sent, and the return code, if it is non-zero.
Errors	Any clause that sends a command to either the primary or alternate host, and the corresponding return code, are displayed if the code is non-zero.
Intermediates	This is the most detailed tracing mode. Every clause is traced, and the intermediate results of expression evaluation—variable values, function results and so on—are also displayed. This mode can generate a lot of output, so it's best used only over short stretches

of code where you really have to know what's going on.

Labels — Every time control reaches a label clause, the clause is displayed, regardless of whether the label was arrived at by a jump (arising from one of the forms of the SIGNAL instruction) or in due sequence.

Normal — This is the default trace mode, discussed above. It is identical to the 'Errors' mode, except that the error must be not just non-zero but equal or exceed the current FAILAT level as inherited from the invoker of the script or set in the script with OPTIONS FAILAT.

Off — In this mode, no tracing takes place. Unlike 'Background' mode, however, 'Off' mode can be overridden by the TS support program.

Results — The result of every expression evaluation is displayed.

Scan — This mode is the same as 'All', but has the additional property that the instructions in the script are not actually executed. The intention is that newly-written scripts—or parts of scripts—can be tested in safety until it is determined that they are syntactically correct, and that the flow of control meets the programmer's design.

Trace options may be set anywhere in a script, with lines like:

```
trace r      /* trace results */
```

The argument is ordinarily taken as a literal value, whether or not it is enclosed in quotes: the existence of a variable named 'r' is irrelevant to the above instruction. Use the VALUE sub-keyword if want the argument evaluated before being given to TRACE:

```
tracepref = 'Labels'
trace value tracepref
```

Two special characters may be used in combination with the mode argument to TRACE, and/or each other, or alone. They are the question mark, which turns interactive tracing on or off, and the exclamation mark, which toggles a feature known as 'command inhibition'.

Interactive tracing

The first time the question mark is used, interactive trace mode is turned on:

```
trace ?n     /* Normal mode, interactive */
```

The next time it is used, interactive mode is turned off again (though for reasons that will shortly be explained, this time either TRACE must be entered interactively or the TRACE function must be used instead of the instruction).

Interactive mode gives you a chance to reply from the keyboard to (almost) every report from the trace facility. If you are in 'Results' mode, for instance, you will get trace output every time an expression is evaluated (or is obtained from PARSE, ARG and PULL), which is pretty often. And on each occasion, the execution of the script will be paused after the output and you will be prompted for input with '>+>'. When you see that prompt, you have 3 ways of responding:

1) press Return; the script will continue executing until the next pause point is reached.
2) press the '=' key and press Return; the clause that has just been traced will be executed again.
3) enter a line of ARexx instructions, and press Return; the line will be treated almost exactly as though it were being executed by INTERPRET at that point in the script. There are three differences:
   * A TRACE instruction encountered in the script during tracing will be ignored (the TRACE function works as usual); but TRACE will be respected if issued from the keyboard.
   * Errors in lines entered interactively don't cause the script to fail, as of course they do in the script itself.
   * If command inhibition mode is on (see below), commands given interactively are not inhibited.

There are a few instructions at which a trace will never pause, regardless of the mode. They are: CALL, DO, ELSE, IF, OTHERWISE and THEN.

Command inhibition

The first time the exclamation mark is used in a TRACE instruction (or function call), command inhibition mode is turned on:

```
trace !c     /* command mode, inhibition on */
```

The next time it is used, the mode is turned off. The effect of command inhibition is to suppress the execution of external commands. The expression that would be sent out as a command is evaluated as usual, but is not issued to the external host. A return

code of zero is assumed. Using command inhibition in combination with the 'Command' trace mode, as in the sample line above, lets the programmer see exactly what commands the script would generate if it were given the opportunity. With potentially destructive commands (such as the AmigaDOS Format and Delete commands), checking a script in this mode is a wise precaution.

Numeric TRACE arguments

Besides the variations already discussed, TRACE accepts both positive and negative numeric arguments, both of which apply to interactive tracing only. A positive number specifies the number of pauses in the interactive trace to skip. If, during an interactive trace, you enter:

```
>+> trace 10
```

the trace will continue as before, but will not pause again until the 10th opportunity.

A negative number as the argument to TRACE is taken as a 'suppression count'. The effect is to turn off tracing for the given number of clauses (ignoring the minus sign). Thus:

```
trace -20
```

suppresses the trace until the 20th clause from the present one.

Trace output

The trace output reflects the logic of the script by applying extra levels of indentation within control structures and internal functions. The trace output for a clause first shows the line number and the clause being traced. The symbol '*-*' identifies it as a traced script line:

```
5 *-* say center('HAPPY',57);
```

Depending on what trace mode is selected, a variety of other information may then be displayed. Each line of output is preceded by a special code corresponding to the type of information given on that line. The possible symbols are:

Code	Meaning
+++	Command or syntax error (you'll see this a lot)
>>>	Expression or parse result
>.>	Value assigned to placeholder in parsing
	The codes below are encountered only in 'i' mode:
>C>	Expanded form of compound name
>F>	Result of function
>O>	Result of dyadic operation (i.e. with 2 operands)
>P>	Result of prefix operation (i.e. with 1 operand)

>V> The contents of a variable
>L> A literal (constant) value

By default, trace output is sent to your standard output (the file whose identifier is 'STDOUT'). This can be messy and confusing if the script itself generates much output, so ARexx will alternatively use the file identified as 'STDERR' for trace output, if one has been opened in the script. For serious tracing, especially interactive tracing, this is often the best way to go. Put a line like the following into the script to prevent its output being intermixed with the trace.

```
call open('STDERR','con:0/0/640/100/Trace console')
```

Another alternative is to use the 'global tracing console', which can be opened with the TCO support program, and closed with TCC. When this console is open, trace output will be displayed in it from any ARexx scripts that do not have their own 'STDERR'.

See also: trace (function); TS, TE, TCO, TCC
(see Appendix 'Support Programs')

For a discussion of tracing techniques in debugging, refer to Chapter 11 ('Tracing and Signals').

Name:	TRACE
Type:	built-in function
Format:	t = TRACE([mode])
Description:	get/set tracing mode

Script example:

```
/* A script with tracing output */
say "(Now turning on trace('i').)"
oldtrace = trace('i') /* i = Intermediates */

say substr("blithe endearments",2*2,length("conclude")-one())

call trace(oldtrace)
say "(Tracing now turned off.)"

exit

one: procedure
     return 1
```

Example output:

```
(Now turning on trace('i').)
     >F> "N"
     >>> "N"
  6 *-* ;
  7 *-* say substr("blithe endearments",2*2,length("conclude")
```

```
      >L> "blithe endearments"
      >L> "2"
      >L> "2"
      >O> "4"
      >>> "4"
      >L> "conclude"
      >F> "8"
   14 *-* one:
   14 *-* procedure;
   15 *-* return 1;
        >L> "1"
      >F> "1"
      >O> "7"
      >>> "7"
      >F> "the end"
the end
  8 *-* ;
  9 *-* call trace(oldtrace);
      >V> "N"
(Tracing now turned off.)
```

Discussion: TRACE sets a tracing mode, and returns the previous one (so that it can be restored later on). Tracing under the new mode begins immediately upon evaluation of the TRACE function call. The TRACE function has the same capabilities as the TRACE instruction, except in a few details:

1) The previous mode is available only through the function
2) A numeric trace suppression argument is valid only for the instruction
3) Calls to the TRACE function in the script itself work during interactive tracing; the TRACE instruction does not.

The modes may be specified by the first character of the keywords in the following summary. The options are covered in detail under the TRACE instruction.

Mode	Trace action
All	Traces all clauses
Background	Like trace off, but can't be changed from outside
Commands	Traces commands and non-zero RC
Errors	Traces commands that generate non-zero RC
Intermediates	Traces intermediate expression results
Labels	Traces label clauses (e.g. function calls)
Normal	Trace RC greater than or equal to FAILAT
Off	Turn tracing off
Results	Trace expression results
Scan	Trace all clauses, but don't execute them

Question marks preceding a trace option toggle interactive tracing; exclamation marks toggle command inhibition.

See also: trace (instruction)

Name:	TRANSLATE
Type:	built-in function
Format:	s = TRANSLATE(str)
	s = TRANSLATE(str,[output],[input],[pad])
Description:	translate *str* using *input* and *output* character tables

***Dialog* examples:**

```
->translate('unicef')
UNICEF
->translate('SUBTLETY',xrange('a','z'),xrange('A','Z'))
subtlety
->translate('uniformity',,xrange(),'*')
**********
->translate('plaintext',
reverse(xrange('a','z')),xrange('a','z'))
kozrmgvcg
```

Discussion: TRANSLATE remaps the input string (the first argument) by looking up each of its characters in an 'input table' and replacing it with the corresponding character from an 'output table'. If the output table is shorter than the input table, it is padded on the right with spaces (the default) or with the *pad* character (see third example).

If only the first argument is given, the string is translated using default tables that simply map all alphabetic characters to upper case and leave other characters unchanged. If more than one argument is given, the default tables are not used.

A null input table is allowed, as in:

```
->translate('hello','abc','')
hello
```

but will have no effect. A null output table is also allowed, and will convert all characters of the input string that also occur in the input table to the pad character.

See also: compress, space, strip, trim, xrange

Name:	TRIM
Type:	built-in function
Format:	s = TRIM(str)
Description:	Remove trailing blanks from string

***Dialog* example:**

```
->'['trim('   Trim it!   ')']'

[   Trim it!]
```

Discussion: TRIM is the tied first-place winner (along with LENGTH, REVERSE and UPPER) for easiest to grasp and use ARexx string function. Use it instead of STRIP for a slight gain in efficiency and readability.

See also: compress, space, strip, translate

Name:	TRUNC
Type:	built-in function
Format:	n = TRUNC(number,[places])
Description:	truncate *number* to *places* decimal places

***Dialog* example:**

```
->trunc(3.14159265359)
3
->trunc(3.14159,3)
3.141
->trunc(3.14159+.5e-3,3)
3.142
->trunc(3.14159+.5e-4,4)
3.1416
->trunc(3.14159,7)
3.1415900
->trunc(3.14159+.5e-7,7)
3.1415900
```

Discussion: TRUNC removes surplus digits or pads with zeros as needed to give its first numeric argument the number of decimal places specified in the second argument. The *places* argument defaults to 0, which has the effect of removing the fractional part of the number altogether. If given, the second argument must be a non-negative integer.

Since TRUNC truncates without rounding, it may be desirable to add .5 with the appropriate exponent to the number before calling the function, as in the second, third and fifth examples. This will achieve proper rounding.

If your script is making many calls to TRUNC, you should consider using the NUMERIC DIGITS instruction instead.

Name:	TYPEPKT
Type:	support function
Format:	cmd = TYPEPKT(pkt)
	count = TYPEPKT(pkt,'a')
	bool = TYPEPKT(pkt,mode)
Description:	extract information from a message packet

Code example:

```
/* This example is meant to be inserted into the
   simple command host given in the entry for GETARG,
   though you may prefer to do that as a thought
   experiment only. If you want to give it a try, put
   the lines that follow this comment immediately
   after these 2 lines from the GETARG example:

      if pkt ~= null() then do
         cmd = getarg(pkt)
   Now when you issue commands to the host, it will
   report on the contents of the packets received.
   Life is pretty dull for a command host, though, and
   the reports will all be the same. To spice up its
   life and yours, try adding it as a function host
   instead:
      ->addlib('SIMPLE_HOST',0)
      1

   Now make some function calls with bogus names, and
   dummy arguments, and let TYPEPKT show you some more
   details about how message packets work. Try lines
   like these:
      ->bogus(1,2,3)
      ->dummy(10,11,12,,14)
      ->weird('Hippopotamus')
...and see what you get. When you're done, remove the
   'function host' with:
      ->remlib('SIMPLE_HOST')
*/
if typepkt(pkt,'f') then
   pt = 'Function'
else if typepkt(pkt,'c') then
   pt = 'Command'
else
   pt = 'Unknown'

say "Packet command field contains: $"c2x(typepkt(pkt))
say "Number of arguments          :" typepkt(pkt,'a')
say "Method of invocation         :" pt
```

Discussion: An ARexx message packet contains two areas of particular interest to any program that sets up shop as a command or function host.

One area is the table of arguments within the packet. From 0 to 15 arguments may be present. A packet sent through the command interface, as with:

```
address simple_host "evaluate pi"
```

has zero arguments: the entire string is taken as the command, and the argument slots 1 to 15 are left empty.

A packet sent through the function interface, as with:

```
say CompareAndContrast('Charles Dickens','Stephen King')
```

may have as many comma-separated arguments as are provided up to the maximum of 15. As it happens, ARexx scripts cannot be operated as well-behaved function hosts without resort to trickery, so the function interface is—from that point of view—merely an interesting irrelevancy.

The other area of a packet that a host must consider is the 'command' field, which is organized as a 4-byte string in which each byte has an individual function. Counting the leftmost byte as number 3, and the rightmost as number 0, here is the meaning of each byte in the command:

Byte	Meaning
3	Specifies the type of packet: 1 for a command, 2 for a function. Other values are not valid in this context.
2	Modifier flags, including the flag specifying that a the originator of the message has requested a result (perhaps by using OPTIONS RESULTS). Hosts written in ARexx reply to their messages using the REPLY function however, which automatically handles any requirements arising from these flag settings.
1	This byte is presently unused.
0	This byte contains the number of arguments in the packet.

The command field from a message packet received by a command host might look like this:

```
01020000
```

meaning:

01	A command packet
02	Result requested
00	(Unused)
00	No arguments

The purpose of TYPEPKT is to extract all this information in a simple way. The first argument is always the address of the packet in 4-byte form, as returned by GETPKT. If no second argument is given, the returned value is the contents of the command field as a 4-byte string: just what we've been looking at.

If a second argument is provided, it must be one of three mode specifiers. As usual, only the first letter of the mode is significant. Here's what they mean:

Mode	Meaning
Arguments	Return the number of arguments in the packet
Command	Return 1 if the packet was sent as a command, else 0
Function	Return 1 if the packet was sent as a function, else 0

See also: getarg, getpkt, waitpkt, reply

Name:	UPPER
Type:	instruction
Format:	UPPER var [var ...]
Description:	convert contents of variables to upper case

Discussion: Although upper case text is not well-suited for reading, it simplifies pattern-matching problems considerably. Any 5-letter string in mixed case, for example, has 32 possible variations; these condense to a single version when the string is converted to upper case. Before checking any type of input keyword or password, therefore, or before building a compound variable name using a string of unknown case, it is normal to convert it to upper case. There are a number of ways to do this in ARexx, but the UPPER instruction is very convenient when you simply want to convert variables 'in place':

```
upper  firstname lastname occupation
```

See also: upper (function)

Name:	UPPER
Type:	built-in function
Format:	s = UPPER(str)
Description:	convert string to upper case

***Dialog* examples:**

```
->upper("Open this door!")
OPEN THIS DOOR!
->upper("C'est çà, chérie!")
C'EST ÇÀ, CHÉRIE!"
```

Discussion: The argument string is converted to upper case. As the second example indicates, characters from non-English alphabets are correctly converted.

Name:	VALUE
Type:	built-in function
Format:	val = VALUE(str)
Description:	treating str as an ARexx symbol, return its value

Script example:

```
/* VALUE */
groucho = 'chico'
chico   = 'harpo'
harpo   = 'groucho'

say value('groucho')
say value(groucho)
say value(value('groucho'))
say value(value(groucho))
say value(value(value(reverse('ohcuorg'))))
say value(value(value(groucho)))
```

Example output:

```
chico
harpo
harpo
groucho
groucho
chico
```

Discussion: VALUE evaluates its single argument, then treats the resulting string as an ARexx symbol. Since in ARexx the term 'symbol' is applied to numeric constants as well as to variable names, this is a valid (though trivial) use of VALUE:

```
->value(12+34.5)
46.5
```

Usually, though, the argument string evaluates to a variable name, and it is the value of that variable that is returned. In the first call to VALUE in the example script, for instance, the argument string evaluates to 'groucho'. If there were no variable of that name, the result of treating it as a symbol would be the string 'GROUCHO', but there is such a variable, so VALUE returns its contents, 'chico'.

In the second call, evaluating the argument string means finding the contents of the 'groucho' variable, which is 'chico', so it is just as though we had said:

```
say value('chico')
```

By analogy with the first call, we can see why this results in the value 'harpo'.

In the third call, the outer VALUE has as its argument the expression:

```
value('groucho')
```

which we have seen evaluates to 'chico'. Therefore it is again as though we had typed:

```
say value('chico')
```

and the result is again 'harpo'.

From this point you should be able to work through the rest of the example script and account for the output it produces.

For a less frivolous use of VALUE, suppose you were writing a script whose behavior is controlled by the user's mode selection, the mode being represented by a single letter, say 'D', 'H' or 'T', stored in a string called 'mode'. Each mode uses a different set of parallel compound variables, whose stems are called, let us say, 'D_INFO', 'H_INFO' and 'T_INFO'. One way to handle this would be with SELECT instructions, or IF-ELSE chains, thus:

```
if mode='D' then
   info = D_INFO.1.WEST
else if mode='H' then
   info = H_INFO.1.WEST
else if mode='T' then
   info = T_INFO.1.WEST
```

With VALUE, the same thing could be done more concisely, and independently of the number of modes that need to be supported:

```
info = value(mode"_INFO.1.WEST")
```

See also: symbol

Name:	VERIFY
Type:	built-in function
Format:	n = VERIFY(str,list,[match],[start])
Description:	determine where characters in *list* occur in *str*

***Dialog* examples:**

```
->verify('Mississippi','Mis')
9
->verify('Mississippi','p')
1
->verify('Mississippi','Mis','m')
1
->verify('Mississippi','p','m')
9
->verify('Two try tic-tac-toe',' -','m')
4
->verify('Try tic-tac-toe',' -','m',5)
8
```

Discussion: VERIFY's first argument is a string whose characters are checked to see whether any occur in *list*. If the optional *match* argument is 'm', VERIFY returns the position in *str* of the first (leftmost) character that occurs also occurs in *list*. If the *match* argument is missing, or is not 'm', VERIFY returns the position in *str* of the first character that does not occur in *list*.

The optional *start* argument specifies a start position in the *str*, characters to the left of the start position being ignored. The default start is 1, the first character in *str*.

Name:	WAITPKT
Type:	support function
Format:	bool = WAITPKT(portname)
Description:	wait for a message packet to arrive at a port

Discussion: When a script has opened a message port (with OPENPORT) and is waiting for message packets to arrive, it should call WAITPKT with the portname as its sole argument. The script's task will then go to sleep until a message arrives, leaving other tasks in the system to run without interference.

WAITPKT's boolean return can be checked to ensure that a message really has arrived, but it is probably better to make that determination

by looking at the message itself: by checking that the packet returned by GETPKT is non-null.

Once the message has been examined, acted upon, and replied, the script will normally loop back to the WAITPKT.

See also: closeport, openport, getarg, getpkt, waitpkt, reply

For an example of a script that uses WAITPKT, see the tiny command host script listed under GETARG.

Name:	WHEN
Type:	instruction
Format:	WHEN test [;] THEN [;] instruction (NB: 'instruction' here means either a single instruction ending at the end of a line, or a compound instruction such as a DO-END block or another IF instruction.)
Description:	introduce an alternative for SELECT

Discussion: The role of WHEN in a SELECT instruction is essentially identical to that of ELSE IF in an IF-THEN-ELSE. Apart from the keyword itself, the syntax of WHEN-THEN is the same as that of IF-THEN. See the entry for SELECT in this Reference Section for further information.

See also: if, then, else, otherwise

Name:	WORD
Type:	built-in function
Format:	s = WORD(str,n)
Description:	extract a given word from *str*

***Dialog* examples:**

```
->'['word("January February March April",3)']'
[March]
->'['word("January February March April",5)']'
[]
```

Discussion: Word *n* is extracted from *str*. The returned string does not contain any blanks. If the specified word does not exist, as in the second example, the empty string is returned.

See also: delword, find, subword, wordindex, wordlength, words

Name:	WORDINDEX
Type:	built-in function
Format:	n = WORDINDEX(str,n)
Description:	determine character position of start of word *n*

***Dialog* examples:**

```
->wordindex(" January February March April",1)
2
->wordindex(" January February March April",3)
19
->wordindex(" January February March April",5)
0
```

Discussion: WORDINDEX returns the character position of a given word within a string. The string is the first argument, and the word of interest is given *n*, an integer greater than zero. The returned character position counts from 1 as the leftmost character of *str*. If the word does not exist, as in the third example, the empty string is returned.

See also: delword, find, subword, word, wordlength, words

Name:	WORDLENGTH
Type:	built-in function
Format:	n = WORDLENGTH(str,n)
Description:	determine length of one word in *str*

***Dialog* examples:**

```
->wordlength("irremediably indecipherable incunabula",2)
14
->wordlength("none too many",5)

0
```

Discussion: WORDLENGTH returns the length in characters of the specified word in the string argument, counting the leftmost word as 1. As usual in ARexx word functions, a word is defined as any sequence of non-space characters bounded by space characters or by the ends of the string. WORDLENGTH returns 0 if the string has fewer than *n* words.

See also: delword, find, subword, word, wordindex, words

Name:	WORDS
Type:	built-in function
Format:	n = WORDS(str)
Description:	return number of words in *str*

Dialog examples:

```
->words("one two three")
3
->words("      .      ")
1
```

Discussion: WORDS returns the number of words in *str*. As usual in ARexx word functions, a word is defined as any sequence of non-space characters bounded by space characters or by the ends of the string.

See also: word, find, subword, word, wordindex, wordlength

Name:	WRITECH
Type:	built-in function
Format:	n = WRITECH(file, str)
Description:	write *str* to *file*, return count written

Dialog examples:

```
->writech('STDOUT','')
0
->writech('STDOUT','Count=')
Count=6
->writech('STDOUT','012345678')
0123456789
```

Discussion: WRITECH writes a *str* to the open *file*. The number of characters actually written is returned as the result. If this is less than the length of the *str*, a write error of some kind has occurred.

Unlike WRITELN, WRITECH does not add a linefeed character to the string, which is why the example strings written to the standard output file 'STDOUT'—generally a console window—are followed immediately on the same line by the function result (0, 6, 9).

An important use of WRITECH is writing 'binary'—that is, non-textual—information to a file. Here is a function that sets the given pen color (use 1, 2 or 3) for subsequent console output. Since the only purpose of the write in this case is to give a special instruction to the console, the addition of a linefeed would certainly be unwanted:

```
/* ConColor - set console color to 1, 2 or 3
   Sends ESC-[3?m to console, where ? is 1, 2 or 3

   Usage: call ConColor(2)
*/
ConColor: procedure
   if arg(1)>=1 & arg(1)<=3 then
      call writech('STDOUT','1b'x'[3'arg(1)'m')
   return
```

See also: readch, readln, writeln, eof

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	WRITELN
Type:	built-in function
Format:	n = WRITELN(file, str)
Description:	write *str* plus linefeed to *file*, return count written

Script example:

```
/* WRITELN demo - split word into characters */
wd = 'bandolier'
do i=1 to length(wd)
   call writeln('STDOUT',copies(' ',i-1)substr(wd, i, 1))
   end
```

Discussion: WRITELN writes *str*, with an appended linefeed, to the open *file*. The number of characters actually written, including the linefeed, is returned as the result. If this is less than the length of the string plus 1, a write error of some kind has occurred.

See also: readch, writech, readln, eof

Chapter 9 (File Input and Output) goes into some detail on all the ARexx file functions, and should be read for an introduction to and overview of file handling in ARexx.

Name:	X2C
Type:	built-in function
Format:	s = X2C(hex)
Description:	return character equivalent of *hex* string

Dialog examples:

```
->x2c(41 61 42 62 43 63)
AaBbCc
->x2c(333333)
333
->x2c(random(61,69)random(61,69))
eh
```

Discussion: As explained in Chapter 5—Other String Forms, character strings can equivalently be represented as a string of hexadecimal digits (0-9, and A-F in either upper or lower case). ARexx lets you use such hexadecimal strings directly, for example:

```
->'5a6f74'x
Zot
```

Note that the hexadecimal string is completely different from this string of mixed numeric and alphabetic characters:

```
->'5a6f74'
5a6f74
```

Appending an 'X' to a string to make it hexadecimal works only for 'literal' (quoted) strings. The X2C function lets you build the hexadecimal string as an expression, rather than having to give it literally. Otherwise, the conversion is exactly the same, and the same rules apply as to the literal strings: only the normal hexadecimal digits are allowed, with optional blanks at byte boundaries (every 2 digits, counting from the right).

See also: c2x

Name:	X2D
Type:	built-in function
Format:	n = X2D(hex,[n])
Description:	convert *hex* string to decimal

Dialog examples:

```
->x2d(234)
564
->x2d('ffff')
65535
->x2d(fffffffc)
-4
->x2d(976f45,2)
69
```

Discussion: The hexadecimal number is converted to a number. If given, *n* specifies the number of digits of *hex* to be used in the conversion. In the fourth example, for instance, only the two rightmost characters, 45, from *hex* are used in arriving at the decimal equivalent, 69.

See also: d2x

Name:	XRANGE
Type:	built-in function
Format:	s = XRANGE([c1],[c2])
Description:	build character string with consecutive ASCII values

***Dialog* examples:**

```
->length(xrange())
256
->xrange('a','e')
abcde
->xrange('ant','elephant')
abcde
->xrange('0','H')
0123456789:;<=>?@ABCDEFGH
```

Discussion: All characters, including those—the control characters—that have no printable form, correspond to values between 0 and 255 inclusive. The ordering of the characters between these limits may be found from the ASCII table at the back of this book. XRANGE returns a string consisting of all the characters between those given as its arguments, inclusive. The default value of *c1* is '00'x—zero. This is not the same as the character zero, whose ASCII code is '30'x. The default value of *c2* is 'ff'x—255, the highest value that will fit in a 1-byte character cell. If the arguments are given in high-low rather than low-high order, the range 'wraps around' so that the values at the end of the range are smaller than those at the beginning.

One use of XRANGE is preparing translation tables for TRANSLATE:

```
/* Translate upper case to lower case */
->translate('ALL UPPER',xrange('a','z'),xrange('A','Z'))
all upper
```

See also: translate

Appendix A
Using a Text Editor

ARexx scripts and macros are usually created with the help of a *text editor,* a type of program that in some ways resembles a word processor but is specialized for entering certain kinds of text, including computer programs.

Perhaps the biggest difference between a text editor and a word processor is that the former considers a document to be a collection of *lines,* while the latter considers it to be a collection of *paragraphs.* Another difference is in the way the text is stored in files on disk: a text editor stores the document as you see it on the screen, as a sequence of characters separated by spaces and linefeeds; whereas most word processors also save non-textual data such as formatting instructions and printer set-up information along with the actual text.

Amiga text editors

If you are familiar with AmigaDos, you may know that two text editors - called *ed* and *memacs* - are supplied with the system software. (There is actually a third, *edit,* but it is not suitable for the task at hand.) Either of these programs is adequate for preparing ARexx scripts. *Ed* is easier to learn than *memacs,* but less powerful: you may want to try both before deciding which is for you. The full instructions for these programs are contained in the system documentation that came with your Amiga. *Ed* has been substantially improved for AmigaDOS 2.0, with the addition of menus, an ARexx interface, and an enhanced command set.

If you write a lot of ARexx scripts (or other programs), you may find it worthwhile to invest in one of the several outstanding commercial text editors available for the Amiga, such as *Uedit, TurboText* and *CygnusEd.*

All text editors have certain essential features in common. Whichever one you choose, learn at the outset from the supplied documentation how to perform these fundamental operations:

File operations

- load a document/script
- create a new document/script
- save a document/script
- save under a new name

Cursor movements

- cursor left and right
- move to start or end line
- move up or down a 'page'
- move to top or bottom of document

Basic editing

- enter text
- indent text (with TAB key)
- mark a range of text
- copy or cut a marked range
- cut a line
- paste copied or cut text
- search for given text

Appendix B
ARexx Support Software

The ARexx package includes a number of files, some of which - like *rexxmast* and *rexxsyslib.library* - are absolutely needed if you are going to use ARexx at all. Chapter 2 of this book describes the installation of these necessary files.

In this appendix we describe the ARexx 'support' software. Most of these are programs that are used as CLI commands, and the last - *rexxsupport.library* - is an ARexx function library.

You don't need all of these files to use ARexx, but we recommend you install them anyway: even the less-used ones will come in handy once in a while, and some - like the four programs relating to tracing - are practically indispensable during script development.

RXC - Terminate ARexx

This program is the opposite of *rexxmast*: it removes ARexx from your system until the next time *rexxmast* is run. RXC has no effect on currently executing ARexx scripts, but once it has been issued new scripts will not be able to launch. Then, when any presently executing scripts have finished running, ARexx will close down completely. The only reason for ever using RXC is to maximize your Amiga's free memory in a low-memory situation: removing ARexx may gain you something like 45,000 bytes. This is a modest amount considering all that ARexx does, but may at times be more than you can afford to spare.

HI - Halt ARexx scripts

Just as RXC negates *rexxmast*, so in a way does HI negate the RX program used for launching scripts. Running HI sends a 'halt signal' to all executing scripts. Unless a script has set a special trap for this signal, it will immediately stop executing. Even scripts that do trap the halt signal should generally respect the user's wishes and shut down as soon as possible after doing any necessary clean-up in their trap routine. The normal reason for calling HI is to halt a runaway script -

one that has got itself into an infinite loop through programming error. Here is the simplest runaway script:

```
/* Runaway! */
do forever
   end
```

Most infinite loops are a little harder to spot than this one, though, so it's a good idea to keep the HI program handy.

TS - Start interactive tracing

The fact that you can shut down a runaway script with the HI program may be very handy, but TS is even more powerful. When you run TS, any currently executing ARexx script is immediately forced into 'interactive trace results' mode, as though the instruction:

```
trace ?r
```

had been executed in the script. The script will pause after the next expression evaluation, and begin generating output. Now that you have control over the script again, you can examine variables, step through instructions one at a time, and execute instructions interactively until you have determined what went wrong. You can read about tracing in detail in Chapter 11 (Debugging, Tracing and Error Trapping), and in the Reference Section entries for the TRACE instruction and the TRACE built-in function.

Running TS affects not only presently-executing scripts, but also those executed subsequently (so you can run it beforehand to start a script tracing interactively from the outset, if you like). To cancel the global 'flag' that forces this trace mode, you must run the TE program (see below). You can, however, turn off interactive tracing during the trace, and return to normal execution by entering the following instruction interactively:

```
trace ?n
```

TE - Stop interactive tracing

Running this program clears the 'global tracing flag' set by the TS program, allowing current and future scripts to operate normally.

TCO - Open global tracing console

As explained in the Reference Section entry for the TRACE instruction (to which you can refer for fuller information), the ARexx trace facility normally directs its output to, and looks for its input from, the script's default console, but will use a file called 'STDERR' for these purposes if the script has opened one.

When you run TCO, a special window called the 'global tracing console' opens on the Workbench screen. While this window is open (until you close it with the TCC program, in other words), all tracing input/output for scripts that do *not* have a 'STDERR' file will be conducted through it. The advantage of using the global tracing console is that your script's ordinary I/O and tracing I/O do not get intermixed.

TCC - Close global tracing console

This program closes the 'global tracing console' opened by TCO.

RXSET - Set value for an ARexx 'clip'

As explained under the Reference Section entries for the built-in library functions GETCLIP and SETCLIP, ARexx maintains a global list (that is, a list accessible to all scripts), called the 'clip list', of names and associated values. The clip list is not a single-purpose resource, but a general one: you can use it however you like, or not use it at all. A typical use, though, is to set from the outside configuration options and other data that may change from one run of a script to the next. The script can use GETCLIP to read the current values, and modify its behavior accordingly.

RXSET lets you set or modify the value associated with a name on the clip list. For instance, having given this command from the CLI:

```
Shell> RXSET DATAFILE work:addresses.dat
```

you might then run an ARexx script which would open your data file with:

```
call open('df', getclip('DATAFILE'))
```

You can modify a name on the clip list simply by running RXSET with a new value string:

```
Shell> RXSET DATAFILE ram:addr.dat
```

Finally, to remove an entry from the clip list, leave off the value string altogether:

```
Shell> RXSET DATAFILE
```

WaitForPort - Wait/Check for message port

ARexx macros and AmigaDOS scripts can test for the existence of a function or command host, or of ARexx itself, by running the WaitForPort program and checking the return code. To interface with ARexx, every host must open a 'public message port' with a characteristic name to identify it as belonging to that host. This is the name you use in the ADDRESS instruction to send commands to a command host; it is the name by which a function host is known in the Library List (look under RXLIB below for more on that). And ARexx, of course, has its own message port named 'REXX'.

Let's say you want to write an ARexx script that will make use of an application program like SoftWood's *Electric Thesaurus*. A carefully-written script would probably begin with a check to verify that *ET* is actually running:

```
if ~show('p','EThes_1') then do
   address command 'run ET'
   end
```

The only difficulty now is that it is going to take a little while for *ET* to load and put up its message port even if the script is correct in assuming that the program is available. We could have the script wait for some set amount of time - a second or two say - then check again, and repeat that process until the port has appeared or we finally decide it never will. Or we could ask the user to press Return or give some similar indication after *ET* has loaded. Or we could run WaitForPort.

WaitForPort works in exactly the same way as the little polling loop we contemplated writing, going to sleep for a short time then checking for the requested port, repeatedly for up to 10 seconds. If it has not found the port by then it returns a code of 5. If it finds the port within the 10 second time-out period, however, it returns at once with a code of 0.

For an example of using WaitForPort, see the sample script for running *Electric Thesaurus* in Chapter 13.

RXLIB - add a name to the Library List

The Library List is the list of 'function libraries' and 'function hosts' currently available to ARexx in its efforts to locate functions that are neither internal to a script nor contained in the built-in library. The functions contained in a library (or host) are not accessible until its special name is added to the list. One way to add an entry to this list is to use the ADDLIB built-in function, to whose Reference Section entry you should refer for a fuller explanation of the Library List mechanism. The RXLIB program gives you an alternative which may be preferable for libraries you use often.

For instance, if you always use 'rexxsupport.library' and 'rexxmathlib.library', you could ensure that they are always available by adding these lines to your startup-sequence:

```
RXLIB rexxsupport.library 0 -30 0
RXLIB rexxmathlib.library 0 -30 0
```

The significance of the numeric arguments to these commands is also explained in the Reference Section under ADDLIB.

Simply adding a name to the Library List does not ensure that a particular library or host is actually available. To be located by ARexx, libraries must be filed in your system 'libs:' directory, and function hosts are normally embedded in executable programs that you must run. If the library or host is not available when ARexx goes looking for it, you will get the error message 'Host environment not found'.

You can get a list of the function libraries and function hosts presently on the library list by issuing the RXLIB command with no arguments.

rexxsupport.library - support functions

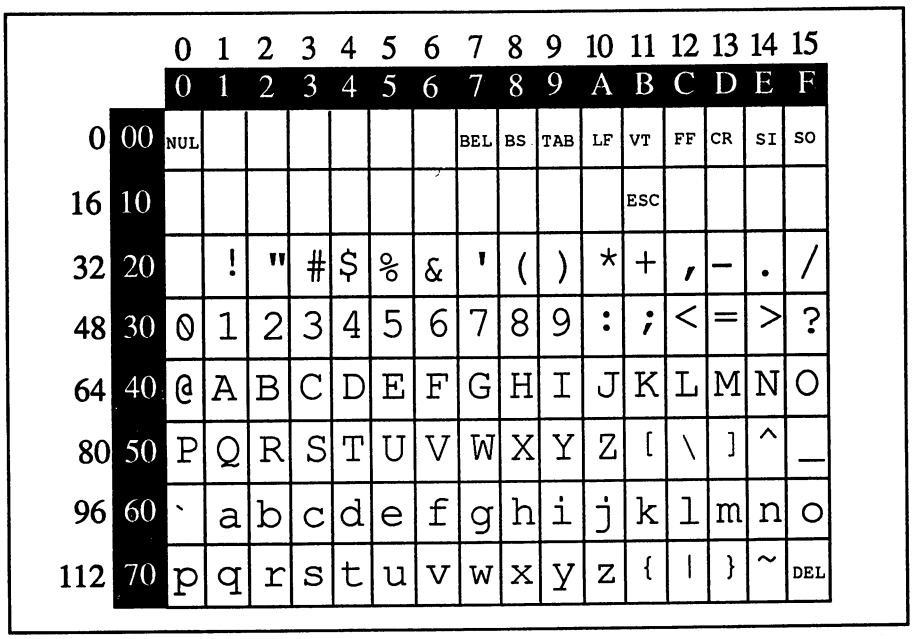
The file *rexxsupport.library* is not an executable program like the ones described above, but is a function library that provides additional functions to the ARexx language. It must be placed in your system 'libs:' directory.

Some of the more than 20 functions will be of value only to advanced users, but a few - like SHOWDIR and STATEF - are so generally useful that it is worth having this small library permanently available. Look

under RXLIB above for the appropriate line to add to your Startup-Sequence to achieve this, along with the associated theory.

To find out more about the contents of the library, check out the Reference Guide (at the start of the Reference Section) for functions whose name is accompanied by the code '[S]'.

Appendix C
ASCII Chart



		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
0	00	NUL							BEL	BS	TAB	LF	VT	FF	CR	SI	SO
16	10												ESC				
32	20		!	"	#	$	%	&	'	(	)	*	+	,	-	.	/
48	30	0	1	2	3	4	5	6	7	8	9	:	;	<	=	>	?
64	40	@	A	B	C	D	E	F	G	H	I	J	K	L	M	N	O
80	50	P	Q	R	S	T	U	V	W	X	Y	Z	[	\	]	^	_
96	60	`	a	b	c	d	e	f	g	h	i	j	k	l	m	n	o
112	70	p	q	r	s	t	u	v	w	x	y	z	{	|	}	~	DEL

Special characters:

NUL	Null character	BEL	Bell
BS	Backspace	LF	Line feed
VT	Vertical tab	FF	Form feed
CR	Carriage return	SI	Shift in
SO	Shift out (normal)	ESC	Escape

To determine the ASCII value, add the number in the left column to that in the top row. Decimal values appear in normal text and hexadecimal values appear in reverse text.

		0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
		0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
128	80					IND	NEL								RI		
144	90												CSI				
160	A0		¡	¢	£	¤	¥	¦	§	¨	©	ª	«	¬		®	¯
176	B0	°	±	²	³	´	µ	¶	·	¸	¹	º	»	¼	½	¾	¿
192	C0	À	Á	Â	Ã	Ä	Å	Æ	Ç	È	É	Ê	Ë	Ì	Í	Î	Ï
208	D0	Ð	Ñ	Ò	Ó	Ô	Õ	Ö	×	Ø	Ù	Ú	Û	Ü	Ý	Þ	ß
224	E0	à	á	â	ã	ä	å	æ	ç	è	é	ê	ë	ì	í	î	ï
240	F0	ð	ñ	ò	ó	ô	õ	ö	÷	ø	ù	ú	û	ü	ý	þ	ÿ

Special characters:

IND	Index (down line)	RI	Reverse index
NEL	Next line	CSI	Control sequence introducer

To determine the ASCII value, add the number in the left column to that in the top row. Decimal values appear in normal text and hexadecimal values appear in reverse text.

Define particular characters in ARexx with d2c(n), where n is the decimal sum, e.g. "A" = d2c(64+1).

Appendix D
Vendors and Products

This is a list of the suppliers of products mentioned in the book. It does not cover all of the many producers of ARexx-supporting software.

ASDG Incorporated
925 Stewart St.
Madison, WI 53713
(608) 273-6585

Art Department Professional

Gold Disk Inc.
P.O. Box 789, Streetsville
Mississauga, ON (Canada) L5M 2C2
(416) 602-4000

HyperBook, ShowMaker
Home Office Advantage

INOVAtronics, Inc.
8499 Greenville Avenue Suite 2098
Dallas, TX 75231
(214) 340-4991

CanDo, C.A.P.E 68k

New Horizons Software, Inc.
206 Wild Basin Road, Suite 109
Austin, TX 78746
(512) 328-6650

ProWrite 3.1

Geodesic Publications
P.O. Box 956068
Duluth, GA 30136
(404) 822-0566

IllumiLink

Gramma Software
17730-15th Ave. N.E., Suite 223
Seattle, WA 98155-3804
(206) 363-6417

FreD, Nag, Cal

Manx Software Systems
P.O. Box 55
Shrewsbury, NJ 07702
(201) 542-2121

Aztec C

NewTek Incorporated
115 W. Crane St.
Topeka, KS 66603
1-800-843-8934

Digi-Paint 3

Oxxi
P.O. Box 90309
Long Beach, CA 90809-0309
(213) 427-1227

A-Talk III, TurboText

Progressive Peripherals and Software
464 Kalamath St.
Denver, CO 80204
(303) 825-4144

Baud Bandit, Intro CAD Plus

Software Visions Inc.
P.O. Box 3319
Framingham, MA 01701
(508) 875-1238

Microfiche Filer Plus

William S. Hawes
P.O. Box 308
Maynard, MA 01754
(617) 568-8695

ARexx

Precision Software
8404 Sterling Street
Irving, TX 75063
1-800-562-9909

Superplan,
SuperBase Professional 4

SAS Institute Inc.
SAS Campus Drive
Cary, NC 27513-2414
(919) 677-8000

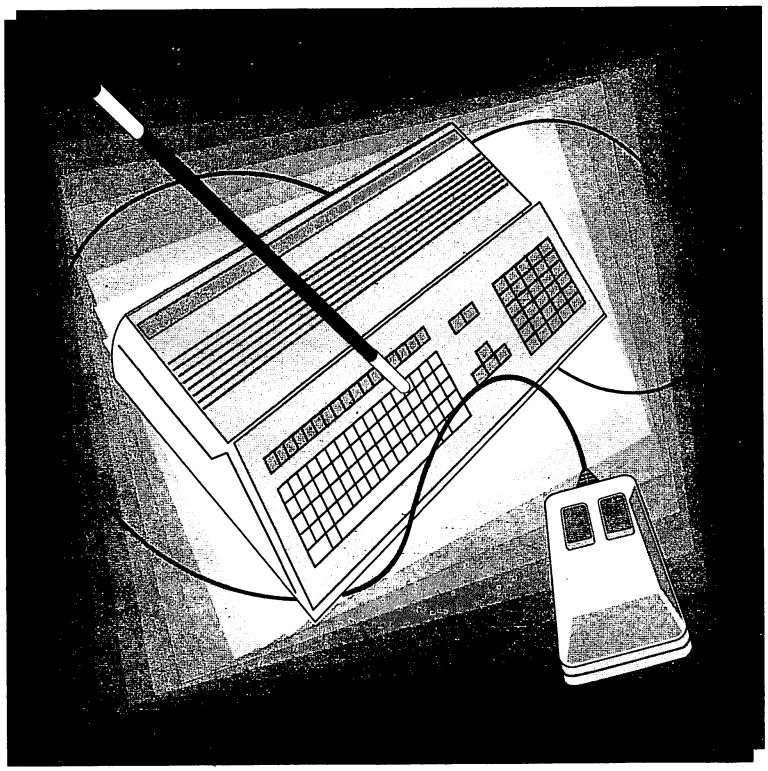
SAS/C Lattice C Development System

Softwood, Inc.
P.O. Box 50178
Phoenix, AZ 85076
(602) 431-9151

Electric Thesaurus,
Proper Grammar

Index

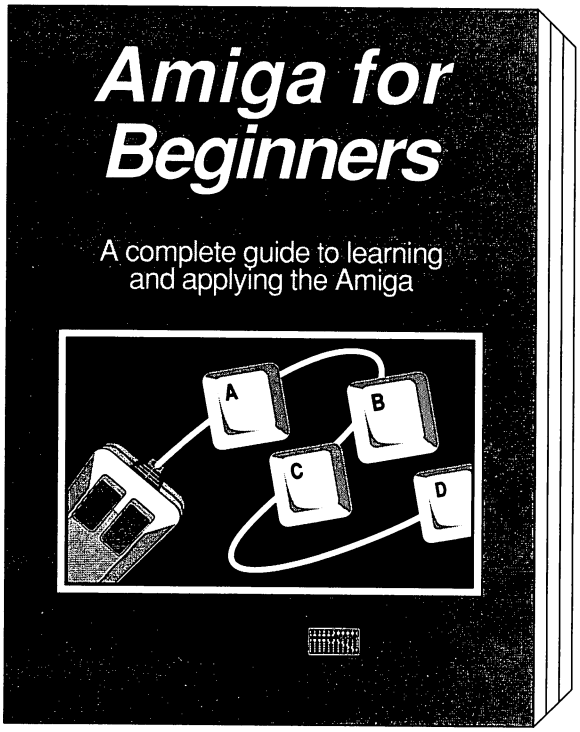
Abacus Amiga Catalog



Order Toll Free 1-800-451-4319

Amiga for Beginners

A perfect introductory book if you're a new or prospective Amiga owner. **Amiga for Beginners** introduces you to Intuition (the Amiga's graphic interface), the mouse, windows and the versatile CLI. This first volume in our Amiga series explains every practical aspect of the Amiga in plain English. Clear, step-by-step instructions for common Amiga tasks. **Amiga for Beginners** is all the info you need to get up and running.



Topics include:

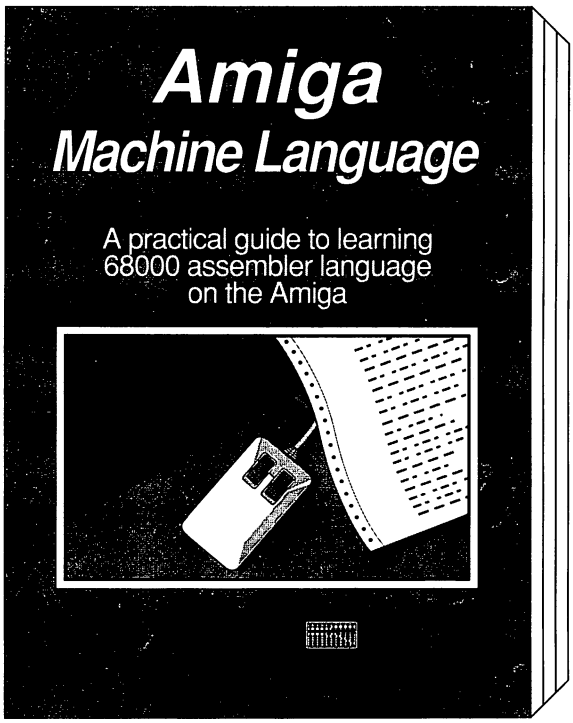
- Unpacking and connecting the Amiga components
- Starting up your Amiga
- Exploring the Extras disk
- Taking your first step in AmigaBASIC programming language
- AmigaDOS functions
- Customizing the Workbench
- Using the CLI to perform "housekeeping" chores
- First Aid, Keyword, Technical appendixes
- Glossary

Item #B021 ISBN 1-55755-021-2. Suggested retail price: $16.95

Companion Diskette not available for this book.

Amiga Machine Language

Amiga Machine Language introduces you to 68000 machine language programming presented in clear, easy to understand terms. If you're a beginner, the introduction eases you into programming right away. If you're an advanced programmer, you'll discover the hidden powers of your Amiga. Learn how to access the hardware registers, use the Amiga libraries, create gadgets, work with Intuition and more.



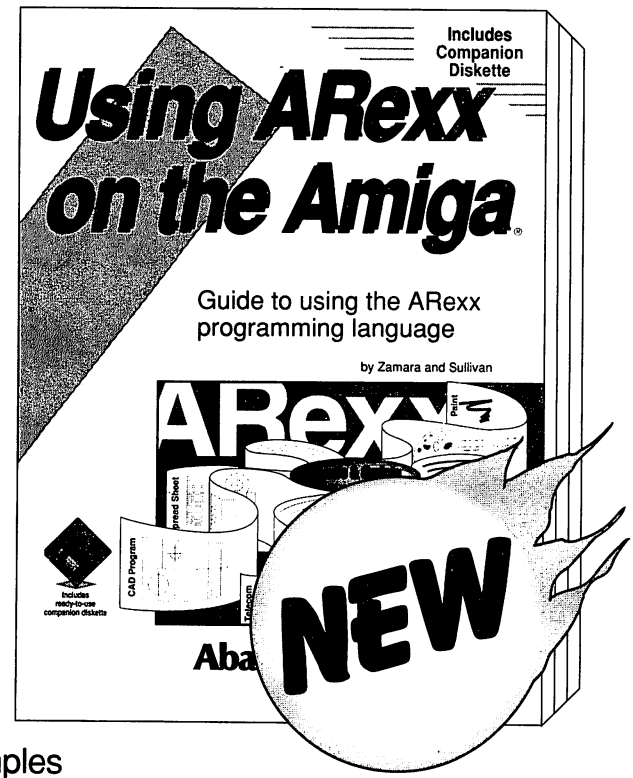
- 68000 microprocessor architecture
- 68000 address modes and instruction set
- Accessing RAM, operating system and multitasking capabilities
- Details the powerful Amiga libraries for access to AmigaDOS
- Simple number base conversions
- Menu programming explained
- Speech utility for remarkable human voice synthesis
- Complete Intuition demonstration program including Proportional, Boolean and String gadgets

Item #B025 ISBN 1-55755-025-5. Suggested retail price: $19.95
Companion Diskette available: *Contains every program listed in the book- complete, error free and ready to run! Saves you hours of typing in program listings. Available only from Abacus.* **Item #B025. $14.95**

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Using ARexx on the Amiga

Using ARexx on the Amiga is the most authoritative guide to using the popular ARexx programming language on the Amiga. It's filled with tutorials, examples, programming code and a complete reference section that you will use over and over again. **Using ARexx on the Amiga** is written for new users and advanced programmers of ARexx by noted Amiga experts Chris Zamara and Nick Sullivan.



Topics include:

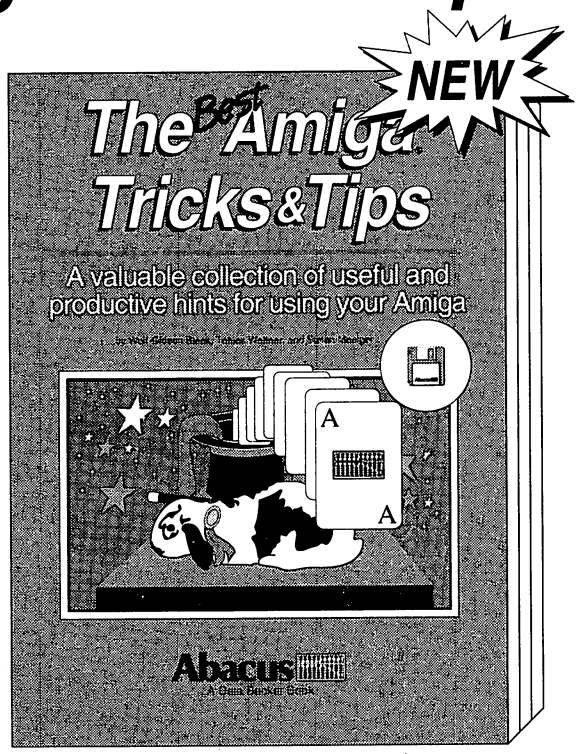
- What is Rexx/ARexx - a short history
- Thorough overview of all ARexx commands - with examples
- Useful ARexx macros for controlling software and devices
- How to access other Amiga applications with ARexx
- Detailed ARexx programming examples for beginners and advanced users
- Multi-tasking and inter-program communications
- Companion diskette included
- And much, much more!

Item #B114 ISBN 1-55755-114-6.
Suggested retail price: $34.95

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

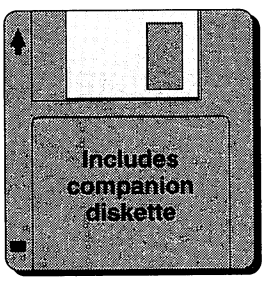
The Best Amiga Tricks & Tips

The Best Amiga Tricks & Tips is a great collection of Workbench, CLI and BASIC programming "quick-hitters", hints and application programs. You'll be able to make your programs more user-friendly with pull-down menus, sliders and tables. BASIC programmers will learn all about gadgets, windows, graphic fades, HAM mode, 3D graphics and more.



The Best Amiga Tricks & Tips includes a complete list of BASIC tokens and multitasking input and a fast and easy print routine. If you're an advanced programmer, you'll discover the hidden powers of your Amiga.

- Using the new AmigaDOS, Workbench and Preferences 1.3 and Release 20
- Tips on using the new utilities on Extras 1.3
- Customizing Kickstart for Amiga 1000 users
- Enhancing BASIC using ColorCycle and mouse sleeper
- Disabling FastRAM and disk drives
- Using the mount command
- Writing an Amiga virus killer program
- Changing type-styles
- Learn kernal commands
- BASIC benchmarks
- Disk drive operations and disk commands
- Learn machine language calls.



The Best Amiga Tricks & Tips includes companion disk. 410 pp.
Item # B107 ISBN 1-55755-107-3. Suggested retail price $29.95
Authors: Wolf-Gideon Bleek, Tobias Weltner, and Stefan Maelger.

See your local dealer or order TOLL FREE 1-800-451-4319 in US & Canada

Book/companion diskette packages:

- Save hours of typing in the ARexx script files from the book.
- Provide complete ARexx script file listings which help avoid printing and typing mistakes.
- The companion diskette contains many of the Arexx scripts listed in this book and more ARexx example scripts.

If you bought this book without a diskette, call us today to order an economical companion diskette and save yourself valuable time.

5370 52nd Street SE • Grand Rapids, MI 49512
Call 1-800-451-4319

Companion diskette contents

AREXX LISTINGS FROM THE BOOK to save you typing time and avoid errors when typing in the ARexx scripts.

SAMPLE AREXX programs to illustrate complex ARexx programming techniques!

USEFUL MACROS to control a variety of ARexx-compatible software applications.

AREXX-COMPATIBLE PROGRAMS to show off ARexx in action, including a simple paint program described in the book.

AREXX UTILITIES to make ARexx even more powerful and useful.

DOCUMENTATION for all utilities and programs included on the Companion diskette.

This Companion diskette will make you an ARexx expert fast. You can use all the ARexx examples on this disk immediately.

ARexx required, not included.

Includes Companion Diskette

Using ARexx on the Amiga®

Using ARexx on the Amiga is the most authoritative guide to using the popular ARexx programming language on the Amiga. It's filled with tutorials, examples, programming code and a complete reference section that you will use over and over again. **Using ARexx on the Amiga** is written for new users and advanced programmers of ARexx by noted Amiga experts Chris Zamara and Nick Sullivan.

The text of the book is sprinkled liberally with tutorials, examples and sample code. The power of ARexx is presented in a clear manner. All of the scripts of more than a few lines are also available on the accompanying disk.

About the authors

Nick Sullivan and Chris Zamara are freelance software developers, writers and editors. Their company, AHA! Software, has been developing for the Amiga since 1986; AHA's recent titles include TransWrite and HyperBook published by Gold Disk Inc. They also contribute to .info magazine as technical editors, producing a monthly Amiga technical section.

Chris and Nick have both been involved in Amiga journalism and software development for over five years, and were founding editors of the now-defunct Transactor for the Amiga, a technical magazine for Amiga programmers.

US $34.95/ CDN $44.95

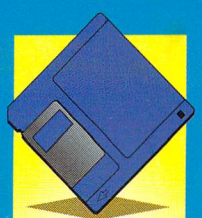
ISBN 1-55755-114-6

9 781557 551146

Guide to using the ARexx programming language

Topics include:

- What is Rexx/ARexx - a short history
- Thorough overview of all ARexx commands - with examples
- Useful ARexx macros for controlling software and devices
- How to access other Amiga applications with ARexx
- Detailed ARexx programming examples for beginners and advanced users
- Multitasking and inter-program communications
- And much, much more!



Includes ready-to-use companion diskette

Abacus

5370 52nd Street SE • Grand Rapids, MI 49512