Discover the potential of your Amiga's best-kept secret!

AMIGA SHOPPER PRESENTS

ARexx: Your Amiga's Built-in Turbocharger

Toby Simpson

- ★ Exploit the features Commodore don't tell you about
- ★ Write your own applications and utilities
- ★ Produce time-saving '*macros*' for top applications

If you have a Workbench 2 or 3 Amiga, you already have ARexx. Now all you need to know is how to use it...!

ARexx: Your Amiga's Built-in Turbocharger

Discover the potential of your Amiga's best-kept secret!

ARexx: Your Amiga's Built-in Turbocharger

Discover the potential of your Amiga's best-kept secret!

Toby Simpson

AMIGA SHOPPER

Author Toby Simpson

Book design Rod Lawton

Book production Ian Pemble

Cover design Frank Bartucca

First published in 1994 by Future Publishing, Beauford Court, 30 Monmouth Street, Bath, Avon BA1 2BW

ISBN 1-898275-09-2

Printed in the UK by Ashford Colour Press

Contents

Foreword

This is a book on ARexx, an easy-to-learn yet powerful computer language for the Amiga. Not only that, but for most Amiga owners ARexx is free! Every Amiga since the A500+ has been supplied with ARexx, which makes it an ideal first computer language to learn.

The aim of this book is to teach you how to program ARexx, to act as a handy reference guide to the language, and to give you some ideas about how ARexx can be put to work to make your life easier.

I'd like to say thanks to all my family and friends (and to the loads of other people who helped as well) and, in particular, to Rod Lawton of Future Publishing – for giving me the opportunity to write this book in the first place, and for his astonishing patience in waiting for me to finish it.

Toby Simpson, 18 July 1994

About the Author

Toby Simpson is 24, unfit, and lives in a little house in Cambridge, England. He writes computer games for a living. His hobbies include (rather sadly) computers and pinball, and he's not particularly good at the latter. Fortunately an increasing interest in caving and climbing offers a chance at redemption. He drinks abnormal amounts of single malt Scotch whisky, has never met aliens, and has yet to successfully grow any form of plant whatsoever. His favourite dinosaur is a Stegosaurus, and he's a whole bunch more interesting than he sounds. Probably.

Preface

'ARexx: Your Amiga's Built-in Turbocharger' was the most descriptive title we could think of. We could have called this book 'Introducing ARexx', or the 'ARexx Reference Guide', but this would miss the point.

And the point is that your Amiga (unless it's an old Workbench 1.3 model) comes with a little-documented but startlingly powerful program called ARexx which really can make your day to day Amiga jobs happen at twice the speed. Not by accelerating the CPU or doubling the RAM, but by letting you write scripts and programs that take the work away from you and hand it over to the machine and the software.

Now then, there's a big difference between what ARexx appears to be, and what it *is*...

It *appears* to be a rather complicated scripting language for high-end multi-tasking operations. It's buried deep in the system disks and hardly gets a mention in your Amiga's documentation.

What is *is*, is a powerful stand-alone programming language every bit as capable as BASIC and hardly more difficult to learn. That's the *big* secret. And this book shows you how to write programs in this excellent Amiga language.

What people also know about ARexx is that it's got something to do with multitasking, and only some software supports it. Well, it's a bit more useful than that. Imagine you had a permanent assistant that would (a) carry out odd jobs for you and (b) tell everyone else what you wanted them to do, and make sure they did it. That's what ARexx will do with your Amiga and its software.

Apart from its ability to produce stand-alone programs, ARexx can 'talk' to any applications that support it. It can make that application carry out a sequence of operations you would normally have to do manually – and it can do them a darned sight quicker. It can also 'talk' to more than one application at a time. For example, it can carry out a sequence of operations on a piece of artwork in one application, then import it into a DTP document running under another application. It could then print the document out if you wanted it to. That's just a simple example. The real potential is up to your own imagination.

Why do you need this book?

Commodore have a habit of making some very sound decisions – and some very strange ones. In this case, the sound decision was bundling ARexx with all Amigas from Workbench 2 on. The strange one was including documentation only on the high-end models. Which means that most Amiga owners have this fabulous untapped resource at their fingertips, and no way of knowing what to do with it.

Which is why we produced this book!

What the icons mean

Books are very big things and can easily turn into a grey mass of words that definitely contain that piece of information you want – if only you could find it.

We've picked out important pieces of information carefully, setting them in bold type and putting an icon in the margin alongside. These are those icons:

Warning! Speaks for itself, really. When you see this sign, do make sure you read the text alongside. It could save you money, save you time and even save your sanity.

Make a note. You'll see this icon alongside important details or information you're going to want to keep referring back to in future. This is our way of making it easy to find.

Top Tip. Another obvious one. Wherever there's a tip to save you a few minutes, a few quid, or even a few sleepless nights, here's how we mark it.

That's enough waffle

Time to get started! Beginners should tackle this book by starting at the beginning and working through as their understanding develops. More experienced users should use the Contents to find the information they most need to know. While even ARexx experts will need the formidable ARexx reference section at the back of the book.

Toby Simpson knows a lot about programming. He also knows a lot about ARexx. The reference information in this book is more complete and up to date even than the official Commodore documentation...

Section A
Introduction

Your Amiga is a multi-tasking computer. Although it only has one microprocessor chip, its advanced programming allows it to give the appearance of running many programs at once, by carefully dividing up the chip's available time. The program responsible for handling this on a computer is called the Operating System – in the Amiga, this is your Kickstart ROM.

As an increasing number of people buy extra memory and hard disks, more and more are making use of the Amiga's multi-tasking by running more than one program at once, or by moving information from one application to another. An example might be a user with an art package and a desktop publishing (DTP) package, moving a drawing from the art package into the DTP one to use as an illustration within a document.

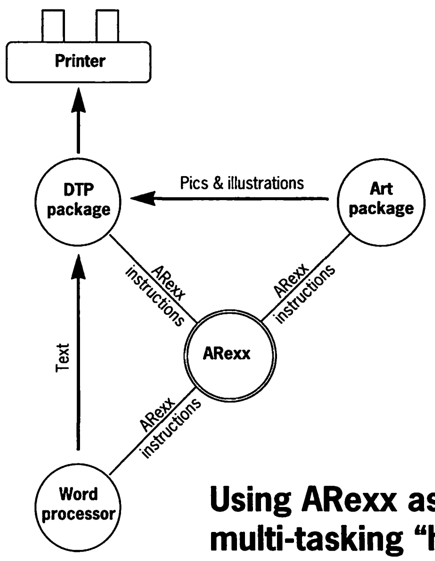
This movement of information from one program to another can be very time consuming, repetitive and extremely boring and, if there's a lot of it to do, then it's easy to make mistakes. The thing is, computers are particularly good at doing repetitive, boring tasks like this. Wouldn't it be great if there was a way of getting the computer to move all this information from application to application for you? Well, there is. Its called "ARexx", and it has been supplied with every Amiga sold since the A500+. If you have Kickstart 2 or above, you have ARexx.

MAKE A NOTE!

ARexx is a programming language rather like BASIC or C. But what makes ARexx particularly clever, and different from other computer languages, is that it can talk to other programs and make them do things.

It might be possible, for example, to write a program in ARexx that could check a document in a word processor to make sure that there was a space after every full stop and

ARexx can control several programs at once (provided each has a built-in ARexx "port"). In this hypothetical example, ARexx is combining the output from an art package and a word processor in a DTP program file, which it then sends to a printer.



Using ARexx as a multi-tasking "hub"

comma – a sort of grammar-checking facility. The ARexx program would communicate with the word processor through its "ARexx Port", issuing it with instructions on what to do.

ARexx can also act as a central hub, and control several applications all at once. Suppose you had a painting package, like Art Department Pro, and 400 pictures that you needed to put a border around and change from Hi-Res to Lo-Res – you would be faced with a very big job indeed. You would need to load each picture into your paint package and add the border, then you'd have to run each frame through Art Department to convert it into Lo-Res. But this job becomes straightforward, quick and easy using ARexx. You can write a small program that runs each picture through the paint package, and through Art Department. You could then sit back and watch your computer do the work for you. However, before you get too excited, ARexx can't talk to every program, only those that come with an "ARexx Port".

WHAT DOES IT MEAN?

An ARexx port is the name for a special point in an application which allows ARexx to communicate with it.

In addition, the amount of control you have over applications with ARexx ports depends on the commands that they allow you to use. When you run a program which supports ARexx, you get a whole load of new commands which directly control that program. So a word processor, for example, might add commands such as "Bold On", "Print Document", and "Load Document".

Armed with these extra commands, and ARexx's flexibility, you can start to make the most of your multi-tasking computer, and those expensive applications – making them do things that you never thought possible. This book will teach you the basics of ARexx, and act as a useful reference guide to the language. It contains a whole section on writing ARexx programs to control popular Amiga applications, from communications packages to word processing, illustrating how to add additional features to these programs yourself.

But as well as being an application control language, ARexx is a very powerful general-purpose language, and quite easy to learn.

The section on Programming ARexx shows how to use ARexx as a stand-alone programming language, and demonstrates the sort of things you can achieve with it.

Running ARexx

ARexx is an interpreted computer language, which means that a special program (the ARexx interpreter) is needed in order to run ARexx programs. This interpreter is found in the System drawer, on your Workbench disk, and it's called "RexxMast". Before you can run ARexx programs you first need to run this, which is done by opening your System drawer and double-clicking on the RexxMast icon. A small window will pop up to tell you that ARexx is now running, and you're in business.

TOP TIP

Running ARexx will take up about 50K of memory. If you are likely to make extensive use of ARexx, and you don't mind losing 50K, then you can configure ARexx to run automatically every time you boot your Amiga by copying the RexxMast program into your WBStartup drawer. This is easily done using the Workbench by dragging the icon from the System drawer into the WBStartup drawer. Now re-boot your Amiga, and ARexx will start up.

Running the interpreter will not affect the operation of your Amiga in any way, other than the loss of 50k of RAM, but without it you cannot run ARexx programs.

If you are using 1.3, then follow the installation instructions that come with ARexx.

The History of ARexx

ARexx stands for "Amiga Rexx." REXX is a language which began its life in 1979 at IBM, in the hands of Mike Cowlishaw. He wanted to create a language that was easy to use and learn, yet flexible and powerful at the same time.

So REXX was designed to be best at processing the information that people use from day to day, such as words and numbers. It is highly readable and, because it is a "typeless" language (you don't have to specify whether you're storing numbers or text – the language will figure it out for you) and it's very easy to program in.

The single most important feature in REXX, which has made it particularly special for Amiga users, is its ability to "talk" to other applications. The result of this is that users only needed to learn one language. This means not having to learn a totally different script language for each application used and, more importantly, it was easily possible to write programs that would allow these applications to share information, thus giving the user the power to make the most of a multi-tasking environment.

REXX was developed through the early '80s, and was used extensively on IBM mainframes. In 1987 it became the standard procedures language for all IBM Systems Application Architecture operating systems (a large number of big machines!). 1987 was also special for another reason. This was the year that William Hawes ported REXX to the Amiga – and ARexx was born.

The Amiga was an ideal platform for a language such as REXX because of its true multi-tasking, something other microcomputers in its price range could not offer. William Hawes developed, supported and sold the product himself, and as its popularity grew, more and more major Amiga applications came with an ARexx Port. When Commodore released Workbench 2, they decided to ship ARexx as standard, and have done so with every Amiga since.

ARexx is the ideal tool for anyone who uses power applications such as art packages, DTP programs, rendering software and so on – it lets users automate repetitive tasks, and move information from application to application.

MAKE A NOTE!

REXX is still developing, and anyone interested in further details, might like to read the book, "The REXX Language, A Practical Approach to Programming", published by Prentice Hall, 2nd Edition, and written by the designer of the Language, M. F. Cowlishaw.

ARexx on the Amiga

When ARexx first became available for the Amiga there were no applications to take advantage of it. However, support of the language grew rapidly as software companies realised that ARexx was an excellent way of allowing users to control their programs. Its rapid rise in popularity was certainly helped by the fact that the Amiga implementation of ARexx was so well written, and this contributed to its acceptance as a standard.

The language is extremely reliable. ARexx does not crash, so it's possible to trust it – a major advantage.

As has already been mentioned ARexx requires very little memory in order to run – about 50k is all it needs. Because of this, it's easier for people to take advantage of it, and they don't have to worry about massive memory requirements in order to do anything useful with it. When it was initially developed, back in 1987, 1Mb of RAM was an awful lot, and not many people had fast Amigas – so it was written to work with these low-end machines too.

TOP TIP **Because the language is so easy to learn and understand, everyone can make use of it. If your word processor does not have a particular feature that you need, but has good ARexx support, then there's a chance that you can add this feature yourself, or even use another application to do it. Before ARexx, this sort of self-expandability was out of the question.**

As more and more programs come with ARexx support, a knowledge of the language becomes increasingly useful. Your Amiga is a very powerful machine – ARexx gives everyday users access to this power.

ARexx for 1.3 machines

What if you have an older Amiga without ARexx and you would like to use it? There is a way. ARexx has been around for a very long time, and is available as a separate package. (Remember that if you have Workbench 2 or above, you already have ARexx and you do NOT need to buy it.)

MAKE A NOTE! **Contact the author, William Hawes, at PO Box 308, Maynard, MA 01754, USA**
Tel: (617) 568-8695.

It should also be available in the UK from large Amiga dealers.

Section B

Programming in ARexx

From the moment you switch your computer on, to the moment you switch it off, all it ever does is perform huge amounts of very simple instructions, such as "Add" and "Subtract."

A long list of these instructions is called a program, and the computer simply acts on each one sequentially. Some instructions allow you to go around in loops, others will allow you to take decisions. These simple instructions are called "Machine Code".

It's very hard to learn to program in Machine Code – because it's so very simple! In order to perform any major action, such as printing your name on the screen, you may need to use several hundred machine code instructions, which makes the programs take longer to write, and it's more likely that you will make errors while typing it in. Of course, the advantage of using Machine Code is that you do have total control over every single tiny operation that takes place while your specified task is carried out.

Learning machine code is not an easy undertaking, and for most of us, who just wish to write a few simple programs, it is totally unnecessary.

It's far better to get to know a language that is quicker to learn, easy to write, powerful and flexible. But learning to program any language on the Amiga can be an expensive undertaking in terms of documentation and software. When learning C, for example, you can easily spend £500, and still not have half of what you'd really like to own.

ARexx is an ideal starting point for a beginner because it's supplied free with the Amiga, and it's also very powerful.

Getting Started

First, you are going to need to run the ARexx interpreter, as described in the introduction. Once this is running, you are ready to write your first ARexx program. However, in order to program in ARexx there are a few things you will have to be familiar with:

- Using the shell. The shell is found in the "System" drawer on your Workbench disk. You can open a shell by double clicking on its icon. With a shell open, you are able to issue commands to your computer from the keyboard, and view the results.

- Using a Text Editor. A text editor allows you to type in your programs, and then save them to disk so that you can test and run them. An editor called "Ed" is supplied with every Amiga. You can only run this editor from the shell. You might also like to look at Micro Emacs, a more powerful but much harder to use editor which you will find in your "Tools" drawer.

Instructions to use the editor and a simple introduction to the shell can be found in the manuals which came with your computer (or see the back of this book for details of Amiga Format magazine's *Workbench & AmigaDOS Reference*). Owners of A1200s might find the instructions on how to use the shell a little sparse, though, and might like to consider buying a book on how to use it that goes into greater detail.

Now let's get going with our first simple program. The first program to write in any unfamiliar programming language is one that prints "Hello World" on the screen! Open a shell, and type:

```
ed ram:hello.rexx
```

Ed will load, and you'll then have a nice empty window ready to type your program in. Type in this, pressing return at the end of each line:

```
/* Our first program */
SAY "Hello World!"
EXIT
```

When you've finished, select save and then quit the editor. Your first ARexx program is now ready to run! Assuming you have correctly run the RexxMast program, you can now type this:

```
rx ram:hello.rexx
```

And, if you have not made any mistakes, then the result should be:

```
Hello World!
```

Well done, you've written your first ARexx program. Now let's go through it in more detail. The first line is a comment. Comments in ARexx start with a /* and end with a */. This, incidentally, is the same way in which comments are expressed in the C programming language.

WARNING

In order for your program to work, you must ensure that the first line of EVERY program is a comment, otherwise ARexx will not recognise the program! The best thing to do is to briefly describe the purpose of the program in the comment.

SAY is an ARexx keyword. Although we will be using capital letters for keywords to make them clearer in this section, they are not case sensitive as far as ARexx is concerned, so you could have written:

```
say "Hello World"
```

or even:

```
sAy "Hello World"
```

It's best to decide which case you are going to use for keywords, and stick to it. SAY means "Show some information to the current output console with a new-line at the end". The current output console in this case, is your shell window, so that's where the "Hello World" goes.

The last line of our program contains the ARexx keyword "EXIT". This means "Stop running this program now.". You'll notice that we won't use this in most of our early example programs – with simple scripts like this it is not necessary, because the last line in our script is always the last thing to be done in our program. In some of our more complicated examples later, this may not be the case.

ARexx has a compact and comprehensive range of keywords and additional power is added by other applications using ARexx Ports, and by special libraries. For your interest, here is the list of actual ARexx Keywords. It probably won't mean too much at this state, but you'll see us introduce them as we proceed through this section, and learn how to use them.

If you're particularly curious about any of them, and perhaps already know a different programming language, you may be interested in looking them up in the reference section.

AREXX KEYWORDS

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	TRACE	WHEN

So, what happened when we ran our first program? Well, ARexx itself consists of the RexxMast interpreter, and a whole load of other assorted little programs required to actually run ARexx programs, normally referred to as "Arexx scripts". For us the most important of these, is the "Rx" command, which means Run ARexx script. This actually sits in a drawer on your Workbench disk called Rexxc, along with some other small programs which we'll come to later in the book.

When you type Rx, it expects you to tell it where the file containing your ARexx program is. We did this in our first example by simply specifying the direct path, which was ram:hello.rexx. The Rx program will add the .rexx to the end of the file if you don't specify it, so if we'd typed "rx ram:hello" then our script would still have run. Furthermore, it's possible to create a drawer and put all your ARexx programs in it, and tell the ARexx interpreter where that drawer is, then you don't even need to specify a path, you can simply type "rx hello". *All* ARexx programs should have a .rexx on the end of the filename so that you can recognise them.

The default place to put ARexx programs is in the S: drawer on your Workbench disk. Your Amiga will set up an "Assign" called "Rexx:" which points to the S: drawer. I would recommend that you don't store all your programs here, because the S: drawer will become very cluttered. The best thing to do is to change this assign to point to a new drawer which you can create especially. Its easy to do this from the shell. Firstly, make a drawer somewhere, using the makedir command. For example:

```
makedir sys:ARexx_Programs
```

If you have a hard disk in your Amiga, you may want to put this drawer on another partition, such as your "work:" partition for example. Then, you need to edit your s:user-startup sequence:

MAKE A NOTE!

If you are using Workbench 1.3, then you won't have a s:user-startup file, instead, edit your s:startup-sequence file, and add the below assign line to the END of that file, just before the LOADWB line. Workbench 2 users should NEVER alter their S:Startup-sequence file, only their s:user-startup.

```
Ed s:user-startup
```

And add one line to the bottom of the file:

```
assign rexx: sys:ARexx_Programs
```

If you are using 1.3, then you will have to add this line to the bottom of your s:startup-sequence, not your s:user-startup. "sys:" just refers to the disk which you booted off, which is likely to be a copy of your workbench floppy, or a hard disk partition containing the Workbench software.

When you have made this change, reset your computer, and you should now have rexx: set up. Now, you should create your ARexx scripts in this new drawer. You could create a program called "world.rexx" like this:

```
Ed rexx:world.rexx
```

Now, you could run it with:

```
Rx world
```

And the ARexx interpreter will find your program by itself, by looking in the rexx: drawer for a file called world.rexx.

Introducing variables

The power of a programming language comes from its ability to store information and perform operations on it. Variables are used to store this information in. It's very easy to assign values to variables in ARexx. For example:

```
age = 23
```

This assigns the value 23 to the variable age. We can now easily perform operations on this value. If we wanted to add 5 to age, we could do this:

```
age = age + 5
```

This simply means "Set the variable age to whatever age is currently equal to, plus 5".

We can then show this value on the screen by using the SAY statement, which we used earlier to print "Hello World" on the screen:

```
SAY age
```

The result on the screen will be:

```
28
```

Note that we didn't put quotes around the variable name. If we had, the result would have been:

```
age
```

The difference is simple. If we put quotes around something, it is treated as a string. A string is simply a series of alphanumeric characters and punctuation, surrounded with quotes.

As well as adding, we can perform a number of mathematical operations on variables. The Arithmetic operators are shown below:

ARITHMETIC OPERATORS

Operator	Name	Priority	Example	Result
+	Addition	5	3+4.5	7.5
-	Subtraction	5	4-1	3
*	Multiplication	6	3*1.1	3.3
/	Division	6	10/2	5
%	Integer Division	6	10/3	3
//	Remainder	6	10/3	1
**	Exponention	7	1.4**2	1.96
-	Prefix Negation*	8	-10.3	-10.3
+	Prefix Conversion*	8	+" 12.123"	12.123

COMPARISON OPERATORS

Operator	Name	Priority	Example	Result
==	Exact Equality*	3		
~==	Exact Inequality*	3		
=	Equality	3		
~=	Inequality	3		
>	Greater than	3		
<	Smaller than	3		
>= or ~<	Greater than or equal to	3		
<= or ~>	Smaller than or equal to	3		

LOGICAL OPERATORS (Boolean)

Operator	Name	Priority	Example	Result
~	NOT	8		
&	AND	2		
|	OR	1		
^ or &&	Exclusive or	1		

* Prefix negation and conversion may appear to be a little confusing. We use Prefix Conversion in ARexx to convert strings to numbers, as in the example given. Prefix negation may appear a little more complex, take this example:

SAY -7

It may not occur that the '-' is an operator at all, and just part of the number. But it is in fact an operator, the prefix negation. Its priority is sufficiently high that you can think of it as a part of the number itself.

* Exact equality/inequality differ from the normal equality/inequality operators in that the comparison has to be totally exact to evaluate to TRUE (i.e. succeed). Take this example:

```
/* Equality test */
string = " lion"

IF string = "lion" THEN SAY "It was a lion, 1"
IF string == "lion" THEN SAY "It was a lion, 2"
```

The output will only show the first SAY, because our string contains one space at the front. Normal equality will quietly ignore this, and assume that " lion " is the same as "lion" and so on. Exact equality requires that both parts of the comparison are exactly identical in every respect for it to succeed.

There are also two "concatenation" operators, || and a blank space. We have used these to join strings. || differs from a blank space in that it joins strings with no intervening blank. The priority of concatenation operators is 4.

Variables aren't limited to storing numeric values. They can also be used to store strings. We could, for example, do this:

```
name = "Toby"
```

And, then using the SAY statement, we can print out the contents of this variable, as before:

```
SAY name
```

ARexx is a "typeless" language. What this means is that it automatically detects and checks to see whether variables hold numbers or strings before using them, and gives an error message if it is incorrect. In other programming languages, you have to define whether a variable is going to hold a number, or a string. If you try to perform an illegal operation with a variable – say, for example add 5 to the string "Fred" – then ARexx will show an error message and will not run the program until you fix it.

Variable names

You can name your variables pretty much what you like. But one of the easiest ways to create totally unreadable programs is to skimp on variable names, opting for simple ones like 'X' and 'N'. This can get you into trouble very quickly, especially with larger programs, because although you can remember what each variable does in the short term, after a few days it gets a little harder. Finding errors in your program with a whole bunch of single letter, or meaningless variable names is very difficult.

VARIABLE TERMS

Character: A single digit, letter or punctuation mark. Examples of a character: Z, or $, or µ. Each character has a unique numeric code which we can use to refer to it. These are called ASCII codes. The ASCII code for the capital letter A is 65, for example, and the ASCII code for the number 0 is 48.

String: A collection of characters joined together. An example of a string is 'Hello', or 'Counting down 5-4-3-2-1-ZERO!!!'. In a lot of programming languages we tend to mark the end of a string with a character with the ASCII code of 0, and refer to such strings as "Null terminated strings". By searching for the Null character you are able to find the end, and hence the length, of the string itself.

Integer: A whole number with no fractional parts. An example of an integer is 0, or 1,234, or -68.

Floating point number: A number with possible fractional parts. Some examples of floating point values are 1.5, 10.0, 3.141592654. The accuracy of the floating point arithmetic in ARexx, and the way in which it outputs it, can be defined by using the NUMERIC keyword. See Section E for further information.

The best thing is to name variables according to what they are and what they will hold. Variables can be made up of any combination of letters, numbers and the characters "$?!_" (dollar, question mark, exclamation mark and underscore). Variables cannot begin with a number and, unlike some other computer languages, they are not case sensitive. Be careful with this, because it means that "x_position" and "X_POSITION" are the same variable. In this example, all three variables are identical:

```
/* Variables */

i_am_a_variable = 2

I_AM_A_VARIABLE = 3

I_aM_a_VaRiAbLe = 4

SAY i_am_a_variable I_aM_a_VaRiAbLe I_AM_A_VARIABLE
```

We often use the underscore character "_" where we would normally use a space. Since you can't use spaces in variable names, the underscore allows for more readable names.

```
/* Some very readable variable names! */
cost_to_learn_arexx = price_of_amiga + price_of_monitor
+ cost_of_this_book

SAY "You spent" cost_to_learn_arexx "learning this
language!"
```

Obviously, you can get carried away. There's no point in using variable names 10 feet long because they'll take months to type in but, in general, think about your variable names to ensure that they make sense to you – and programming in ARexx will be much easier!

Simple Programming

The sort of programs we have tried so far are totally sequential, in that they start at the top and every instruction in the program is acted upon, one after the other, until we run out.

But before you can do anything useful with a computer language, you have to be able to take decisions and perform loops. Lets have a little look at this example, which prints the 12 times table on the screen:

```
/* Show the 12 times table on the screen */
SAY "Here is the 12 times table:"

DO loop = 1 TO 10
  SAY loop * 12
  END

SAY "All Done!"
```

The important part of this example is the three lines starting from the DO statement. In this example, all the instructions between the DO and the accompanying END will be executed 10 times, and each time around, loop will be incremented by 1.

MAKE A NOTE!

Note that in ARexx we denote multiplication with an asterisk ("*") rather than the "x" that you might expect, to avoid confusion between the lower case X letter.

Also note that we have indented the statements which get executed as part of the loop. This is just programming style, and it makes the script easier to read because you can see where the loops fall at a glance. See the section below on "Programming Environment and Style" for further information. Indenting your scripts like this is quite optional, but definitely recommended!

We could change it to be this:

```
DO loop = 100 TO 200
```

...and we'd get the 12 times table from 100 x 12 to 200 x 12. In fact, we could change it to do any part of the 12 times table by changing these two numbers. At the moment, the results on screen are not very tidy, just a list of the results. It would look much nicer it we tidied it up. Change the SAY line to look like this:

```
SAY loop " x 12 = " loop*12
```

This looks a little more complex, but it's quite straight forward. Firstly, the SAY command would print the current value of the variable loop on the screen, then the string " x 12 = " and finally, the value of the loop variable times 12. The results of this new program would be this:

```
Here is the 12 times table:
1  x 12 =  12
2  x 12 =  24
3  x 12 =  36
4  x 12 =  48
5  x 12 =  60
6  x 12 =  72
7  x 12 =  84
8  x 12 =  96
9  x 12 =  108
10  x 12 =  120
All Done!
```

This is far better. It's easy to change this to print the 16 times table, by changing every instance of 12 to 16. But it would be even better if we asked the user to enter the times table they wanted and printed that. This is done with the PULL command. PULL is used to ask the user to input a value, which is then assigned to a specified variable. For example:

```
/* Times Table example */
SAY "Which times table would you like?"
PULL timestable

SAY "Ok, here is the "timestable" times table."

DO loop = 1 TO 10
  SAY loop " x "timestable" = "loop * timestable
  END
SAY "All Done!"
```

This is far more flexible, because the user gets to decide which times table they want to see. Try entering a string instead of a number when you run the program, to see what happens when ARexx tries to perform multiplication on it:

```
6.System3.1:> rx ram:hello.rexx
Which times table would you like?
fred please
Ok, here is the FRED PLEASE times table.
+++ Error 47 in line 9: Arithmetic conversion error
Command returned 10/47: Arithmetic conversion error
6.System3.1:>
```

ARexx is pretty helpful with its error messages. It tells you what line the error happened on, and what the error actually was. In this case it's pretty obvious what is happening, ARexx is trying to multiply FRED PLEASE by 1 and failing dismally, exactly as you'd expect.

MAKE A NOTE!

Where we show the example output of a script, to show the line we used to actually run our script the shell prompt is also shown with the text. This will be x.System3.1:>, where x is an unimportant number. Whatever is shown after the shell prompt is what you type to run the script itself, in the above example, you type:

```
rx ram:hello.rexx
```

...from the shell to run our program.

Now we can perform loops. An important part of writing programs is being able to make a decision. If something, do something. In ARexx there are a number of mechanisms to allow us to make decisions, and the simplest is the IF statement. Let's illustrate its use using this simple example, which will ask you to input your age and then come up with a suitably inspiring comment depending on what you entered:

```
/* Our Age Cruelty Program! */

SAY "Please Input your Age:"
PULL age

IF age > 29 THEN SAY "You are really past it!"
  ELSE SAY "You're still very young."

IF age > 25 & age < 30 THEN SAY ".. but 30 is looming
closer."
```

Here is an example of us running this program, if we were to enter the age as 27:

```
6.System3.1:> rx ram:hello.rexx
Please Input your Age:
```

```
27
You're still very young.
.. but 30 is looming closer.
6.System3.1:>
```

OK, so 30 isn't really past it and I certainly won't think that way in a few years time! But this does demonstrate the decision-making process. We were able to adapt the results of the program to suit what the user entered.

In the above example we used the IF statement a couple of times; let's go into these in more detail. The structure for IF looks a little like this:

IF expression THEN conditional statement

The ARexx interpreter evaluates the expression to see if it is true. If it is, then the "conditional statement" is executed. For example:

```
IF 5=5 THEN SAY "Hello"
```

This will always show the string "Hello" on the screen, because 5 definitely equals 5, so the statement evaluates to true. If we were to change it to read:

```
IF 5=4 THEN SAY "Hello"
```

Then we would never get the string "Hello" on the screen. The expression can be quite complex, and can consist of a number of individual operators. In our age example, we use the "&" operator, which means AND, in order to check to see if your age was greater than 25 and smaller than 30 in order to work out if you were nearly 30 years old. In addition to the AND operator, there is an OR operator " | ". Using OR allows us to do things like this:

```
IF age < 20 | age > 80 THEN SAY "You're either very
young or very old"
```

You can also use the "~" operator, which means "not", or invert the result of the expression, to do things like this:

```
IF ~(5=5) THEN SAY "Hello"
```

The expression 5=5 evaluates to TRUE and we then invert this result, making it FALSE, so Hello is not shown. Dealing with TRUEs and FALSEs this way is referred to as simple Boolean algebra – because the expression evaluates to either TRUE or FALSE, there is no in between. In programming we tend to associate FALSE with the value 0 (Zero), and TRUE with ~0 (Not Zero). ARexx treats the value TRUE as 1. If we tried this small example program:

```
/* Boolean test */
IF 0 THEN SAY "Zero"
IF 1 THEN SAY "One"
```

We would get the results:

```
7.System3.1:> rx bool
One
```

This is because 1 is TRUE, so the IF statement can work, but the first one will never happen, because the expression is simply '0', which is FALSE. Obviously using IF like this is pretty pointless, because we know what the result will be. As one final footnote on Boolean results, try this out:

```
IF 12345 THEN SAY "Hello"
```

When run, it produces the result:

```
7.System3.1:> rx bool
```

```
+++ Error 46 in line 3: Boolean value not 0 or 1
Command returned 10/46: Boolean value not 0 or 1
```

You should be able to realise why this is. IF tried to evaluate the expression '12345'. The result was '12345', which is neither TRUE nor FALSE. So it rejected it with an error statement along the lines of "That should have been Boolean. But it wasn't. What are you up to?".

As well as IF's single conditional statement, we can also have a statement that is run if the expression is not true. For this, we use the ELSE command. This is also demonstrated in our age example, because we are able to show one string if your age is greater than 29, or another if it is not.

Of course the problem with what we know so far is that we can only have one conditional statement. What if we wanted to do a number of things?

This is where the DO and END statements come in. We've already seen DO and END used to perform a loop in our times table example. We can also use it to group together a number of statements into one block. For example, we could alter our age program to show you how old you will be in 10 years if you're older than 30, replacing the existing IF age > 29 lines with these:

```
IF age > 29 THEN DO
    SAY "You are really past it!"
    SAY "And in 10 years time you will be "age+10"
years old!!"
    END
  ELSE SAY "You're still very young."
```

We could have any number of statements between the DO and END.

A Note About Quotes

Single and double quotes are interchangeable in ARexx as long as pairs match up. For example:

```
name = "Toby"
```

... and ...

```
name = 'Toby'
```

... are identical as far as ARexx is concerned. You can't mix them in pairs, though, so:

```
name = "Toby'
```

... is not valid, and will generate an error "Unmatched quote". This feature, however, can be quite useful at occasions. Imagine if you wanted to show the string:

```
"Watch it!", said the Armadillo, "Don't do that."
```

Just typing it in with a SAY command will generate all sorts of errors, but we can show it by using other quotes:

```
/* Show the Armadillo string: */
SAY '"Watch it!", said the Armadillo, "' || "Don't do
that." '"'
```

Here is another example:

```
/* Quotes */

single_quote = "'"
double_quote = '"'

SAY single_quote
SAY double_quote
```

When run, this program will show a single quote (') and a double quote (") on the screen on separate lines.

Although quotes are interchangeable in this way, it makes good sense to stick to using one type consistently. In this book, we use double quotes for everything, except for sending commands to another application (see Section C), when we put our commands in single quotes to make it easier to distinquish between the two. You can choose whatever convention you like, but sticking to the one used in this book will help to make your more complicated ARexx programs easier to read.

More on looping and decisions

We've looked at some simple ways of performing loops, and making basic decisions, now let's move on to some more complex options. There are instances where the methods discussed above are simply not efficient enough, and result in unnecessarily long and complicated programs. ARexx provides a wide range of powerful choices, all operating slightly differently and all designed to suit different circumstances.

The SELECT Statement

Suppose you have a program which pops up a requester on the screen, gives the user a choice of five buttons and, depending on the button pressed, causes a particular action to take place. We can simulate this kind of requester now with a PULL statement, so we can input a number and make one of the actions occur. If we use the ARexx commands we already know, we might end up with a program that looks a little like this:

```
/* Button test program */
SAY "Input the button number:"
```

```
PULL button_pressed

IF button_pressed = 0 THEN DO
  SAY "This is the action on button 0 (Cancel)"
  END

IF button_pressed = 1 THEN DO
  SAY "This is the action on button 1"
  END

IF button_pressed = 2 THEN DO
  SAY "This is the action on button 2"
  END
```

...and so on. The more options we have, the more IF statements we will require. However, programs designed like this can soon get very long and complex and it's easy to make errors. At the moment, if we typed in a button which does not exist, then the program could not cope with it. There are some ways around this. If, for example, the maximum button was 5, we could encase all of the IF statements in another IF statement, like this:

```
IF button_pressed >=0 & button_pressed < 6 THEN
  DO
  put all our IFs here
  END
ELSE
  SAY "Illegal button pressed"
```

Generally, we can work around it, but it's certainly not ideal. However, there is an ARexx command which does all of this for you and which results in neater programs. It's called "SELECT" and it operates like this:

```
SELECT
  WHEN button_pressed = 0 THEN
    DO
  SAY "Button 0 here"
  END
WHEN button_pressed = 1 THEN
  DO
  SAY "Button 1 here"
  END
OTHERWISE
  SAY "Illegal button pressed"
END
```

This is a much neater way of solving the problem. It's a far neater program and it's easier to read. SELECT is ideal for situations where you need to make a number of decisions based on some action taking place. Menus, for example, are very easy to implement with the SELECT statement – you simply allow the user to input a choice from the menu, and then act on it accordingly. You can also detect incorrect entries easily, and act on them using OTHERWISE. Before we leave SELECT, let's just summarise its operation:

```
SELECT
  WHEN expression THEN statement
  WHEN expression THEN statement
  .... as many WHENs as you like....
  OTHERWISE statement
  END
```

You can have as many WHEN lines as you like, but only one otherwise. When you use SELECT in your program, ARexx then expects one or more WHEN lines. As we showed above with IF, you can use DO and END to allow more than one statement to happen. The OTHERWISE line is optional. If it is included, then if ARexx does not execute any of the WHEN lines it will run the OTHERWISE. When you have finished with select, you have an END line to tell ARexx this. A common mistake in your program might be to miss off END statements when they were needed. ARexx will report an error in this case. If it says something along the lines of "Invalid statement in SELECT" then it is quite likely that this is your problem:

```
7.System3.1:> rx button
Input the button number:
1
Button 1 here
+++ Error 23 in line 25: Invalid statement in SELECT
Command returned 10/23: Invalid statement in SELECT
```

The above output came as a result of running the SELECT example program above, without the final END statement.

The DO statement

We've already seen this used in a number of examples, mostly to allow us to make a number of statements happen after an IF, and for some simple looping. DO is actually much more flexible than this, and can be used to perform some powerful looping. One of the first programs most schoolchildren write is one which prints their name all over the screen in an endless loop. In ARexx, we can use DO FOREVER, like this:

```
/* The DO statement */

DO FOREVER
   SAY "Enter your name"
   PULL users_name
   SAY "Hello "users_name", how are you?"
   END
```

In this example, we're getting a name and then saying "Hello name, how are you?" forever. It's a pretty pointless program, but there are occasions where looping like this is quite useful, particularly if you have a way out. In the program above, the only way to quit is to hold down CTRL and press C, (CTRL-C) then releasing CTRL and C, and pressing return. ARexx will then halt execution of the program like this:

```
Enter your name
john
Hello JOHN, how are you?
Enter your name

+++ Error 2 in line 6: Execution halted
Command returned 10/2: Execution halted
7.System3.1:>
```

You may notice, incidentally, that the PULL instruction appears to have converted the lower case "john" that we actually entered into upper case. Surely it should not have done this? Well, actually it is working correctly. PULL is the shorthand for a much more complex command, called PARSE. PULL really means "PARSE UPPER PULL". We shall be dealing with PARSE later on. In the meantime, if you wish string variables to retain their correct case, then you can use:

```
PARSE PULL
```

...instead of PULL. PARSE is a complex but powerful instruction, and we have purposely left it till later. Full information on it is in the Reference section.

Fortunately, ARexx has provided us with a proper way of exiting from a DO FOREVER loop, while continuing to run the program. This is the LEAVE statement. We could, for example, add this line right after the PULL instruction in the program above:

```
IF users_name = "QUIT" THEN LEAVE
```

If, instead of a name, we now typed QUIT, the program will end. This could be quite handy, for example, if you had written a telephone number storing program, and you wanted the user to be able to enter a number of phone numbers into the book and exit easily when they'd finished. You could use a word like QUIT to exit the DO loop.

We have also used DO in simple loops, which looked a little like this:

```
DO loop = 1 TO 10
  SAY loop " x "timestable" = "loop * timestable
  END
```

DO is capable of far more than this because it allows much more control over the looping range than a simple start and stop value.

Normally, our loop counter increases, or increments, by one each time starting from the first value, until it reaches the last. We can specify the amount it goes up by using BY:

```
/* The DO statement */
DO loop = 10 TO 1 BY -2
  SAY "Loop value is "loop
  END
```

This program produces a list of the numbers 10 8 6 4 2 and then exits.

We can also loop while, or until, an expression evaluates to TRUE. Take this simple example, which continues until the user types in an age greater than 30:

```
/* The DO statement */
DO UNTIL age > 29
  SAY "Enter your age:"
  PULL age
  END
SAY "You are too old!"
```

In this case, the loop will continue *until* age is greater than 29, i.e. 30 or above, and then exit. We could have also written this using WHILE:

```
age = 0
DO WHILE age < 30
  SAY "Enter your age:"
  PULL age
  END
SAY "You are too old!"
```

This loop will continue *while* age is less than 30. Notice that in the WHILE example, we set "age" to a known value before the DO loop starts. This is to prevent the loop going wrong the first time around, before the user has had a chance to enter an age. This is not necessary for the UNTIL example, as we shall see below.

A common programming fault is "uninitialised variables". Some computer languages will spot this and prevent you from running a program if some variables have not properly been set up. ARexx, however, does not – when it sees a variable for the first time, it creates it on the spot. This sort of thing is not a problem in the "DO loop = 1 TO 100 BY -2" type loops, because the starting value is specified.

So when do you use WHILE and UNTIL? And how do you know which to use? Well, there is a distinct difference between WHILE and UNTIL. Let us briefly look at the structure of a DO WHILE loop:

```
DO WHILE expression
  statements...
  END
```

The statements between the DO WHILE and the END will be executed WHILE the expression evaluates to true. WHILE expressions are evaluated and tested *before* each loop starts. DO UNTIL loops, however, are different, because the UNTIL expression is only evaluated and tested at the *end* of each loop. This means that in the above age testing example program we did not really need to set the age variable to 0 before we started (as we had to do in the WHILE example), because it will have been set to something sensible – an age value – before the UNTIL expression gets evaluated.

If you are unsure about how any of these commands work, try saying them to yourself:

```
DO WHILE age < 30
  END
```

..."Loop while the age value is smaller than 30". You'll find that it can help a lot to read a program back to yourself in real English. ARexx is excellent for checking in this way.

More about variables

As we discussed above, variable names are made up of a load of alphanumeric characters, are not case sensitive, can contain the punctuation symbols $?!_ and may not begin with a digit. Sensible variable names are vital for large programs, because they make them considerably easier to understand – and to debug – should something go wrong. However, try not to go overboard with the length of the name, instead make it more readable by sensible use of upper and lower case characters:

```
COSTOFBOOK = 12.50
costofbook = 12.50
```

```
CostOfBook = 12.50
Cost_Of_Book = 12.50
```

The first three examples above are equivalent, because case is ignored. The last however is not, because we've used some underscores to space the words out and make the variable more readable.

It's a matter of personal choice how you use variable names. Personally, I prefer to keep them all lower case, separate words with underscores, and keep all functions and statements in upper case, for better readability. If you use a variable and do not put a value into it, it will default to the variable name itself, in upper case. For example:

```
/* A test to see what happens with an uninitialised
variable */
SAY new_variable
```

Produces the result:

```
7.System3.1:> rx new
NEW_VARIABLE
```

The other key thing to remember about variables in ARexx is that they are typeless. This means that you do not specifically have to state what each variable is going to hold. In other languages, such as C, you have to declare all variables before you use them and state what type they are – such as an integer, or floating point, or string variable.

One of the things that makes ARexx so attractive to beginners is that it does not require any such declarations, ARexx sorts out the types for you when you use them.

The DROP statement

Perhaps not the most useful of statements, but there are still occasions when it is handy to be able to "uninitialise" a variable, i.e. put it back to its initial empty state – which means that its contents will be the variable name itself in upper case. ARexx has a special statement for this – DROP:

```
/* DROP example */

SAY Test_Variable
Test_Variable = 5       /* Set variable to 5 */
SAY Test_Variable
DROP Test_Variable      /* Forget it again */
SAY Test_Variable
```

Which produces the result:

```
7.System3.1:> rx drop
TEST_VARIABLE
5
TEST_VARIABLE
```

As you can see, we first showed the contents of the variable without initialising it, which was simply its own name in upper case. Then, we assigned a value to it, and used SAY to show that too. Finally, we DROP'd the variable and then displayed the variable contents one last time.

Compound symbols

The variables we've used so far have been simple and straightforward – as each variable stores one value. Imagine you have just written a program in ARexx to store friends' phone numbers. How would you go about doing this?

Well, with the knowledge we have acquired so far, we might end up with a pretty large, long-winded program that could look like this:

```
/* 3 name phone directory */

DO FOREVER
  SAY "Which record to change?"
  PULL record_id

  SELECT
    WHEN record_id = 1 THEN
      DO
      SAY "Name?"
      PULL name_1
      SAY "Number?"
      PULL number_1
      END
    WHEN record_id = 2 THEN
      /* Etc etc, repeat above for EVERY number */

    END
  END
```

Goodness, what a nightmare. If we had 100 names and phone numbers, the above program would contain 100 of the WHEN chunks in the select. And that's just to enter the names and numbers. Repeat that lot again for a routine to display each name and phone number? It doesn't take an Einstein to come to the conclusion that there really must be a better way of doing this and, amazingly enough, there is.

In most computer languages there is a mechanism for arrays. What is an array? An array is a list of similar items which share the same variable name. Still lost? OK, let's use a small example:

```
/* Array example */

name.1 = "Toby Simpson"
name.2 = "Mr Blobby"
name.3 = "Stegosaurus"
name.4 = "Habbish"

SAY name.1
SAY "Which name to see?"
PULL name_id
SAY name.name_id
```

Firstly, although in most other programming languages, such as C and BASIC we use the word array, in ARexx, they are "compound variables", and as you'll see, they are very powerful indeed – far more so than simple arrays. But before we discuss the above example, it's probably best just to see some results of running it, because that way you'll probably be able to guess what is going on:

```
7.System3.1:> rx array
Toby Simpson
Which name to see?
3
Stegosaurus
7.System3.1:> rx array
Toby Simpson
Which name to see?
8
NAME.8
7.System3.1:> rx array
Toby Simpson
Which name to see?
john
NAME.JOHN
```

The first thing our program does is to set up a compound variable called "name". This is called the "stem symbol". What then follows is a number of nodes, each separated by a full stop.

In our simple example, we only have one node, which is a numeric value dictating which of the names we would like to view. A node is a fixed or simple symbol. Fixed symbols are just numeric values, such as 9876, or 10. Simple symbols are variables, like the ones we have dealt with before. So how does this all work? When ARexx encounters the use of a compound variable, it expands it fully by replacing each node with its current value. In our program, for example, we did this:

```
name.3 = "Stegosaurus"
```

name is the name of our stem symbol, and since 3 is a fixed symbol, the string "Stegosaurus" is assigned to name.3. Later on we could view this by doing something a little like this:

```
name_id = 3
SAY name.name_id
```

ARexx expands this out to:

```
name.3
```

...which just happens to contain the value "Stegosaurus". So, what happens if we refer to a node which does not exist? Well, just like with simple symbols that we have used before, they default to the name, used in upper case. In our above example program output, we tried to get the result of name.8 and name.john. Both values did not contain anything, hence the result.

If an assignment is made to the stem symbol, then that value is automatically assigned to every possible compound symbol made up from that stem, a sort of "default value". This means, that if we were to make such an assignment before we put the names in, like this:

```
name. = "No name entered"
```

...then if we tried to SAY a value which we had not yet defined, such as name.john, we would get this result instead:

```
No name entered
```

Neat, isn't it? Note that we have tried name.john a couple of times, and john is quite obviously not a number, it's a name, or a variable. ARexx compound variables can be regarded as "associative memory", in that you look things up by name rather than the single numeric index which normal array systems in other computer languages use. This makes for some incredibly powerful uses. Take this small example of a phone book:

```
/* Phone book example */

name. = "(no name entered)"
name.steg = "Stegosaurus     0101 010 101 10101"
name.toby = "Toby Simpson    1234 566 789 98762"

SAY "Which name would you like to look up?"
PULL person's_name

SAY name.person's_name
```

This is what happens when we run this little program:

```
7.System3.1:> rx array
Which name would you like to look up?
fred
(no name entered)
7.System3.1:> rx array
Which name would you like to look up?
steg
Stegosaurus     0101 010 101 10101
```

By typing in the actual name of the person we wanted, the compound symbol was then expanded on the SAY line at the end. If we type in a name which we do not have, we simply get (no name entered) on the screen, which is of course the value we assigned to our stem symbol at the start of the program.

As well as using compound symbols to look up by name, as we have demonstrated above, they can also be used as conventional integer indexed arrays, such as those you might find in C or BASIC. Have a look at this example, which shows a list of all the names we have in our phonebook using a DO loop:

```
/* Indexed Array example */

name. = "(no name entered)"
name.1 = "Stegosaurus     0101 010 101 10101"
name.2 = "Toby Simpson    1234 566 789 98762"

loop = 1

DO UNTIL name.loop = "(no name entered)"
  SAY name.loop
  loop = loop + 1
  END

SAY "End of list"
```

When it's run, the results of this program are:

```
7.System3.1:> rx array
Stegosaurus     0101 010 101 10101
Toby Simpson    1234 566 789 98762
End of list
```

We're using DO UNTIL so that we are able to detect when we've reached the end of the list. Since we've assigned "(No name entered)" to our stem symbol, we are able to stop as soon as we encounter it. In the meantime, we're incrementing our index loop counter by one each time.

This is a pretty simple example, and if our names ran from 1-5 and 10-15 then this listing routine would stop at 5 when it found the first empty array entry, but at least it demonstrates how this sort of compound symbol can be very useful to the programmer.

Records and fields

We've seen how to use compound symbols, both associatively, by name, and also in a more conventional indexed manner to store simple information. In the case of our small telephone directory, we were storing the name and phone number. We might also want to store the address, perhaps a postcode and some additional information for each person.

A much better way of handing more complex data like this is by using records.

Normally in such a situation we would have a single file – either a space on your floppy or hard disk, or an area of memory which contained everyone's name, address and other data. Each person's unit of data is called a record. Each single part of this record, such as "Postcode", or "Phone number" is called a field. In ARexx, setting up a file of records and fields is very easy using compound symbols. Let's expand on this phone book to hold addresses and postcodes too. First, we create a new stem symbol, and set everything to a string 'Empty':

```
/* Records example */
address_book. = "Empty"
```

Then, for demonstration purposes, we'll create one record, for 'steg'. We'll have four fields in our record – name, address, phone number and postcode:

```
/* Create one record, for "steg" */
address_book.steg.name = "Stegosaurus"
address_book.steg.phone = "123456789"
address_book.steg.postcode = "STEG 999"
address_book.steg.address = "Jurassic Park, Pangea"
```

Our compound symbol is now made up of three parts – the stem, the name of the record, and finally the name of the field itself. Finally, we get the name of the record we'd like to examine, and then show it on the screen using SAYs:

```
/* Ask user for choice of person to see */
SAY "Whose details would you like to see?"
PULL persons_name

/* Now show the contents of that record */
SAY address_book.person's_name.name
SAY address_book.person's_name.phone
SAY address_book.person's_name.postcode
SAY address_book.person's_name.address
```

The name of the record we're after is stored in the variable 'persons_name', and compound symbol expansion does the rest for us. The result might look like this:

```
7.System3.1:> rx array
Whose details would you like to see?
fred
Empty
Empty
Empty
Empty
7.System3.1:> rx array
```

```
Whose details would you like to see?
steg
Stegosaurus
123456789
STEG 999
Jurassic Park, Pangea
```

We could take our program one step further and ask the user for a field name too, by altering our program from the first PULL statement onwards:

```
/* Ask user for choice of person to see */
SAY "Whose details would you like to see?"
PULL person's_name
SAY "And which field?"
PULL field_name

/* Now show the contents of that record */
SAY address_book.person's_name.field_name
```

Quite clever isn't it? Of course, if instead of using record names we'd used record numbers, for example:

```
address_book.1.name = "Stegosaurus"
```

Then we could use DO loops to scan through the list and perform actions on them. Compound variables are complex, as you can see, and it's quite easy to get into a bit of a pickle with them, particularly over substitution – where ARexx expands your compound symbol by replacing every node by its current node name's value. It gets worse because, in theory, it's pretty much unlimited:

```
/* Compound symbol nightmare */
economic_data.england.exports.car.rover.214.engine.bhp
= 105
SAY
```

```
economic_data.england.exports.car.rover.214.engine.bhp
```

Here we have economic data. The first node is the country we're after, the second is either exports or imports, the third cars (could be wheat), next is car name, type, engine details, and finally the bhp value...

Functions

The programs we have written and experimented with so far have been reasonably straightforward. They have a start, some statements in the middle, and an end. They are all entirely self-contained, they do not need to relate to the outside world for anything.

This may sound pretty obvious, but it's an excellent introduction to functions.

Without functions we have a pretty plain programming language as you have seen. With them, we have the ability to execute a group of statements which might perform an action, or provide us with information. We can write our own functions in ARexx itself and use them many times from within a program, or we could use functions which ARexx has built in, or maybe even use new functions in the form of external function libraries.

Does this still leave you baffled? Let's explain with the aid of a simple example.

Our Program:

```
   /* Test Program */

   /* Call the date function, perform the action and ¬
then return */ the_date = DATE("NORMAL")
```

```
SAY the_date
```

What happens here? The first line of the program says:

```
the_date = DATE("NORMAL")
```

DATE is the name of the function. When ARexx executes this line, it sees that we are assigning something to the variable "the_date", and recognises it as a possible function. DATE() just so happens to be one of the built-in functions internal to ARexx, so it actually transfers control AWAY from your program, and to the DATE() function code itself. The DATE() function takes one additional parameter, to define what sort of date type you'd like. We'd like the normal date, DD MMM YYYY (for example, 10 Jan 1994), so we specify NORMAL. When this function completes, it actually returns to us in the form of a string, the correct date, which is then assigned to our variable "the_date". Then, we can use SAY to show this on the screen. So long as your date is set correctly and you have a real-time clock, running the above program will indeed show the correct date on the screen. We have now called our first function.

It's important to realise that functions are simply small programs, that may optionally take some parameters to describe which operation they are to take, and may or may not return a result to your program. There are three types of function in ARexx, and they are listed below in their search order. (See the section below: 'a final note on functions')

Internal Functions: These are functions that you write yourself, in ARexx, in your own program. See the section below: 'Internal Functions'

Built-in Functions: These are functions provided by ARexx for your use and are built in. Their names are in upper case, like the one we used above called 'DATE'. There are over 80 of these in total, and they are very fast and efficient. See the reference section (E) for a detailed description of all these functions, or below for further examples of their use.

External Function Libraries: In addition to the built-in functions, it's possible to add other functions to ARexx. One such library, called "rexxsupport.library", is supplied with ARexx and, once it has been added, provides a range of special functions for the advanced ARexx programmer. We'll discuss how to add such external libraries later. As well as the supplied library, other third party developers, particularly in the public domain and shareware arena, have written external function libraries.

As well as the three types of function listed above, there is one other type of external function, and that is one supplied by an external host or, to put it in English, supplied by another program entirely separate from ARexx which is able to make functions available. Documentation about these functions is supplied with supporting applications, but they work like the built-in and external functions.

As well as external function hosts, it's worth mentioning that the power of ARexx is extended considerably by the addition of host applications and their associated commands. Section C is devoted to discussing this powerful ARexx feature.

Internal functions

One of the key parts of modern, organised structured programming is the ability to call sub-routines and functions. As we have discussed above, when you call a function, program control is transferred away from where you were when you called the function and goes to that function. When the function is completed, then your own program will continue where it left off. In the case of internal functions that you might write, there are two basic types of function:

- Those that perform an action, but do not return a result.
- Those that perform an action and then return a result. These are "real" functions, in that some kind of value is returned to you.

Both of these types may or may not require some parameters in order to work. We call these "function arguments". Here's an example which demonstrates a function of the first type, without parameters, which shows a message on the screen stating that an error has occurred:

```
/* Function test */

CALL ShowError()
EXIT

ShowError:
  SAY "An error has occurred."
  RETURN
```

We've introduced a few new concepts here. First, you may recall at the start of this section we used the ARexx statement "EXIT" to leave a program and then stopped using it because, as I explained, it was not really necessary because in all of our programs the point when we wanted to stop running them just happened to be the last line. In the above example, this is no longer the case. The first line of the program uses the statement CALL to call a function named "ShowError". This function takes no parameters. Then we exit our program.

So, what happens when the the function call happens? Well at this point ARexx searches to see if we have defined a function of the name ShowError(). We have done this, at the end, by stating the function name with a colon at the end. All statements after this identifier can be considered part of the function itself until a RETURN keyword is reached, and at this point, control goes back to where the function was originally called. When we run this script, predictably, we get the result:

```
7.System3.1:> rx function
An error has occurred.
```

Well, it seems a pretty pointless function. It takes no parameters, produces a fixed non-flexible result, shows it directly to the screen for us, and returns no information at all. It's hardly general! So why would we need such functions? In the above example, it would have been much simpler just to have the line:

```
SAY "An error has occurred"
```

...whenever we needed it, and save having to write a function to do it. Good point. But suppose that, as well as printing the message "An error has occurred", we also wanted to show the time and date when the error happened, and format the result neatly too.

Well, by now we're up to a couple of lines; we have to read the time, and the date and produce a result. If we wanted to do this, say, 30 times in our program, it would mean a lot of repetition. And what if we wanted to change the format? Get rid of the date and just show the time? Well, we'd have to search for, and painstakingly change, 30 small chunks of code – and that's asking for trouble. This is an occasion when you need a function to do the job. That way the routine only appears once in your program, but you can use it as often as you like, and if you want to change it, you only have to change the function itself.

Let's make our error-reporting function more useful. It would be nice if we could specify what the error actually was, and perhaps put the time of day somewhere as well, so we know when the error occurred. In order to show the error name, we're going to need an argument ("ARG"):

```
/* Function with argument test */
CALL ShowError("Out of memory")
EXIT

ShowError:
  SAY "Error at" TIME("CIVIL")
  SAY "  >" ARG(1)
  RETURN
```

So what happens now when we run this program? Well, first we call our function as normal, but this time we are specifying one argument – in this case, the string "Out of Memory". The ShowError function uses the ARexx built-in function TIME in order to print the time of day on the screen, and then shows the error name itself by using another built in function – ARG. Since we are only going to deal with one argument, we specify ARG(1), which is the first. Then, having shown this information, we exit.

The results of running this particular script are:

```
.System3.1:> rx function
Error at 1:17AM
  > Out of memory
```

This function may now be finished. But in order to make it even more flexible, instead of showing the error message directly on the screen, we might return it as a function result, allowing the caller to decide what to do with it. After all, it might be logged in a file and not necessarily shown on the screen.

With this in mind, we might change our ShowError function to look a little like this:

```
ShowError:
  error_string = "Error at " || TIME("CIVIL") || ": "
|| ARG(1)
  RETURN error_string
```

We've introduced another new concept here, the concatenation of strings. We're building a string variable called "error_string". We use the concatenation operator || to join strings together in this way, joining the string "Error at " to the current time, a single colon, and then the first argument of the function. After building the string, we return it. Previously we've just used RETURN on its own but, by specifying an expression, the value of that expression is returned to the caller. This means, we can call it like this:

```
SAY ShowError("This is MY error")
```

Or...

```
my_error = ShowError("Errors! Errors! Everywhere! But ¬
not a bug in sight!")
SAY my_error
```

Note that we are using our function in the same way we have used the built in DATE() and TIME() functions in previous examples. So what happens to the variable "error_string" which we define inside our ShowError() function? Surely, we could call the function, and just do:

```
SAY error_string
```

...and ignore the result of the function altogether? Well, we could – but this is not good programming practice and indeed could be the cause of problems in the future. If our program uses a lot of variables it's possible that some might clash with variables inside other functions, or in the main program, and cause problems.

OK, so this would not be a disaster by any means – we would simply ensure that all our variable names were different. However, a better way to handle this kind of problem is to ensure that the variables which a function uses are totally local to itself and can not affect any other variables in the program, even if they were to share the same name.

Another, good side effect (although you might not think so while you're getting to grips with programming!) is that it forces you to program properly and adopt good healthy programming practices. It makes you write your functions with more care, ensuring that all the information they require is provided in arguments, and that the result of the function, if any, is returned using RETURN – rather than generating spaghetti programs where functions steal values out of your main program. A bad habit which can easily be the cause of many program failures.

So how do we make a function's variables all local to it in this way?

Procedure and Expose

Let's write a small test program:

```
/* Procedure test */

a_variable = "Chocolate Pudding"
CALL Test()
SAY a_variable
EXIT
```

```
Test:
  a_variable = "Strawberry Cheesecake"
  RETURN
```

As you would expect, when we run this program we get the following result:

```
7.System3.1:> rx function
Strawberry Cheesecake
```

Although we set the variable "a_variable" to Chocolate Pudding at the start of the program, when we call our function Test() it sets it to Strawberry Cheesecake which, apart from not being nearly as nice as Chocolate Pudding, may not be the desired result. Let's make a small change to the program in the first line of our function. From

```
Test:
```

... to this:

```
Test: PROCEDURE
```

... and then run the program again.

```
7.System3.1:> rx function
Chocolate Pudding
```

Now that's more interesting. This time, when we alter the variable "a_variable" from within our function, we are altering our own version of it, which has no effect on the main program's variable at all. All variables within a function which is declared as a PROCEDURE in this way are local to the function. When you exit the function, using RETURN, its contents are lost.

However, all this good programming is great in theory, but what happens when we want to affect a main program variable in some way? Or indeed, since RETURN only allows us to return one value, what if our function returned several? Surely it would be much better to store the results in variables that the main program could read? Well, yes. There *are* occasions when a function might want to alter main program variables in this way, but not often. ARexx provides a proper method for doing this, and it's best to use it sparingly. There are *very few* occasions where you will be forced to do this and, in general, it can result in messy, unreliable code which does not comply with modern modular programming techniques. On the same line as the PROCEDURE keyword and function name you can also specify a list of variables which we want to share with the caller, using the EXPOSE statement. Let's alter our original example to show Strawberry Cheesecake using EXPOSE:

```
/* Procedure test */
a_variable = "Chocolate Pudding"
CALL Test()
SAY a_variable
EXIT

Test: PROCEDURE EXPOSE a_variable
  a_variable = "Strawberry Cheesecake"
  RETURN
```

The result of running this program is "Strawberry Cheesecake" because, since we are now sharing the variable "a_variable" with the main program, if we alter it from within our function we also alter the main one. All other variables we might define and use within our function remain local. Only those we name as EXPOSE'd are shared with the function caller.

So when might you need to use EXPOSE?

- For returning more than one value from a routine. If you wrote a graphics routine, for example, which returned an X and a Y co-ordinate, EXPOSE could be used to allow you to put values into an X and Y variable that the main program could read.

- Accessing major global variables. In large programs there is often a collection of important global information which applies for every routine, and frequently it would become silly to put in every bit that might be needed as parameters for functions. Alternatively there might be a single data structure, such as a file containing large numbers of records which might need to be accessed globally. In this case, EXPOSE can be useful.

In the case of both the above examples of places you can use EXPOSE, there are ways around it which would mean you don't have to. But let's face it, there is a limit to how far you can take "good programming" before it stops becoming fun and becomes seriously tedious. The best policy is, when you think you might want to use EXPOSE, stop and think again. Ask yourself "Do I really need to do this?" and if the answer is yes then it's probably OK.

Using the built-in functions

There are over 80 built-in functions in ARexx and the reference section (E) lists them all in alphabetical order. It's obviously beyond the scope of this programming guide to show you a use for every single one, but we'll certainly look at a handful of them and give some examples of their use.

Once you have read this section, you might find it interesting to have a browse through the list of functions at the end of this book to see for yourself.

The built-in functions all return some information to you and, for some functions, you may not have to bother checking the result. The kind of information returned varies dramatically, and ranges from time date, or time in various formats (which we have already encountered) to Boolean results which indicate whether a function failed or succeeded – TRUE or FALSE – and, if you recall from our discussion of Boolean results at the start of this section, in ARexx's case this will either be a 1 for TRUE or 0 for FALSE. With Boolean results, we can normally assume that FALSE means that the function failed, and TRUE that it succeeded, but it is always best to check the function's documentation if you're not sure.

Types of functions available

The built-in functions have been chosen carefully to provide a wide range of operations that will be useful in day-to-day programming. We have only used a few of them so far in our programs – TIME, DATE and ARG. There are many more, quite a few of which will be introduced later in this section as we get to external functions, arguments, files and advanced programming techniques.

Some of the most awkward things to deal with in any computer language are strings. We've used quite a few examples of strings so far, mostly to hold names or in the use of compound symbols. In languages such as C, string handling can be a pain at the best of times and C provides a range of basic string-handling functions for the programmer but, if you want to do anything remotely complicated, you have to write your own routines. ARexx, on the other hand, has stacks of very flexible string processing routines which will do everything from extracting parts of strings to automatically stripping out unwanted characters.

Let's try an example:

```
/* String functions test */
SAY "You are in a dark room. What next?"

/* Enter the command */
words = 0
DO UNTIL words = 1 | words = 2
  PULL command_name
  words = WORDS(command_name)
  IF words = 0 THEN SAY "You entered nothing"
  IF words > 2 THEN SAY "Too many words, expecting ¬
VERB NOUN"
  END

/* Process the words */
SELECT
  WHEN words = 1 THEN
    SAY "You entered one word, it was" ¬
WORD(command_name, 1)
  WHEN words = 2 THEN
    SAY "You entered two words, they were" ¬
WORD(command_name, 1) "and" WORD(command_name, 2)
  END
```

This is a simple part of an adventure game, the bit where you have been given the description of the room and now have to enter a command.

Imagine that in our simple adventure, written entirely in ARexx, we only accept simple VERB/NOUN structures, such as "TAKE APPLE", and "GO NORTH" and, in some circumstances, single words like "LOAD" to load a game off disk. The above example achieves this in ARexx by using the WORD and WORDS functions.

First we use a DO UNTIL loop to continue getting a command until the user inputs one with one or two words. We input a string from the user using PULL as normal, and then use the WORDS function to return the number of words. Our two IF statements deal with the two illegal cases – no words at all and too many words.

We then use a SELECT to perform operations depending on how many words were entered, and in this simple case, simply show them back to the user using the WORD() function, which takes two arguments, the string and the word number we want. If we were to run this we might get the result:

```
8.System3.1:> rx adventure
You are in a dark room. What next?
take the apple and give it to the old man
Too many words, expecting VERB NOUN
go north
You entered two words, they were GO and NORTH
```

In addition to string functions, there are many numerical ones. Let's demonstrate a couple with an example:

```
/* Numerical functions test */

/* Pick a nice seed */
our_seed = TIME("SECONDS")
SAY "Random seed was" our_seed

/* Pick a random number */
my_number = RANDOM(1,5,our_seed)

/* Get the users guess */
SAY "I am thinking of a number from 1 to 5. Make a
guess."
PULL users_guess
IF users_guess < 0 THEN DO
  users_guess = ABS(users_guess)
  SAY "You entered a negative number. I've converted it
```

```
to positive"
  END

/* Print results */
IF users_guess = my_number THEN SAY "Well done!"
  ELSE SAY "Bad luck."
```

This is our first game. Hardly a blockbuster, but it works. The computer picks a number between 1 and 5, and you have to guess it. Now let's look at the functions we've introduced. Note that we read the SECONDS value from the clock. This is the number of seconds since midnight, and one of the many handy things you can use the TIME function for. We're using this to seed the random number function so it will generate a different number each time.

RANDOM takes up to three parameters: a minimum and maximum number, and a seed. It then returns a random value between the minimum and maximum. The default minimum value is 0, and maximum is 999, so if you tried:

```
SAY RANDOM()
```

...you'd get a value from 0 to 999 inclusive. Once we have been given the computer's guess, we ask the user for a choice of their own. If they enter a negative number we use the ABS function to convert it to a positive one, so if you entered -3, it would be converted to +3 by the ABS() line. Finally, we then see if the user's guess was correct and show an appropriate message on the screen. Running this program would produce a result like this:

```
8.System3.1:> rx numeric
Random seed was 60577
I am thinking of a number from 1 to 5. Make a guess.
2
Well done!
```

```
8.System3.1:> rx numeric
Random seed was 60580
I am thinking of a number from 1 to 5. Make a guess.
3
Bad luck.
```

Well, you win some and you lose some. Note the seed value. Since we happen to know it is elapsed seconds since midnight, you can see that there was a three second delay between the first and second running of the script, and armed with a calculator you can see that it was 16:49 when the programs were run. Alternatively, we could just do SAY TIME("NORMAL") – but that's the easy way out!

Using external function libraries

As well as built-in functions, you can add external function libraries. An external function library is a special program in the form of an Amiga shared library. You don't need to worry too much about this, except that it will be found in your LIBS: drawer. You will already have many libraries and you can see them by opening a shell and typing "dir libs:".

You can't add just any old library and expect it to work, you have to add ARexx external function libraries. There is one which comes with ARexx called "rexxsupport.library". But before we can use any of the functions in it we have to tell ARexx to add the library to its search list.

We can do this in either of two ways: from the shell using the utility command "RXLIB" (see the reference section for further information), or by using the built-in function ADDLIB. Adding it from the shell is easy:

```
8.System3.1:> rxlib rexxsupport.library 0 -30 0
```

RXLIB means "add a new external function library" and it takes four parameters: the name of the library itself (which *is* case sensitive), a priority, an entry point and a version number. The priority can be important, because when ARexx comes to see if the function it has belongs to any of your external functions libraries, it will check with them in priority order. Priorities run from 100 (high) to -100 (low). For example, suppose you had two libraries – called fredrexx.library and tyranosaurusrexx.library – and they both contained the function "DINOSAUR". If fredrexx.library was at a higher priority, then its DINOSAUR function would be executed, and the other one would not. Normally it's best to add libraries at a priority of 0 unless you have problems with conflicting function names.

The entry point is a special value for libraries which tells ARexx where to find its information. Look at the documentation that came with any libraries you might have for this data. In the case of rexxsupport.library, this value is -30. The final parameter is the library minimum version number. If this is not important, specify zero.

As well as adding external libraries using the shell command RXLIB, we did mention that it can be done using the built-in function ADDLIB. Here's an example:

```
/* Addlib demo */
IF ~ADDLIB("rexxsupport.library",0,-30,0) THEN DO
  SAY "Cannot add this library"
  EXIT
  END
SAY "Library added. Ready for use"
```

A reason why ADDLIB might fail is if the library is already available. Adding a library using ADDLIB or RXLIB does not load it off disk and make it immediately ready to use, it simply adds it to ARexx's library search list. Then, when a function is used that is not internal, or built-in, ARexx opens each of these libraries in turn, in their priority order, sees if the function belongs to that library and, if so, runs it. You can remove a library using the function REMLIB.

What's in the rexxsupport.library?

The short answer is, a lot of complex, advanced functions that can get you into a real pickle very quickly unless you're quite sure what you're doing. When using REXX's built-in functions and keywords it's fairly unlikely that you could crash the computer no matter what you did and, to a certain extent, it would also be quite hard to corrupt files – unless you were exceptionally careless. Once you get to the rexxsupport.library, however, it's not at all difficult to crash your computer completely. The moral of this lecture, of course, is that you should be careful. Read the reference section on rexxsupport.library (Section E) very carefully before using any of its contents, and make sure you understand what they do first.

Unlike ARexx's normal functions, some of those in rexxsupport.library are not resource tracked. This means that ARexx does not know what you're doing, and therefore can't un-do any mess you make when your program exits. This is particularly true for memory allocation. Should you need to allocate any memory to perform a particular function using the rexxsupport.library, you *must* free it before exiting your program, otherwise it will never be freed – and you will slowly run out of memory until you are forced to reset your computer.

Lectures aside, there are some extremely handy functions that you can use, such as SHOWDIR, with which you can fetch directories off your disks, and SHOWLIST which builds lists of all sorts of useful system structures. For example, you can use SHOWLIST to generate a list of waiting tasks currently on your computer, or all the

ASSIGNs you might have. It will generate many lists of this sort. Let's use an example to demonstrate these:

```
/* rexxsupport.library demo */

IF ~SHOW("L", "rexxsupport.library") THEN DO
  IF ADDLIB("rexxsupport.library",0,-30,0) THEN
    SAY "rexxsupport.library added"
  ELSE DO
    SAY "rexxsupport.library not available"
    EXIT 10
    END
  END

SAY "Here is a directory of your SYS: drawer:"
SAY SHOWDIR("sys:")

SAY "And a list of assigns:"
SAY SHOWLIST("A")
```

Whether you have a lot of assigns and maybe a hard disk or two, will dictate just how much output you get on the screen. You will see that both functions return one long string, with each item separated by one space. You could now use the WORDS() and the WORD() function we used in the earlier section on built-in functions to process this string or, by more advanced use of the SHOWLIST/SHOWDIR function, you can get it to change the space to an alternative character. SHOWLIST is particularly handy because it lets you see if a given name is in a given list. For instance, you could check if a particular task was running, or an assign existed, and report to the user accordingly. In this case, SHOWLIST returns a Boolean TRUE/FALSE rather than a string to indicate whether the name was in that list.

A final note on functions

Now you have been introduced to the various kinds of functions, there are just a couple of things to add – the first of which is the search order. When ARexx comes across a function it first checks to see if it is an internal function – one that you have written yourself. Then, if it can't find it there, it checks against its built-in functions and, if there is still no luck, it moves on to the external function libraries in priority order (remember that when you add a library, such as rexxsupport.library, you specify a priority), and external function hosts. Finally, should it not find it in any of these places, it checks to see if your function name is an external ARexx program, searching in the current directory first.

One of the advantages of all this to the programmer is that since internal functions are checked first, you can override any of the built-in or external functions and replace them with ones of your own, just by using the same name. This means that ARexx will come to yours first, and simply stop searching – it never gets as far as the built-in and external function lists.

A second minor point concerns case sensitivity. The built-in functions, for example, are normally referred to using upper case, but are not actually case sensitive – nor are those defined in the rexxsupport.library. Some external function hosts, however, might be case sensitive. Check with the documentation that came with an application before making any assumptions.

Program Arguments

We've seen that it is possible for functions to have many arguments. However, programs run from the shell using the RX command can only have one argument – although you can PARSE this yourself into separate parts using the built-in string functions, or by using the PARSE instruction (see the following section for more detailed information on PARSE).

Let's use this little program example, which takes a string and shows us some statistics about it:

```
/* Program arguments demo */
PARSE ARG line

SAY "Argument is:"line
SAY "It has "WORDS(line)" words in it."
SAY "The first word is "WORD(line, 1)
IF DATATYPE(line, "BINARY") THEN
  SAY "It was a valid binary string!"

SAY "The string is "LENGTH(line)" characters long."
```

Note the use of the DATATYPE function to check if the argument was a valid binary number. A binary number consists of 1's and 0's. See the section 'Advanced programming techniques' for more information. Running this could produce:

```
8.System3.1:> rx parse fred is a ginger cat
Argument is:fred is a ginger cat
It has 5 words in it.
The first word is fred
The string is 20 characters long.
8.System3.1:> rx parse 1001
Argument is:1001
It has 1 words in it.
The first word is 1001
It was a valid binary string!
The string is 4 characters long.
```

Program arguments are useful when you want to write Arexx scripts which take parameters. There are examples later in the book.

The PARSE statement

The PARSE statement is an incredibly complex instruction designed to extract one or more sub-strings from a string and assign them to variables. Everything it does can be done using the built-in string functions, but since PARSE is more generalised, it's often better to use it than to manually PARSE strings yourself – not just because it is quicker, but because you are less likely to make errors. PARSE is particularly useful for processing strings input by the user from the console window, or from function arguments. However, some of its more complex operating modes can take a bit of understanding.

There are many occasions where you might want to use PARSE, because of the powerful way it allows you to strip down strings into component parts and get those parts into variables which you can use.

So how does it work? PARSE takes a string and then, by searching for markers, breaks it up and puts the parts into variables. You can specify where and what these markers actually are, and how many target variables are used, by specifying a template. The PARSE statements format looks like this:

```
PARSE [UPPER] inputsource [template] [,template...]
```

Items in square brackets are optional. The UPPER keyword, when specified, ensures that the input string is translated to upper case before being PARSE'd. Following this is the input source. This describes where the string that is going to be PARSE'd is coming from. It can be from many sources, such as arguments of a function by specifying ARG (we've seen this in use in the earlier section on Program Arguments), or from a variable using VAR variable_name. Finally, after the input source comes the template itself. This tells PARSE how to break up the string received from the input source, and which variables to put the parts into.

So what does a template actually look like? Well, it consists of a list of target variables broken up by markers. In its simplest form, this is just a variable list separated by spaces. In this case, the spaces are treated as markers. PARSE searches the string from left to right for a space and assigns the section of the string to the left of it to the first variable, before going on to the next... until it runs out of spaces. Let's illustrate this with an example:

```
/* Parsing example */

our_string = "This is our string"
PARSE VAR our_string one two remainder
SAY "*"one"*"two"*"remainder"*"
```

When run, it produces this result:

```
8.System3.1:> rx parse
*This*is* our string*
```

Let's talk through this. First, we set up our string variable to contain something sensible. Since we'd like to PARSE from a variable, we specify PARSE VAR and the variable name itself. After that we need to specify the template, which is just three variable names; one, two and remainder. Then, when we have completed the PARSE, we show the results one after the other with *'s between them so we can see exactly what the results are. It should be pretty obvious what has happened here, except perhaps for the result of the last variable, remainder. There is an implied marker at the end of a string and when PARSE meets that it puts the remainder, whatever that might be and including the space, into the last target.

If the input source string is used up before all the target variables in the template are filled, then the remaining targets are assigned null values. As well as specifying markers within a template, with some PARSE usages you can also specify more than one template. Templates are separated using commas. This is handy when using PARSE ARG, and PARSE PULL.

Templates and pattern markers

So far we've looked at a basic template with a simple list of target variables, and PARSE assumed that a space was a marker when parsing the source string. PARSE becomes particularly clever when we start specifying markers.

Let's illustrate this with an example in which we read the current time, and then separate it into hours and minutes and decide whether it's am or pm:

```
/* Parsing example */
current_time = TIME("NORMAL")
PARSE VAR current_time hours ":" minutes ":" seconds

SAY "Hours:"hours
SAY "Minutes:"minutes
SAY "Seconds:"seconds
```

When you run this program, depending on the time of day, the result will be something like this:

```
9.System3.1:> rx parse
Hours:19
Minutes:54
```

```
Seconds:24
```

So it's 7.54pm. With a PARSE template, any strings specified are always markers. We specified two markers, both colons, because the TIME('NORMAL') function returns a time in the format HH:MM:SS, that is, hours, minutes, seconds separated by colons.

Absolute markers

In the above examples, the targets have all been of variable length. In our time example, if the TIME() function had returned "0:0:0", we'd have still got valid values out. There are occasions, however, where you might want to be able to specify fixed markers – for instance, the first target might be exactly 10 characters long, and the next four, and the next three. This is where absolute markers are especially useful.

One excellent application is in the area of fixed-length records in files. Suppose you have an address book application where you have the name of the person 20 characters long, the address 50, the postcode 10, and finally the phone-number, another 20. Using fixed-length markers, you can extract this information very easily. We can demonstrate a simple use of fixed-length markers with another TIME() example. In a different mode, time returns the time of day in a 12 hour clock, with an am/pm after it. It might look like this:

```
1:51PM
```

With our current marker system we have nothing to search for to separate the minutes from the am/pm flag, but there is one key thing – the minutes are *always* two digits, and am/pm is also always two characters. So, armed with absolute markers, we can write a new program:

```
/* Parsing example */
```

```
current_time = TIME("CIVIL")
PARSE VAR current_time hours ":" minutes +2 ampm +2

SAY "Hours:"hours
SAY "Minutes:"minutes
SAY "AM/PM:"ampm
```

When this is run, depending once again on the time of day, we will get a result like this:

```
.System3.1:> rx parse
Hours:11
Minutes:15
AM/PM:PM
```

How did this work? And what did the +2 and +2 mean? Well, absolute markers can also be relative. We can say move on two characters, or move back four characters. As well as moving relative distances from our current position, we can also move to a fixed point. Positions are measured from 1, so be careful. For instance, if we had a line that we wished to PARSE, which was defined as this:

```
whisky_name = "Springbank          2545Extremely ¬
nice winter warmer           "
```

This might be for our whisky database. The name of the whisky is 20 characters long (which is why there are spaces after the word "Springbank"), then there is the two-digit number of years old it is (if less than 10 it has a leading zero), then the percentage volume of alcohol, also two digits (rounded up if there is a fraction), and finally a 40 character description. We could PARSE this information out like this:

```
/* Parsing example */
whisky_name = "Springbank          2545Extremely ¬
nice winter warmer           "
```

```
PARSE VAR whisky_name name 21 years 23 alc 25 desc 65

SAY name
SAY years
SAY alc
SAY desc
```

The results, as you might expect, are:

```
9.System3.1:> rx parse
Springbank
21
45
Extremely nice winter warmer
```

This time we used fixed absolute markers. We stated the exact positions at which the targets begin, bearing in mind that since we start at 1, something 20 characters long ends at 20, not 19.

Parsing user input

We have already used the PULL instruction in ARexx to get information from the user. As previously mentioned, PULL is actually shorthand for PARSE UPPER PULL, which is why everything we enter always seems to end up in upper case characters. If we wanted to get a string from the user which was not converted, we could use the long-hand version of PULL, but without the UPPER:

```
/* Parse PULL example */
SAY "Type something in lower case"
PARSE PULL something
SAY "You entered:"something
```

Of course, you could specify a full template and PARSE the string inputted as you would with other PARSE operations.

Parsing arguments

In previous examples we have seen both the built-in function ARG(), and the PARSE ARG functions used to process arguments. Now we have dealt with PARSE in more detail, you can see that it is possible to PARSE arguments to a function in a comprehensive way, using templates, all in one line which may not be possible using the ARG() function. The ARexx keyword ARG, which you will encounter in the reference section (E), is shorthand for PARSE UPPER ARG, and of course, converts all arguments to upper case, which you might not necessarily want.

Extracting environment information

Although not particularly useful in day-to-day applications, it is possible to extract some general information about the environment in which you are running, using PARSE VERSION and PARSE SOURCE. By using PARSE VERSION you can get information about the actual system you are running on, and the version of the ARexx interpreter in use:

```
ARexx VERSION CPU FPU VIDEO FREQ
```

Where VERSION is the ARexx interpreter version, CPU is the processor type, FPU is the floating point unit (or NONE if not present), VIDEO is either PAL for UK and European systems or NTSC for American systems, and FREQ is the mains frequency – normally 50Mhz. For example:

```
/* Version Information */
PARSE VERSION whole_lot
SAY whole_lot
```

Which, depending on the type of Amiga you have produces something along these lines:

```
9.System3.1:> rx version
ARexx V1.15 68030 68881 PAL 50HZ
```

Of course, as with other PARSE usages, you could specify a template to extract, for example, just the processor type, or video type. PARSE SOURCE is generally more useful, and can be used to find information about how your script was called, where it was called from, and whether a result is going to be required from it or not. The format of the returned string looks like this:

```
{COMMAND | FUNCTION} {0 | 1} CALLED RESOLVED EXT HOST
```

This will tell you the following information. You will be shown COMMAND or FUNCTION, which tells you if your program was executed as a function or a command. The number which follows states whether a result is expected from your program or not. This is a Boolean flag, so 0 would mean no result was expected. Then you get the name used to run your program, normally the script name without a .rexx at the end. The resolved section shows you the final filename used to run your program, and host shows the initial host address for commands. Here is an example of what might happen if you simply had a script which did PARSE SOURCE variable and then SAY'd the result:

```
8.System3.1:> rx source
COMMAND 0 source Development:ARexx/source.rexx REXX REXX
```

This tells us that we ran our program as a command, no result was expected, we were called as "source", our complete file path, the default file extension is .REXX, and the initial host for commands is REXX.

Final notes on PARSE

Try not to think of PARSE solely as a mechanism for processing arguments and user input. Think of it as a powerful string processor. It has uses other than those described above – for a complete run-down of the PARSE instruction, take a look at the reference section. In the meantime, as one final example of an application of PARSE, the following program takes a string and strips each word out, one by one, until there are no more, and shows them on the screen:

```
/* Parse word stripper example */
words = "These are my words, Oh YES they are!"

DO WHILE words ~= ""
  PARSE VAR words single_word words
  SAY "Word was:"single_word
  END
```

To explain briefly, we have a DO WHILE loop which continues while the words string still has some content, and we use the comparison operator not equal to (~=). We then PARSE the string words into two targets – one contains the first word, while the other contains the rest of the string. We just happen to PARSE the rest of the string back into the words variable, so next time round words has one word less, and we've just PARSE'd and shown that word on the screen. Have a go at it yourself... Clever isn't it?

Files

When we talked about compound symbols earlier in this section, we mentioned such terms as "files", "records" and "fields" when talking about address books or other such applications. The problem with all of our examples was that the information we were processing was held in the computer's memory, so when the program finished, the file was lost. Ideally, of course, we would be able to save out our data to floppy or hard-disk and recall it next time around, so that we could then write a really useful address book application.

It's a fact of life that there are very, very few programs that do not require input or output. In fact any that don't can't actually do that much. That input could come from the keyboard, like the examples we have used, and the output could be to the screen, but in a lot of cases programs need to be able to store and recall information on a more permanent basis. This is where we start dealing with file access.

What exactly is a file? Well, if you want to see a whole load scroll past on your screen, open a shell, and type:

```
dir sys: all
```

Everything that doesn't have the word (dir) after it is a file. There are lots of them aren't there! A file is simply an area on your disk that holds information of some kind. We can refer to a file by name. A file can hold a program, or perhaps some data for a program (such as a document you have saved from a word processor).

ARexx contains some special built-in functions to allow you to deal with files.

WARNING **But be warned: only deal with files you know are yours to deal with. If you alter files that belong to other programs, particularly executable programs off your disk, you can damage them. Another point – OS files, such as those which contain your preferences settings, will change in the future. So if you write a script which fiddles with these files and works right now, it may not work in the future and could have disastrous consequences.**

Dealing with files

When you wish to either read information from a file or write information to it, you have to first "open" it. When you open a file you are issued with a magic number called a file handle. This is your key to the file. Whenever you wish to perform any operation on it, you have to quote your filehandle, so that ARexx knows which file you are talking to. When you have finished with your file, you are expected to close it. This frees up any system resources which may have been used and ensures that all data that you were writing to that file is totally written.

In theory, you could have a whole load of files open at a given time and, as long as you quoted the right handle at the right file, you would be able to perform actions on them all at once.

With languages such as C the file handle is a fixed number. In ARexx you are able to decide what you'd like to call that file by – in effect, to name your own file-handle.

When you open your file, you specify what your file handle is going to be, what the filename is, and what you intend to be doing with that file. The last bit is optional but if you omit it, the file will be opened for reading. You can open a file in one of three ways: for reading, for writing, or for appending. If you open a file for writing, it is created as a new file, and the old copy of the file (if there was one) is deleted – so be warned. If you open a file for reading, then when it's opened you are positioned at the beginning of the file. If you open a file for appending, you are positioned at the end of the file.

READ and APPEND are essentially the same (they both open an existing file), the difference being the position you start from. The confusing thing is that even if a file is opened in READ mode, you can still WRITE to it – and so can other programs! If you open a new file yourself, you have exclusive access to it until you close it. But if you open a file for reading or appending, you share it with the rest of the system. Usually this won't be a problem because, if you are dealing with your own files, you are normally the only user!

So, how *do* we open a file? Let's look at the following example, which creates a new file in the RAM disk and writes a string to it:

```
/* Open example */
IF ~OPEN("WriteFile", "RAM:test.file", "WRITE") THEN DO
  SAY "I can't open my file!"
  EXIT
  END

SAY WRITELN("WriteFile", "This is a test!")

CLOSE("WriteFile")
```

When run this program creates a file called ram:test.file. Here is the result of running it, and also of using the shell command TYPE to see what the file contents were when the program finished:

```
7.System3.1:> rx open
16
7.System3.1:> type ram:test.file
This is a test!
```

Let's look at this program in detail. The OPEN function returns a Boolean value – TRUE for success, FALSE for a failure – so we use the NOT operator "~" so that, by using an IF statement, we can detect if the OPEN failed. We are going to call our file "ram:test.file", and it is going to be a new file created from scratch. Our file handle will be "WriteFile". If we successfully open our file, we call the WRITELN function, which is built in. This takes two parameters: the file handle, and the string to write to the file. It returns the number of characters that were written.

Finally, we close the file. You can see that when we ran the program, it reported that 16 characters were written to the file, the 15 characters in the string, and an end of line character, which has the ASCII code of 10 (New Line).

If we change the file access mode from WRITE to APPEND, and re-run the program, then an extra line is added to the end. This is because the APPEND mode opens the file and positions us at the end of it, ready to add more data. As well as the WRITELN function, there is one called WRITECH, which operates in the same way, but does not add the end of line character.

There are corresponding READ functions to complement the WRITE functions, READLN and READCH. This example counts the number of words in a file, using READLN and other functions we have used earlier in this section:

```
/* Word counter */
PARSE UPPER ARG file_name

IF ~OPEN("WordFile", file_name, "READ") THEN DO
  SAY "I can't open the file "file_name
  EXIT
  END

total_words = 0
DO UNTIL EOF("WordFile")
  this_line = READLN("WordFile")
  total_words = total_words + WORDS(this_line)
  END

SAY "Words in file:" total_words

CLOSE("WordFile")
```

This program uses several of the more advanced features we have already come across. It takes a parameter which is the filename to work on, opens the file for reading, reads each line in turn, counting the number of words in it, and continues until the end of file has been reached. The EOF() function returns TRUE if the end of file has been reached for the quoted file-handle. When we're done, we simply show the total on the screen and then close the file. Here's the result when used on my startup-sequence:

```
9.System3.1:> rx wordcount s:startup-sequence
Words in file: 161
```

As well as the functions we have seen for reading, writing and detecting the end of file, there is one other that is necessary when working with complex files – particular those that are made up of records with fixed-length fields (like the whisky example we used when dealing with fixed markers in the PARSE section earlier). This function is SEEK, and it allows you to position yourself in the file relative to the current position, the beginning, or the end of the file. Here are three examples. They are not part of a full program, but are shown so that you can see how they might be usefully employed:

```
SAY SEEK("FileHandle", 10, "BEGIN")  /* Seek to ¬
character position 10 from the start of the file */
SAY SEEK("FileHandle", 0, "END") /* Move to the end ¬
of the file */
SAY SEEK("FileHandle, 10, "CURRENT") /* Move 10 ¬
characters forward */
```

Seek returns the new position that you have reached in the file. It takes up to three parameters. The first is the file-handle we are talking to, the second is the offset from the anchor position, and the third parameter – which is either the BEGINning of the file, the END of the file, or relative to the current position. If no anchor position is specified, then it is assumed to be CURRENT.

Signals, tracing and debugging

Programmers of compiled languages, such as C, are used to dealing with their own errors in an orderly fashion. ARexx programs, however, are interpreted and if an error occurs during execution the program will stop and an error will be shown in the console window – usually the shell. This may not always be a good idea – there comes a time with larger ARexx programs when you may wish to deal with these conditions yourself. ARexx provides a facility for programmers to do this by using the SIGNAL statement. A number of things can happen which will cause an ARexx program to stop, but they are all controllable using SIGNAL. It is possible to decide which of the conditions you wish to intercept. For example, one of the signal conditions is called "SYNTAX" and it occurs when a syntax or program execution error occurs. To catch this yourself, you simply use the statement:

```
SIGNAL ON syntax
```

...and then provide an internal function called syntax, perhaps like this:

```
syntax:
   SAY "An error occurred, you silly billy. It was ¬
   "ERRORTEXT(rc)
   EXIT
```

When a syntax error occurs, ARexx jumps to your syntax: function. When control arrives at your function, the variable RC contains the error code and there is a built-in function, called ERRORTEXT() which shows the appropriate error string. In the above example, we simply exit. It is possible, however, to then return control back to a sensible point, by using the signal statement in a special form:

```
SIGNAL label_name
```

This is the ARexx equivalent of the "GOTO" instruction found in many popular programming languages, such as C and BASIC. It means "jump immediately to the named point in the program". The label name refers to a name with a colon after it, just like an internal function definition. For example:

```
/* Goto from the depths of... */

a_label:
SAY "Hello"
SIGNAL a_label
```

This just continues forever, just like a DO FOREVER loop. GOTOs are generally frowned upon, and quite rightly so – they encourage messy, disorganised programming which leads to bad habits. However, there are occasions, such as those which arise from error handling using SIGNAL, where you have no choice. But use SIGNAL in its GOTO format sparingly. The only way to exit from the above example program is to use the CTRL-C sequence (hold control down, tap C, release control). Of course you can trap this too, using SIGNAL, to present a gloriously un-exitable program:

```
/* Goto from the depths of... */

re_start:

SIGNAL ON break_c

a_label:
```

```
SAY "Hello"
SIGNAL a_label

break_c:
  SAY "You're not getting out that easily..."
  SIGNAL re_start
```

Try getting out of that one. In fact, there are several ways. If you do try it and then want to exit it, open another shell window and type:

```
HI
```

...at your shell prompt and press return. This sets the global halt flag, which makes all currently running ARexx scripts stop immediately. It's a useful debugging tool to allow you to make those runaway programs come back under control. (You can trap the halt too with SIGNAL halt.) The HI command is one of several useful command utilities, described in detail in the reference section (E). Some are also discussed below when we get on to the subject of debugging your programs.

For a complete list of all the signals you can trap, look up the SIGNAL statement in the reference section (E).

Tracing and debugging

With most of the scripts we have written so far, there is little that can go wrong – apart from the odd typing error perhaps, which is easy to remedy.

However, as you start to use ARexx in real, everyday applications by writing your own scripts you will find that they get increasingly larger. This means that sooner or later errors (bugs) are going to creep in which may not be so easy to find. This is when ARexx's powerful tracing abilities spring into action.

Tracing is a way of letting you see what is going in inside your program and it's an invaluable debugging facility. ARexx will show you each line that it has executed and, depending on the trace mode set, other information about each line as well. If you set a special tracing mode called "Interactive tracing" you are able to "single step" through your program. In this mode, after each line in your program has been run, it pauses and waits for you before continuing. This means that you can go through your program a line at a time, see exactly what is going on – and find the source of problems quite easily.

There are three ways in which you can start tracing. The first is by using the TRACE ARexx statement, followed by the tracing mode you require. The second is to use the built-in function TRACE(), which takes the same tracing mode parameter as the trace instruction, and also returns the previous tracing mode that was in operation. One of the advantages of this is that you can revert to the previous tracing mode if you wish to. The final method is by using the global tracing utilities. These are described in detail in the reference section (E).

Normal tracing information is shown in the console window, normally that of the shell from which you ran your script. If you had run your script from within another program, there may not be such a console window. The global tracing facilities allow you to open a special tracing window and get tracing information sent to that for any ARexx program currently running.

Tracing with the TRACE statement

The TRACE statement takes a parameter to indicate which mode of tracing you would like. There are quite a few to choose from and they range from the ALL option, which traces everything; to BACKGROUND, which means that no tracing takes place and the program cannot be forced into interactive tracing mode using the TS option (see section E). Tracing mode normally defaults to NORMAL, which means that when an error occurs, or a command called returns an error code, the execution of the program stops and tracing information is shown for the line that failed. Let's take a look at an example:

```
/* A Tracing example */

TRACE ALL

DO loop = 1 TO 10
  SAY "loop is "loop
  END
```

When you run this program you will suddenly see a whole load of information rushing past in front of your eyes – this is the tracing data. Let's have a look at a little of this in closer detail:

```
    5 *-* DO loop = 1 TO 10;
    6 *-* SAY "loop is "loop;
loop is 10
    7 *-* END;
```

For every "clause" executed, a line of information is shown. A clause is the smallest single unit of a language that can be executed as a statement. Up until now we've only put one command clause on each line. For example:

```
SAY "I am an instruction clause"
```

However, ARexx does allow us to put multiple clauses on a line by separating them with a semi-colon. Like this:

```
SAY "Hi"; SAY "Hi Again"
```

Incidentally the reason I'm bringing this up now is that, if we ran the above example with two SAYs in it through TRACE, we'd get two trace lines here, one for each clause.

Back to the tracing information itself. We can see that during trace, each executed clause is shown *before* it is actually executed. By changing our TRACE ALL to TRACE SCAN we get an entirely different method of tracing. In this mode the entire program is examined and checked for errors, but nothing actually happens. It's neat way of giving a new program a "test drive" knowing that it cannot cause any damage.

Perhaps the most useful tracing mode is TRACE RESULTS. In this mode all statements are traced, as in TRACE ALL, but the final result of each executed expression is shown as well. This is perhaps the most useful form of tracing for general purpose debugging, because it clearly shows what is going on in your program.

For a really complex display, try TRACE INTERMEDIATES, where almost everything you can imagine is displayed, showing you the intermediate results during expression evaluation. The following table shows a complete list of tracing modes and gives a brief description of their use.

TRACING MODES & CODES

All the tracing options below may be shortened to one letter. For example, TRACE A is the same as TRACE ALL.

ALL	All clauses are traced
BACKGROUND	The program runs with no tracing information, and cannot be forced into interactive tracing using the TS/TCO command utilities (see Section E).

COMMANDS	Command clauses are traced. Command clauses are clauses which are sent to an external host for execution. Non-zero return codes are displayed on the console. (Normally your shell window or the global trace console.)
ERRORS	All commands that generate a non-zero return code are traced after the clause has been executed.
INTERMEDIATES	All clauses are traced and intermediate results are displayed during the evaluation of expressions. This generates a considerable quantity of trace information and includes values of variables.
LABELS	Only label clauses are traced. Labels are specific points in the program to which control can be directly transferred. They are traced after such a transfer takes place.
NORMAL	This is the default tracing mode and nothing is traced unless an error occurs – in which case the failed clause is shown with an error message.
OFF	Tracing is switched off.
RESULTS	All clauses are traced before execution and the final result of every expression is displayed. This is one of the most useful tracing operations because it shows the values assigned to variables from PARSE and other such statements.
SCAN	All clauses are traced and checked for errors, but nothing is executed. This is a program dry run facility – a kind of "test drive".

TRACING CODES

Special three-letter codes are used during tracing to indicate what information is actually being delivered. This is what they all mean:

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

Here are the trace results for the execution of one clause in our tracing test program:

```
    6 *-* SAY "loop is "loop;
        >L> "loop is "
        >V> "2"
        >O> "loop is 2"
        >>> "loop is 2"
loop is 2
```

What does all this mean? Well, the first line shows us the line number of the clause, and the clause itself. We are then shown lots of information preceded by codes such as ">L>". These codes are intended to show you what information is shown after them. >V> for example, means "value of a variable". >>> is "expression or template result".

Interactive tracing

Simple tracing is all very well, but a lot of information goes flying past you on the screen and you are unable to stop it and go through it slowly, a step at a time. This is where interactive tracing comes in. This does allow you to single-step through your program, executing statements if you wish and examining the results. Interactive tracing can be activated to work with all of the tracing modes by simply prefixing it with a question mark. For example:

```
TRACE ?RESULTS
```

This would start interactive tracing using the RESULTS option. In this case, after each clause has been shown, executed and the expression result also shown, the program pauses and gives the prompt "+++" in the console window. Simply pressing return at this point makes the program continue to the next pause point and, in the case of the tracing mode RESULTS, this means the next clause. If we typed an equals character and pressed return, the preceding clause would be executed again. Anything else we type is treated as a special debugging statement and is actually executed. Using our small DO loop example program above, tracing in ?RESULTS mode, we might get something like this:

```
>+>
    7 *-* END;
    5 *-* DO loop = 1 TO 10;
        >>> "2"
    6 *-* SAY "loop is "loop;
        >>> "loop is 2"
loop is 2
>+> SAY loop
2
>+> 6
```

Note that we are able to examine the contents of variables. Interactive tracing in this manner is incredibly powerful, as you will soon discover.

MAKE A NOTE!

While in interactive tracing, all TRACE statements are IGNORED. This means once you start interactive tracing, any further TRACE statements in your program are not executed. Should you need to get around this, you can use the built-in function TRACE() instead which, as well as returning the previous tracing mode, allows you to set a new one, even during interactive tracing.

Another advantage of using the TRACE function is that it can take a string expression as a parameter. This means that you could, for example, ask the user to type their desired trace mode and set it from that variable:

```
/* A Tracing example */

SAY "Type in your desired tracing mode"
PULL trace_mode
CALL TRACE(trace_mode)

DO loop = 1 TO 10
  SAY "loop is "loop
  END
```

Running this and typing ?ALL would trace the program in interactive mode using ALL as a tracing option – which means all clauses will be traced before execution.

Debugging help with Command Utilities

Several special utilities are provided with ARexx to help programmers with debugging. These are found in the RexxC drawer, and are all run directly from the shell. Full reference on them can be found in Section E – Utility Programs Reference. Some of them are particularly useful, particularly the global tracing window.

One of the Murphys Laws of programming is that, when there is a serious bug in your program, it will happen either when you're not looking, or when tracing mode is not on. The global tracing window allows you to open a special console window (from the shell) for tracing, and then with a further utility command, force all active ARexx programs into interactive tracing mode. Very useful!

To open the global tracing window, use the command TCO, and to close it, TCC. You can then use the TS command to force all active programs into tracing mode, or TE to switch tracing off again for all programs.

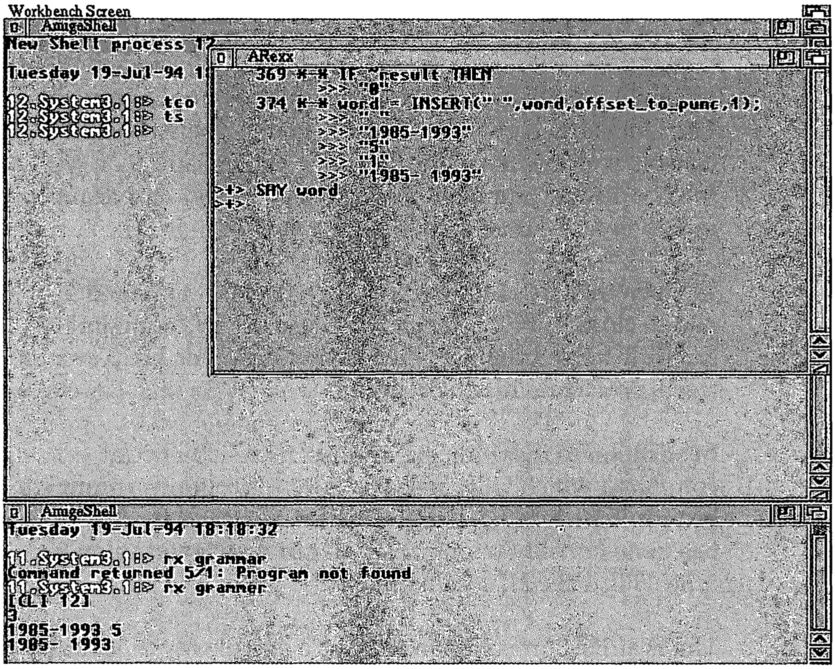
In addition to these tracing utilities, there is a special command which will cause any ARexx programs running to be terminated immediately. This can be useful if something has gone horribly wrong and you just want everything to quit. This is the HI command.

The WaitForPort command can be a useful way of waiting for an external port to arrive. You will learn more about this in section C, but its syntax is quite simple:

```
WaitForPort port_name
```

There is a command to help access the Global Clip List (see below) called RXSET. When called with no parameters, RXSET lists all current entries in the clip list, with their corresponding values.

Finally, in the RexxC drawer is the RX command, which is used to run an ARexx program. But we already know how to use this one!



```
New Shell process 12
Tuesday 19-Jul-94 1

12.System3.1:> tco
12.System3.1:> ts
12.System3.1:>
```

```
   369 *-* IF ~result THEN
          >>> "0"
   374 *-* word = INSERT(" ",word,offset_to_punc,1);
          >>> " "
          >>> "1985-1993"
          >>> "5"
          >>> "1"
          >>> "1985- 1993"
>+> SAY word
>+>
```

```
Tuesday 19-Jul-94 18:18:32

11.System3.1:> rx grammar
Command returned 5/1: Program not found
11.System3.1:> rx grammer
[CLI 12]
3
1985-1993 5
1985- 1993
```

ARexx offers several debugging utilities.

General debugging techniques

Debugging a program can be a very stressful way of spending your time. If you don't approach it in the right way, you can end up shouting, throwing things around, and turning to drink! However, if you are calm and methodical about it, then more often than not it will all be over in no time and will be a relatively painless experience. Like a visit to the dentist!?

When you are debugging a program, think of yourself as a detective. Once you've worked out where the bug can't possibly be, then what's left is where the bug is! Say you have a bug in a huge 10,000 line program. That can be quite daunting, but the bug can't be *everywhere*. Simply discount the areas where you know for a fact that it isn't, and the amount of program code you actually have to look through will become much smaller.

Effective debugging is an art-form in itself but, with ARexx's powerful tracing facilities, and a little calm common sense, you can continue to lead a long healthy life!

Programming style

Programming style? It's best to have some! Keep your programs neat and tidy. Indent your loops so that you can see at a glance where they all are. Comment your code well. You may think something is obvious now, but when you come back to a program to update it several months later, you can find yourself wasting time trying to figure out how it worked in the first place. Make sure you use meaningful variable names – variables like a, b and xyz are no use to anyone. They may be quicker to type but, in the long run, you'll quickly forget what they meant.

If you write neat, tidy, well commented and structured programs, you will find it considerably easier to find bugs, or to update the program, in the future. Don't learn the lesson the hard way like I did. One of my first programs in C was 15,000 lines long and had a skeleton crew of comments. In the end I found it so hard to follow and understand that I re-wrote it from scratch...

Advanced programming

We've covered the basics of programming in ARexx now, but we've left a few loose ends – more advanced subjects which have not neatly fitted in to the "plot". The remainder of this section is concerned with these items and, although none of them are strictly necessary for you to be able to program and use ARexx, you may find some useful background information that's quite handy in the future.

The Interpret statement

INTERPRET is a unique and very special statement. It treats any specified expression as an actual block of ARexx clauses. The expression is evaluated, and the result is executed, as one or more program statements. This may sound a little odd. Let's illustrate with an example:

```
/* Example of Interpret: A simple calculator */

OPTIONS PROMPT ">"
SAY "Calculator. Enter a command:"
PULL statement

INTERPRET SAY statement
```

There are a couple of new things here. The first is using the OPTIONS statement to change the prompt which appears whenever ARexx expects something to be typed from the keyboard. It can be quite useful in instances like this. Then we enter a string into a variable using PULL, and use INTERPRET to evaluate and SAY the result. This program is not just a calculator, it can do anything that SAY can now:

```
8.System3.1:> rx interpret
Calculator. Enter a command:
>TIME('C')
8:39PM
8.System3.1:> rx interpret
Calculator. Enter a command:
>4+3/3*2
6
8.System3.1:>
```

...and if we were to remove the SAY from our interpret line altogether, we could just type in any ARexx statement we wished and have it executed. This could be the basis for an ARexx shell – a small program which allows you to try out ideas by simply typing in anything and viewing the result.

```
/* Simple ARexx shell */

OPTIONS PROMPT ">"
RC = 0

/* Deal with host commands returning non-zero return
code */
error:

SIGNAL ON error

IF RC ~= 0 THEN DO
  SAY "+++ Error: RC =" RC
  RC = 0
  END

/* Deal with syntax errors */
syntax:

SIGNAL ON syntax

IF RC ~= 0 THEN DO
  SAY "+++ Syntax Error:" ERRORTEXT(RC)
  RC = 0
  END

/*
**Loop until "QUIT" is entered, asking for and ¬
then executing
**entered statements
*/
```

```
DO FOREVER
  PARSE PULL shell_line

  /* Deal with the user typing QUIT to leave the ¬
shell */
  IF UPPER(shell_line) == "QUIT" THEN EXIT

  /* Interpret the command now */
  INTERPRET shell_line
  END
```

You will find this a very useful program because you can see the results of simple command tests and functions easily and quickly without having to write an entire program every time. For example:

```
8.System3.1:> rx shell
>SAY "Multiple Clauses"; SAY "Using Semi-Colons!"
Multiple Clauses
Using Semi-Colons!
>SAY TIME('C')
8:50PM
>SAY TRACE()
N
>SAY ABS(-2.2340)
2.2340
>quit
8.System3.1:>
```

I use this program regularly as a simple test bed and calculator. Interpret is a very powerful statement and allows you to actually build and execute programs dynamically.

GOTO or not GOTO

If you've programmed in BASIC before, one command you are likely to be familiar with is the GOTO statement. It's quite simple, it's a direct absolute jump. It allows you to go from where you are now, directly to another part in the program – normally by specifying a line number. In these modern days, we don't need line numbers (indeed, even some modern versions of BASIC have dispensed with them), we get to specify "arrival points" for GOTO style commands ourselves, using names. In ARexx these "arrival points" are called labels, and we have already seen them used in conjunction with the ARexx SIGNAL statement – which is our equivalent of GOTO. Take this example:

```
/* Goto or not goto */
my_label:
  SAY "Forever..."
  SIGNAL my_label
```

This program will continue forever until it is terminated with CTRL-C or by using the global program halt command utility 'HI' from another shell window.

MAKE A NOTE! **With the exception of using SIGNAL to trap error conditions, as we have used it in this section, there is no need at all to use SIGNAL as a direct GOTO.**

Lecture over. GOTOs are generally frowned upon in these modern programming times because they tend to create unstructured, disorganised spaghetti programs which are hard to follow and get you into a mess. Having said this, there are some rare occasions where using SIGNAL in this manner could save you a lot of work. But usually, when you think there is no other solution but to use SIGNAL as a GOTO, there is *always* a way around it using alternative methods – it's just that sometimes the work involved is not worth it in a small program. But don't get into the habit of using SIGNAL in this way!

The NUMERIC statement

This is of particular interest to anyone who uses ARexx to perform complex mathematics, and it allows you to alter the precision, among other things, of ARexx numerical results. For example, we could change the number of digits of precision from the default of 9 to 2 like this:

```
NUMERIC DIGITS 2
```

Then, using our ARexx shell, we could try this result:

```
SAY 1.23+2.34
```

...and get the result 3.6. If we tried 14.2 + 23.34 we could get the result 38. Trying to create a result with more than 2 digits gives us a result in exponential notation, either scientific or engineering, depending on what we have set it to. The default form for expressing exponential notation is scientific. We can change it using NUMERIC FORM SCIENTIFIC or NUMERIC FORM ENGINEERING.

As well as dealing with the precision and the way in which exponential results should be expressed, we can specify the number of digits that will be ignored in a numeric comparison operation. This allows us to make two different numbers appear equal as far as ARexx is concerned. It's a risky thing to do, and is not always entirely accurate – at least not on the current version of ARexx. (1.15)

Operator Priority

We first dealt with arithmetic operators at the very start of this section. A full list of them is detailed in the table on page 17. Mathematicians will recognise the importance of operator priority and how it affects the results of a calculation. We've all seen lines like this at school:

```
(x+y) * (2x+2y)
```

In this case, we specify brackets to show that the two sections inside brackets must be evaluated and multiplied together. If we simply omitted them, we'd end up with this:

```
x+y*2x+2y
```

And if X is 2, and Y is 2, the result would be 20, whereas in the correct version which we intended, above, the result would be 32.

Running the above statement through ARexx gives yet another result – 14. Why is this?

Operators in ARexx have priority values which dictate which will be done first. Since multiply is a higher priority than add, the y*2x will be executed first, giving the result 8, to which the remaining 2y (4) and x (2) are added, giving the incorrect result 14. It is vital to be aware of how operator priority works and, if in doubt, specify by using brackets how each individual part of your expression is to be grouped, or you will get the wrong answer.

The Clip List

The Clip List is a list of global variables which can be accessed by all ARexx programs running on the same system. A clip is a name with an associated value, a bit like a variable with something in it. We can use the built-in functions SETCLIP and GETCLIP to either read or write to these values, and in addition, the SHOW function can be used to list clip values. The command utility RXSET, used from the shell, can be used to change values of clips directly from outside ARexx programs. This can be handy if a lot of your programs share the same global values, or need to pass information between each other in a very simple way. Here is an example:

```
/* Clip Demo */

DO FOREVER
  SAY GETCLIP("my_name")
  END
```

This is an extremely pointless program, but if you run it you'll see a list of empty strings flying past. If you open another shell and either use the command utility RXSET, like this:

```
RXSET my_name "F'Tang"
```

...or run the ARexx shell listed when we talked about INTERPRET earlier in this section and use the SETCLIP() function:

```
CALL SETCLIP("my_name", "Habbish")
```

You'll see that as soon as these commands occur, the result in the window changes, and it starts immediately showing the new result.

Data storage and advanced number formats

With languages such as ARexx we deal with information at a very high-level, like strings and numbers, without ever having to worry about how this information is dealt with inside the computer. This is because, in most cases, we do not need to know and are shielded from this additional complexity. However, there are instances where you might need to deal with information, particularly with strings and characters, on a much lower level, closer to the way which the computer itself understands them.

We know that a string is made up of a collection of characters. ARexx stores these characters in memory one after the other, with a special character with the value of zero to mark the end of a string. This is often referred to as a NULL terminated string. Each character you use is assigned a unique numeric code by which it is recognised. In the 1960's and 70's there were several different codes in use, but the most common now is the ASCII code. ASCII stands for American Standard Code for Information Interchange.

In ASCII, for example, the character "A" has a code of 65, and the character "2" has the code 50. There are 128 defined codes in the normal ASCII set, with a further 128 used to cover special foreign characters, such as the German umlaut "¨", and some mathematical symbols like pi "π"

Codes with values less than 32 are special. There are many, no longer in use, which were designed for the control of teletype terminals back in the days when you didn't have a screen – just an electronic typewriter-style device on which you typed your commands and then viewed the results on paper. All very slow (as bad as 10 characters per second for results from the computer), tedious and noisy.

The code 0 means NULL and, as we have already explained, it is used in ARexx to mark the end of strings (and several other languages, such as 'C'). Here is a brief list of some of the other more common special ASCII codes and what they do. They all apply when shown on an ASCII terminal output, such as the Shell window, for example.

EXAMPLES OF ASCII CODES

8	Back Space.	Moves the cursor back one position.
9	Forward tab	
10	Line Feed.	On the Amiga this moves to the beginning of the next line, and is used to separate lines of ASCII text inside text files.
11	Reverse feed	Goes up to the previous line

12	Form Feed.	New page, on the Amiga this clears the screen.
13	Carriage return.	Moves the cursor to the start of the current line without moving to the next.
27	Escape.	This is a special character and, on the Amiga, it is used to then generate the ANSIπ text sequences, which can change the text colour, for example.

ANSI (American National Standards Institute) sequences are detailed in the Amiga Rom Kernel Manual: Devices, in the section on the console device.

Perhaps the oddest thing about ASCII is that there are 256 total combinations. A strange number? Well, it is and it isn't. To us, 256 is a pretty odd number. So are 32, 64, 128 and 512. But the odd thing is that when we deal with computers they pop up all the time. 512k of memory for example. Why 512?

Computers are digital devices. They only deal in straight ONs and OFFs, with no in-betweens. We refer to an ON as being TRUE, or a Binary 1, and OFF as FALSE, or Binary 0. It's likely you've heard of binary and it's the base two number system. We refer to each individual binary digit as a BIT (Binary digIT)

In everyday life we all work to base 10, or the Decimal system, which gives us 10 digits, from zero to nine. When we reach 10, we add one to the "tens" column and clear the "units" (think back to when you were at primary school and doing this in the classroom!). For larger numbers we might have a thousands, hundreds, tens and units column, because our column headings are in powers of 10.

In Binary arithmetic we only have two digits, 0 and 1, so our column headings are in powers of 2. So instead of units, tens, hundreds, we have units, twos, fours, eights and so forth. The number 10101 in Binary therefore looks like this:

16s	**8s**	**4s**	**2s**	**1s**
1	**0**	**1**	**0**	**1**

You can see we have one in the 16, 4 and 1 column. Add these together and we get the result 21. So, the Binary for 21 is 10101. You can easily convert from Decimal to Binary and back again this way by writing down the column headings and then filling in the ones and zeros accordingly. 13 in Binary, for example, would have ones in the 8, 4 and 1 columns (1101).

So, lets pop back to the number 256. There are 256 valid ASCII character codes, running from 0 to 255. And the binary for 255 is:11111111!

A-ha! A nice round computer number. Eight BITS together. A group of eight bits like this is called a BYTE, and 1,024 of them is a kilobyte, or a k. 1,024k is a Megabyte, or an Mb.

Your computer might have several Mb of memory. The Amiga A1200, for example, as 2Mb of RAM, or 2,048k, or 2,097,152 individual bytes of storage space available. Since each ASCII code is one byte, that follows that you can store a document 2,097,152 characters in length or, assuming an average word length of 4.4 characters, 487 thousand words.

In reality your 2Mb of memory gets eaten up in huge chunks by the Amiga itself. If you do own an A1200 with only 2Mb of memory and you look at the workbench title bar, you will see a display saying how much "graphics mem" you have – and this number will be a whole lot less that 2097152!

Obviously Binary is a very people-unfriendly number system because, with even relatively small numbers such as 60,000, we end up with Binary numbers 16 digits long. So, when we are dealing with computers, instead of talking in Binary, people use Hexadecimal.

Hexadecimal is base 16, which means that it has 16 digits, which run from 0 to 9 and then A to F, where F equals 15. The convenient thing about hexadecimal (normally called Hex), is that when you have a long binary number, each four bits in it can be shown as a single Hex digit, as the maximum number you can store in four bits is 1111, which is 15 and the value for F in Hex. So, ASCII codes consist of eight bits, or any value from 00 to FF in Hex. You may well have seen Hex used but not known what it meant or what it was supposed to do.

Open a shell and type the following:

```
type s:startup-sequence opt h
```

... and you'll get a result on the screen like this:

```
0000:
  3B202456 45523A20 53746172 7475702D
    ; $VER: Startup-
0010:
  53657175 656E6365 5F4D6170 524F4D20
    Sequence_MapROM
```

This is a "Hex Dump" of the file "s:startup-sequence", which you should recognise as the file which is executed when your computer starts up. Down the right hand column we have the ASCII characters themselves in text form, and in the centre are their ASCII codes, in Hexadecimal. At the far left is the offset into the file from the first character, again in Hex. Notice that 16 characters are shown on each line, so the start of the second line will be 16 from the beginning, and 16 in Hex is 10.

If you haven't already managed to confuse Decimal and Hexadecimal, you soon will – especially Hex numbers that don't have A-Fs in them. To get round this we distinguish between the two by adding a special identifier to every Hex number to set it apart. Unfortunately each different computing language seems to do this in a different way. Assembly language programmers are used to adding the $ (dollar) prefix before a Hex number; for example, $123A0. In 'C' the 0x prefix is added, as in 0x123A0. In ARexx, however, we refer to Hex numbers using the x postfix character. For example:

```
SAY "41"x
```

... produces the result:

```
A
```

on the screen. This is because 41 in Hex is the ASCII code for the letter A.

In the same way we can define Binary strings with the b postfix, and if we did this:

```
SAY "100001"b
```

We would also get an A on the screen because 1000001 in Binary is $41 in Hex, or 65 in Decimal – and all are the ASCII code for A.

The most useful way of using Binary and Hex strings in general, day-to-day ARexx programming is to insert special

ASCII characters into strings, such as the linefeeds for example:

```
/* Hexadecimal characters into strings */
  SAY "A string"'0a'x"with a new line in the middle!"
```

This produces the result:

```
A string
with a new line in the middle!
```

The ARexx built-in function library contains loads of functions especially for the conversion of Hex and Binary to Decimal and vice versa (see Reference Section E for further information and examples of using these functions). This may all seem a bit irrelevant right now but the first time you find you need to deal with calls to the Amiga operating system, or do any remotely advanced string handling, you'll find the information above very useful.

A knowledge of how ASCII works and how numbers are stored can be a life saver when it comes to certain routines. For example, did you know that to convert lower case characters to upper case ones, you simply subtract 32 (Hex $20) from their ASCII code values? Armed with this knowledge, and a few of the string-handling functions ARexx offers, you could write a script which neatly formatted a sentence and corrected the spacing if the original didn't have spaces after full stops. Appendix 3 at the end of the book shows the entire ASCII character set.

Another look at Boolean algebra

Although, strictly speaking, it's not necessary to teach Boolean algebra in a book like , you'll have a much better understanding of how ARexx programming works if you have a little background information. Besides, the fog which will cloud your appreciation of a large number of functions such as BITXOR() will mysteriously clear and you might even find yourself using them!

Computers are simple beasts, very simple beasts indeed. They are made up of hundreds of thousands of little electronic switches, and these switches can either be on, or off. Instead of a mechanical switch, where human intervention is required to change it from on to off, or vice versa, with these electronic switches the on-or-off state is controlled with a third input. Those of you who have tinkered with electronics in the past might recognise these as transistors.

Certain combinations of these switches can be used to generate "black box" functions, where by what you shove in one end dictates what you get coming out of the other. For most Boolean algebra you really only need to understand a few of these "black boxes" AND, OR, and NOT (which we're going to call "logic gates"). And it's all amazingly simple.

Lets deal with the simplest first – the NOT logic gate. This has one input and one output. We have already seen the ~ NOT operator in ARexx. It simply inverts the state so that, for instance, if you put a binary 1 (TRUE) in, you get a 0 (FALSE) out, and vice versa.

How about an AND gate? Well, AND is slightly more complex, it has two inputs, and one output. If we label these inputs A and B, and the output Q, we can draw the following truth table:

"AND" TRUTH TABLE

A	B	Q
0	0	0
0	1	0
1	0	0
1	1	1

If you followed the above using Binary, you'll realise that with two inputs, there are four possible combinations we can put in, 0 0, 0 1, 1 0 and 1 1. With an AND gate, if we put 1 and 1 in, we get a 1 out, but any other combination gives us 0 (FALSE). We've used the ARexx & (AND) symbol before, in IF statements. Now, we can see exactly how works:

```
IF age>10 & age<15 THEN SAY "It gets worse now"
```

We know from our first discussion of Boolean algebra that ARexx will evaluate the two sub-expressions (age > 10 / age < 15) and come up with a TRUE/FALSE. If both are TRUE, then the SAY gets executed. Let's assume TRUEs are 1s. Now run that through the above truth table and you should see that the & operator in ARexx actually performs an AND gate operation to get the result, because it's only when both parts of the expression are 1 that the entire expression evaluates to 1.

OR, and the ARexx OR symbol " | " works in much the same way. Here is the truth table for an OR gate:

"OR" TRUTH TABLE

A	B	Q
0	0	0
0	1	1
1	0	1
1	1	1

The way it works is that if either of the two inputs are TRUE (1) then the output is 1. Again, let's use an IF statement as an example:

```
IF age < 10 | age > 80 THEN SAY "You're either very
young, or old."
```

The two sub-parts of the expression are evaluated and fed into an OR logical operation. If either one of them were TRUE, then the output is TRUE.

It may be difficult for you to see why you might need these, but if you get into programming ARexx seriously then you probably will have to use functions which work with logical operations like this.

The moral of this sub-section is that, if we'd been born with 16 instead of 10 fingers, using computers would be a whole load easier than it is today... However, having said that, many veteran machine code programmers can do everyday mental arithmetic in Hexadecimal just as easily as you can in Decimal!

Section C
Controlling External Applications

We've seen ARexx used as a language in its own right but, as you'll recall, there is one vital feature which allows it to talk to other applications and control them. However, you may still be wondering why this is so useful.

Imagine you had 300 pictures that had just been generated using a ray tracing package, such as Real 3D, and which formed an impressive animation of a spaceship flying past a planet. These frames may well have taken you days to produce. If the pictures were all 640 x 512, in HAM mode, and you wanted them in 320 x 200, and 32 colours as well, how would you go about converting them all? Well, even if the ray tracing package had the facility to output in 320x200, 32 colours and you could re-generate the frames, this would take a long time.

Alternatively, you could load each picture, one at a time, into a package such as Art Department Pro, and convert them all that way. A very boring, repetitive task.

Then again, you could write a very short ARexx script which loaded each picture into Art Department, converted it, and saved it for you, while you went and put the kettle on and watched TV.

If you regularly use a simple text editor for writing small documents and so on, you might be pining for some of the full wordprocessor features, such as a spell-checker and thesaurus. But if your text editor had an ARexx port, and you also had a word-processor that had these features, you could write a small ARexx script which, at the press of a button, did the work for you. In the same way, you could make a wordprocessor have text-editor style operations, or simply write an ARexx program which, when used from the shell, spell-checked a named file, or looked up a word in the thesaurus for you.

It's this ability to use other applications' features in a controlled way that makes ARexx so powerful. It can be used for a vast range of things – with a paint package and ARexx, you could draw graphs of results of mathematical equations, for example (useful in education applications) – and the ARexx scripts you would need to write are all very straightforward and use techniques that we have already covered in Section B.

If this is still not enough to convince you of the power ARexx gives you to squeeze every ounce of usefulness out of your Amiga applications, then look at the section on controlling specific applications in this book. This gives real, useful examples of a number of diverse programs, and even shows how to use two applications together, with ARexx, to create whole new features.

ARexx "ports" and degrees of support

If you intend to make regular use of ARexx, then it makes sense to buy software which supports it, and which provides you with a wide range of control. Most major commercial applications in the fields of desktop publishing, wordprocessing, text editors, utilities, and art packages for example, now come with an ARexx "port".

Programs which have an ARexx port are able to act as "host applications". This means the application provides a range of commands to allow ARexx to communicate with it. Depending on the level of ARexx support, and the type of application, this might include commands to control the application directly – or get information which can be used by the ARexx script to make decisions.

As you come to know ARexx a little better, and read the section on controlling specific software (D), you'll learn which kind of ARexx commands will be useful to you. It's then much easier to decide which programs to buy, because with good ARexx support you can expand the program yourself as you go along.

ARexx-supporting programs differ greatly in the type of commands provided. They could provide ARexx control of the user interface, allowing an ARexx script to bring up requesters, open windows and access the options present on the menu bar.

You might also be able to control the core features of the program. In the case of an ARexx-supporting paint package, this might allow you to draw lines, circles and squares and to colour areas of the screen. In a wordprocessor you might be able to scan a document for specific words, and perhaps generate an index at the end.

You might also be able to get information out of specific features of packages using ARexx. Wordprocessors might allow you to write a script which passed words in, had them spell-checked, and returned a list of closest matches if the word was incorrectly spelt – very handy for spell-checking files directly from your shell window without having to do the work yourself. A paint package might let you cut brushes out and save them. You could then load these brushes directly into another program and perform some other operation with them, all with the help of ARexx.

And by using a Mandlebrot fractal generator program with ARexx, it is possible to write simple ARexx scripts which will generate animations as if you were travelling around the fractals, all while you were free to do something else.

ARexx support today

In the early days it was quite hard to find major commercial programs which supported ARexx – now it is equally difficult to find one that does not. A brief (and certainly in no way exhaustive) list of some of the applications supporting ARexx can be found at the end of this section.

Software publishers have realised how useful ARexx can be for themselves (to add missing features and flexibility by writing some simple scripts) and for the user (to enable them to perform repetitive operations automatically, and to expand the power and uses of their investment).

Increasingly, public domain and shareware titles also come with good ARexx support. Communications programs allow you to write scripts which log on to bulletin boards by themselves, wait for all the routine prompts and then respond with the appropriate answers – all without you having to press a key.

TOP TIP **A good source of shareware and public domain (PD) utilities in general, and a wide range which support ARexx, can be found on the Fred Fish PD disks. Any good magazine will have plenty of adverts for such disks, and they normally cost around £2-3 each. Users with CD-ROM drives can buy CDs with over 500 disks-worth of PD and shareware software. Bulletin boards are also an excellent source of ARexx PD software.**

Finding ARexx scripts

Other than the scripts which you write yourself, and any that may have come included with programs, there are many scripts which are available in the public domain. You can find these on disks from PD libraries, and also from bulletin boards around the world. It can be a good way to learn more about the language by seeing what other people have done with it to solve their problems.

As we'll see later in this section some applications also provide you with a recording facility which allows you to perform actions within a program, while recording what you're doing as an ARexx script which you can "play back" at a later date.

Communicating with applications

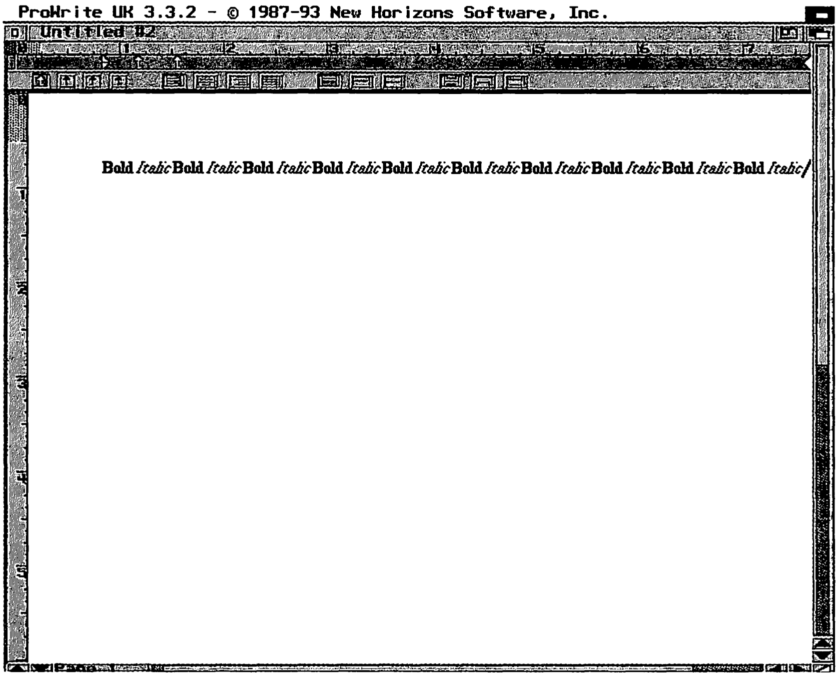
ARexx can only talk to one ARexx port at a time. Every ARexx port has a name, which is called its "host address". To talk to an application, you tell ARexx the host address, using the "address" command:

```
address "ProWrite"
```

This sets the CURRENT HOST to the ProWrite wordprocessor. ARexx also remembers the previous host address, and you can return to this quickly and simply by using "address" with no host name.

Assuming the address existed, which means that ProWrite would first have to be loaded, we can now access it:

```
/* ProWrite Demonstration! */
ADDRESS "ProWrite"

ScreenToFront

DO loop = 1 TO 10
  StyleBold
  Type "Bold "
  StylePlain
  StyleItalic
  Type "Italic "
  StylePlain
  END
```



Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic* Bold *Italic*

ProWrite supports ARexx and we can write a little ARexx script to type the words bold and italic ten times in the appropriate style in a ProWrite document.

The results of this little script can be seen in the screen grab shown above.

So, what did we do? First, we set the name of the ARexx host, using the address command. Since we are now talking directly to this host, any keyword we use which ARexx does not understand will be passed on to ProWrite, to see if it is a valid ProWrite ARexx command. ScreenToFront simply brings ProWrite's screen to the front so that you can see the action occur.

We then used a simple DO loop to show the words "bold" and "italic" 10 times, in the appropriate style. The ProWrite

ARexx command "Type" acts as though the text that follows it had been typed into the wordprocessor directly from the keyboard.

It's certainly not the most useful ARexx script since the dawn of mankind, but it is our first example of communicating with an ARexx port – and we can use the same type of loop to do far more useful things, such as print out numeric tables from spreadsheets and so on.

What would happen if the host name did not exist? If we change the address line to read

```
ADDRESS "ProJelly"
```

instead, and then run the script, ARexx would stop when it could not execute "ScreenToFront":

```
7.System3.1:> rx write
+++ Error 13 in line 5: Host environment not found
Command returned 10/13: Host environment not found
7.System3.1:>
```

An unfriendly error like this is all well and good, but it would be much more professional if we could detect whether the host name required actually existed and, if not, load the appropriate application.

Although there is not an ARexx keyword to do this, there is an in-built function which does the job called Show():

```
/* ProWrite Demonstration! */

IF ~Show(Ports,"ProWrite") THEN DO
  SAY "Prowrite is not running!"
  EXIT 0
  END
```

```
ADDRESS "ProWrite"
SAY "ProWrite is running"
```

In this example, if the ProWrite host address is not available, the script exits with a message to say why it failed, which is far neater. Show(), used in this way, returns TRUE if the named port existed, or FALSE if it did not. So, by using the NOT symbol "~", we are able to say "If the host address 'ProWrite' is NOT available then do...". Of course, to put the icing on the cake, we could make our ARexx script load up ProWrite if it was not already loaded:

```
/* ProWrite Demonstration! */
ADDRESS command

/* Run ProWrite if it is not already running */
IF ~Show(Ports,"ProWrite") THEN "run ¬
programs:prowrite/prowrite"

SAY "Finding Prowrite.."

/* Wait until port becomes available */
DO UNTIL Show(Ports,"ProWrite")
  END

SAY "ProWrite is running"
ADDRESS "ProWrite"
```

When run, the script produces the following message:

```
7.System3.1> rx write.rexx
[CLI 8]
Finding Prowrite..
ProWrite is running
7.System3.1>
```

By now your ARexx knowledge should tell you how the above script works. Perhaps the most interesting line is the first "ADDRESS command". In this case "command" is the host address of the Shell, allowing you to run AmigaDOS commands. In this example if the ProWrite port is not currently available, we run the program. Then, using a DO UNTIL loop, we wait until the port appears before continuing. We could improve this script still further, particularly in the DO UNTIL loop. If the loading of ProWrite failed, the loop would continue forever, or until the user used CTRL-C to stop it. A better method might be to check the port every second for 10 seconds, and then give up.

This method of waiting for a port to become available will work with any disk-loadable application which supports ARexx. Simply change the host address. NComm, for example, is a shareware communications package. You could write a script which automatically dials and connects to your favourite bulletin board, in this case the popular CIX service:

```
/* NComm dial example */

ADDRESS command

/* Run NComm if it is not already running */
IF ~Show(Ports,"ncomm") THEN "run ncomm:ncomm"

SAY "Finding NComm.."

/* Wait until port becomes available */
DO UNTIL Show(Ports,"ncomm")
  END

ADDRESS "ncomm"

/* Dial the number: */
```

```
SAY "Dialing BBS.."
dial "cix"
wait "login:"
send "qix\n"
```

We're using three of NComm's ARexx commands here – one to dial a number that is in the phone book, the next to wait for the string "login:", and finally to send the correct response together with a newline (\n) to enter it.

One of the problems with the scripts we have shown so far is that they are only giving instructions to the host application and are not receiving any responses.

We would have no way of knowing, in the second example, if the dial command had failed for any reason (say the phone number at the other end was engaged), and our script would quietly wait forever for the "login:" string to arrive from our modem, even though it would never actually appear.

To really make the most of a host interface, you have to be able to make decisions based on what is happening. In the previous section we looked at functions as a way of finding things out and acting on the results.

Information from the host

If ARexx host commands are simply treated as functions, some of which just perform direct actions with no result, and some which return information of some sort, then it's easier to understand them. There are two basic ways in which a host can return information to your ARexx script.

RC: Return Code

The first is by using a special variable called RC, meaning "Return Code". Normally, a host command would return 0 (i.e. a low number), which would indicate "Everything was OK." Returning any other value normally means that an error has occurred – the higher the number, the more severe the error. ARexx can be told at what stage to treat a returned error as fatal, and it will then automatically halt execution of a script. This defaults to 10, although you are able to set it yourself using the "OPTIONS FAILAT" command, for example:

```
OPTIONS FAILAT 20
```

Now only host commands returning a value in RC greater than 20 will cause your program to stop. This can be quite useful if you wish to deal with errors yourself, respond with sensible messages, and perhaps give the user a way of continuing. In this example a script attempts to load an IFF picture into Art Department Professional. We're assuming the host will be available:

```
/* Art Department Pro, Picture loading script */
ADDRESS     "ADPro"
'ADPRO_TO_FRONT'
'LFORMAT'   IFF
'LOAD'      '"my_pic"'
IF RC >0 THEN SAY "Could not load picture!"
```

This script initially brings ADPro's screen to the front so that we can see it, and then sets the load-format to IFF. Finally, using the LOAD command, it attempts to load the picture "my_pic". If the picture does not exist, LOAD returns 10 in RC. We're then testing it with an IF statement to show an error message if the LOAD command failed.

There are two things of interest about this particular script. The first is that we've put ADPro's commands in single quotes. This is to force ARexx to treat them as external host commands rather than trying to recognise them as internal functions, variables and keywords. Because both single and double quotes are interchangeable (as discussed in Section B), we could achieve this by using double quotes – but the convention in this book is to use single quotes when forcing commands to be sent directly to a host without further processing by the interpreter. We'll talk more about this later, but to illustrate why it's important, if we'd just used:

```
LOAD '"my_pic"'
```

... instead of surrounding LOAD with quotes, then ARexx would have first tried to see if it was an internal keyword – in this case it was not. Then it would search the variable list, and see if there was one called "LOAD" and, if so, substitute the variable with its contents. If we'd had a variable called LOAD this would have caused the line to fail, and ARexx would have used the contents of the variable as a command. (There are occasions where this is useful, see "Variable Substitution" later on.)

But it gets worse. If it could not recognise it as a variable, then it would convert it to upper case and then pass it to the current host. This may or may not make a difference, depending on whether the host is case-sensitive. NComm is not, for example, but other applications might be.

The moral of the story is to put quotes around any command that is intended to been passed directly to a host application without translation. It's also faster, because ARexx does not try and recognise it as a keyword or variable before sending it to the host.

The second thing about this script is that it does not actually do what we want at all! When running it, if the picture is not recognised, you'd expect the IF line to print the string "Could not load picture!" on the screen. It doesn't. Instead, ARexx halts execution with this message:

```
    5 *-* 'LOAD' '"my_pic"'
+++ Command returned 10
```

Because the default value for FAILAT is 10, ARexx stops running your script immediately. If we wanted our IF line to work, we'd have to raise the FAILAT value to something a little higher, by adding a line something like this to the start of the program:

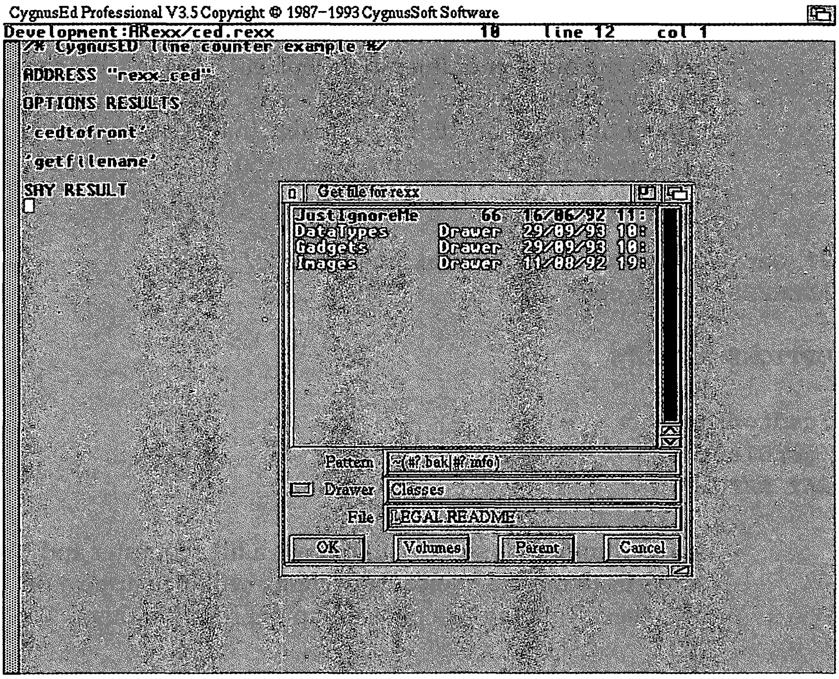
```
OPTIONS FAILAT 20
```

You can also process host command failures using the SIGNAL command, and interrupts. See the previous section for a more thorough discussion of interrupts.

Using the OPTIONS RESULTS line

More often than not host commands that perform actions will want to return something more substantial than a simple error code indicating success or failure. This could be a numeric or a string value. ARexx provides a standard mechanism for this, using a special variable called "RESULT". When you call a host command, it is able to place values in the ARexx RESULT variable, and you can then process it as required. The only catch is that you must remember to tell ARexx that you are expecting host commands to return values in RESULT using the OPTIONS RESULTS line. This should be placed very near to the start of your program:

```
/* CygnusED line counter example */
```



```
CygnusEd Professional V3.5 Copyright © 1987–1993 CygnusSoft Software
Development:ARexx/ced.rexx                    18     line 12     col 1
/* CygnusED line counter example */
ADDRESS "rexx_ced"
OPTIONS RESULTS
'cedtofront'
'getfilename'
SAY RESULT
```

```
Get file for rexx
JustIgnoreMe      66    16/06/92 11:
DataTypes      Drawer   29/09/93 18:
Gadgets        Drawer   29/09/93 18:
Images         Drawer   11/08/92 19:
Pattern ~(#?.bak|#?.info)
Drawer  Classes
File    LEGAL.README
```

ARexx can be used to call up a filename requestor in CygnusED and return a text string (the chosen filename).

```
ADDRESS "rexx_ced"

OPTIONS RESULTS

'cedtofront'
'status 17'

'okay1' "Lines in file:" RESULT
```

This example shows the number of lines in the currently selected text window on ASDG's CygnusEd Professional text editor. Having set the host address and told ARexx to use the RESULT variable, we bring the CygnusED screen to the front, and call the status command. Status 17 means "return the number of lines in the current text file". We then pop up a little requester on CED's screen showing the total number of lines. The RESULT variable in this case returned an integer value. Again, using CygnusEd, we could get a filename choice, and the result would be a string:

```
/* Get a filename choice from the user */
ADDRESS "rexx_ced"

OPTIONS RESULTS

'cedtofront'
'getfilename'
SAY RESULT
```

When running this script, CED shows a file requester and allows us to make a choice. That choice will then appear on the screen. Running this program might result in this:

```
7.System3.1:> rx ced
Devs/DataTypes/Windows Bitmap
7.System3.1:>
```

Take a look at the grab on the previous page to see the script running.

Quite a few ARexx supporting programs allow you to make use of user interface functions, such as file requesters and information requesters, in this way. It means that you can make your ARexx scripts fit in and look neat. CygnusEd gives you functions to get all sorts of information from the user in this fashion, and also display data-in requesters.

Host addresses & commands

In this section we've talked to several applications which support ARexx, and each has a very different range of commands – and there are certainly things to be wary of.

WARNING

Although it has been mentioned before, do watch out for command case-sensitivity. Check the documentation with a software package and follow the instructions carefully. It's definitely recommended that you surround any commands intended to go to a host application with single quotes, since this avoids ARexx misunderstanding what you wanted to do and, instead, treating your command as an ARexx Keyword, or a variable.

Variable substitution

As we've already established, if ARexx can't identify a command as an internal keyword it tries variable substitution before passing it to the current host. Let's look at this example:

```
/* Substitution Demo */
ADDRESS command

SAY "Enter a command:"
PARSE PULL BLOOP

BLOOP
```

We're using the "command" host address, which allows us to use AmigaDOS commands. If you run this program, and type "dir", then you will get a directory. When ARexx gets to the line BLOOP, it realises that it is not an ARexx keyword, so it looks at the variable list. Since BLOOP contains a valid value, this is converted first to upper case, and then passed to the host – in this case, the Shell. So if we enter "list", we'll get a detailed directory listing. If we had typed an invalid command, ARexx would still send it to the host, but the host would reply to ARexx saying "No, that isn't a valid command here.", and ARexx would stop – a little like this:

```
8.System3.1:> rx variable
Enter a command:
list downloads: since Wednesday
Directory "downloads:" on Wednesday 24-Nov-93
ar134.lha                    81368 —-rwed Monday
18:28:17
scratchpad.old               69351 —-rwed Today
13:38:27
SCRATCHPAD                   11177 —-rwed Today
14:23:04
wack.lha                     73728 --arwed Wednesday
16:22:37
4 files - 466 blocks used
8.System3.1:> rx variable
Enter a command:
feedle-de-de!
feedle-de-de!: Unknown command
feedle-de-de! failed returncode 10
  8 *-* BLOOP;
+++ Command returned 10
8.System3.1:>
```

We could also type ARexx commands in and run them too:

```
/* Substitution Demo */
SAY "Enter a command:"
PARSE PULL BLOOP

BLOOP
```

Here's a likely result of running this script:

```
8.System3.1:> rx variable
Enter a command:
'SAY Hello "Hello"'
HELLO Hello
8.System3.1:>
```

Host addresses

We've used the ADDRESS keyword throughout this section to demonstrate communications with other hosts. Host addresses (ARexx port names) are case-sensitive. This means that "ADPro" and "adpro" are two different port names so you will have to get it right before you can make use of that host. Check with the application's documentation to be sure.

Since the Amiga has a multi-tasking environment, there are times when you might have several copies of a program running at once. But if the program supports ARexx, doesn't this mean you would get several copies of the same ARexx port? And if this was the case, how would you specifically talk to one in particular?

Well, programs which can be run multiple times usually append a number to the end of additional ports which are created. If you were to run CygnusEd Professional twice, the first one would have a host address of "rexx_ced" and the second "rexx_ced1".

Applications can handle circumstances like this in various ways, but the most common method is to add ".X" to the end of a host address, where X is a number which increments depending on how many copies of this program are running.

In addition to ARexx's ADDRESS keyword, there is also a special function available in the built-in function library, also called ADDRESS. This returns the name of the current host address.

```
/* Address () Demo */

ADDRESS command
SAY ADDRESS()

ADDRESS
SAY ADDRESS()
```

When we run this program we are likely to get a result like this:

```
7.System3.1:> rx addr
COMMAND
REXX
7.System3.1:>
```

This also demonstrates the way that the ARexx keyword ADDRESS, on its own, reverts to the previously set host address, in this case ARexx itself. Incidentally, if you ever want to clear the current host, so that you are no longer sending unrecognised variables/keywords, then you can use ADDRESS "REXX".

Using ADDRESS "REXX"

REXX is the name of the default host. If you don't specify an ADDRESS at all in your program before running it from the Shell, then if ARexx comes across something it would normally send to the host address, it tries to execute it as an ARexx script instead. This means that you can run scripts from within scripts:

```
/* Save this as dino1.rexx */
/* This address statement below isn't strictly
necessary as REXX is the default, but it's there for
```

```
completeness in this example */
ADDRESS "REXX"
DINOSAUR
SAY "Ok, it's finished that now."
```

And the second script:

```
/* And this as "dinosaur.rexx" */
SAY "STEGOSAURUS IS THE BEST DINOSAUR EVER"
```

If you now run the first script, dino1, you will get:

```
8.System3.1:> rx dino1
STEGOSAURUS IS THE BEST DINOSAUR EVER
Ok, it's finished that now.
8.System3.1:>
```

Notice that the first script waited patiently for the second one to complete before continuing.

More on ADDRESS()

You may wonder what use ADDRESS() actually is because, to set the address in the first place, you must know what it is – so why would you need to find it in this way?

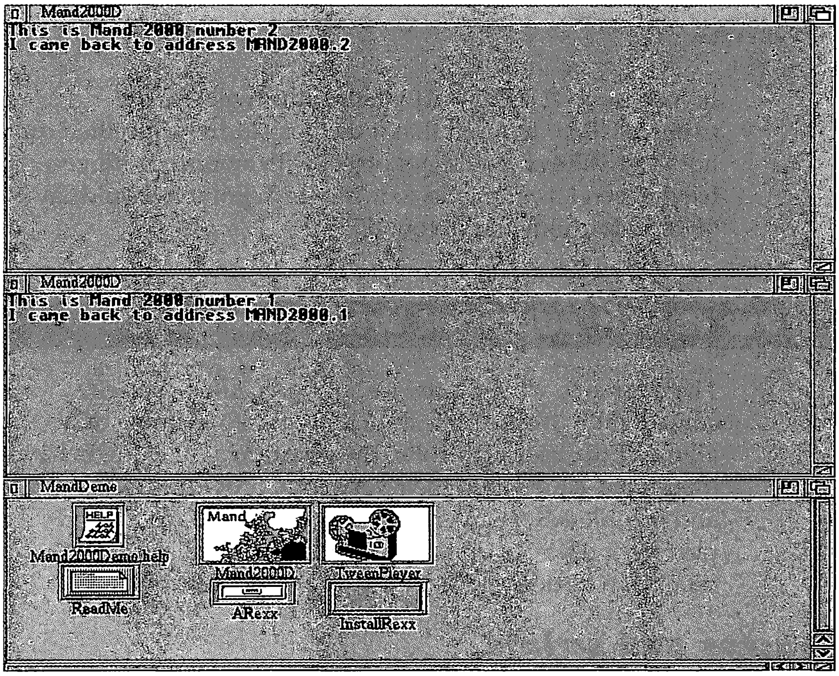
As you get to know ARexx in more detail you'll begin to see how useful it can be. If your script is executed from within an application the address is often set for you to that of the program. This is quite important because, for the reasons set out above, the address may not necessarily be the same each time – it could be "APP.1" or "APP.2" depending on the number of copies of the host application running at the time. So you may wish to add some code which verified that the address was correctly set or, bearing in mind that ADDRESS only remembers one previously set address, if you're going to communicate with several hosts during a script you may need to remember which host you were at:

```
/* Address () Demo */

/* Confirm we're using Mand2000 */
mand_addr = ADDRESS()
IF LEFT(mand_addr, 8) ~= "MAND2000" THEN DO
  SAY mand_addr "is not a valid MAND 2000 address!"
  EXIT 0
  END

/* Now find out which Mand2000 we're using */
IF LENGTH(mand_addr) = 10 THEN DO
  mand_number = RIGHT(mand_addr, 1)
  SAY "This is Mand 2000 number" mand_number
  END

/* Now do lots of address things */
ADDRESS command
ADDRESS "ProWrite"
ADDRESS "REXX"

/* Now back to Mand 2000 */
ADDRESS VALUE mand_addr
SAY "I came back to address" ADDRESS()
```

This is an ARexx script which was run from a user-defined menu in CygnusSoft's excellent Mandlebrot program. It uses several functions to verify that we're using the correct address, and then extracts the number from the end of the address to work out which we are using.

The grab opposite shows the results of this script when run from two different copies of Mand2000 both loaded at the same time.

Notice at the end of the program we're using ADDRESS VALUE, rather than address. This is the format of



```
This is Mand 2000 number 2
I came back to address MAND2000.2
```

```
This is Mand 2000 number 1
I came back to address MAND2000.1
```

Address() can be used to keep track of your host applications when things get complicated and you're running several at once.

ADDRESS you need if you intend to use variables to store host addresses. Look at this example:

```
/* Address test */
my_address_name = "ProWrite"
ADDRESS my_address_name
SAY ADDRESS()
```

You might normally expect the host address to be set to "ProWrite", but ADDRESS takes "my_address_name" literally, and the output of this script is:

```
8.System3.1:> rx pro
MY_ADDRESS_NAME
```

```
8.System3.1:>
```

If you use ADDRESS VALUE then this will work as expected. Bear in mind that if the host address specified did not exist, ARexx would not object with an error until you tried to send a command to it, at which point it would say "Host environment not found".

How much support will you get?

The degree to which software supports ARexx varies considerably from application to application. Some only allow a little basic control of the program, while others have integrated ARexx into their very core, providing a powerful and flexible ARexx interface with facilities to record and play back ARexx scripts and recall them off disk at the touch of a button.

Macros and scripts

Unfortunately, you'll find that both these terms can be used to describe a number of things in the computing world, and different programs use them to describe different things. In this book we refer to our ARexx programs as "scripts". A lot of programs (particularly in the IBM-PC and telecommunications world) use the term "scripts" to describe special programs written in custom script languages, sometimes not unlike a simple version of ARexx. But remember that these scripts are unlikely to be ARexx-compatible so a program that claims it supports "execution of scripts" does not necessarily support ARexx.

A number of programs have the facility to record sequences of actions carried out by the user and then play them back at the press of a button. This feature is mostly found in text editors and is there to allow you to program in simple sequences of key presses and repeat them quickly throughout a document – although this facility is rapidly becoming popular in other applications too.

These recorded sequences are often referred to as macros. Programs which support macros in this fashion usually integrate them with ARexx support so that sequences are actually recorded as ARexx scripts. This allows you to look at and edit the results easily. It is a handy way of generating quite complex scripts very rapidly. The really useful thing about recorded sequences such as these is that you can effectively "teach" a program how to do something, by going through it once (and then maybe making some minor changes to the ARexx script yourself) so that it can repeat the job next time.

This allows you to perform repetitive tasks within applications very efficiently indeed. You could teach an art package, for example, how to load a picture, make changes to the palette, add a picture frame around the outside, and then save it out. Now you could encase the resulting script in a DO loop = 1 TO Number_Of_Frames loop and convert an entire animation in one go – yet all you've had to do is write a couple of lines in ARexx.

As well as macro recording, ARexx-supporting programs frequently have a menu set aside for execution of ARexx scripts. ProWrite, for example, has a macro menu which enables you to specify 10 ARexx scripts which can be run directly from the menu or by pressing a key. This way you can write a number of useful wordprocessor-related scripts and integrate them into the program – making them available at the touch of a button.

CygnusED and Art Department Professional can also run ARexx scripts this way, but ADPro does not have a special menu and uses function keys to activate scripts instead.

Programs that support ARexx

To give you some idea of what kind of programs have ARexx ports, here are some of the most common application types on the Amiga, and a few which support ARexx. Some of these are dealt with in much greater detail in the next

section, Controlling Specific Applications. Please remember that this list is in no way complete or comprehensive.

TOP TIP **A list which is maintained voluntarily by Daniel Barrett in America currently contains a list of 340 applications, shareware, public domain and commercial packages, all of which support ARexx. Users with modems might like to try and get a copy of this list, which is available for FTP from AmiNet sites, and is called "ArexxAppList.lzh".**

New applications with ARexx support appear every day, so check the ads in the latest Amiga magazines. Unless indicated otherwise, all software listed below is commercial and you'll have to check for the latest prices and availability. Where possible a contact number is given for the company supplying the package. The numbers include all necessary dialing codes to call from within the UK.

Text Editors and Wordprocessors

Title:	**ProWrite**
Description:	**Powerful wordprocessor**
Author:	**New Horizons Software Inc., USA.**
Contact:	**0101 512 328-6650**

Title:	**Final Copy**
Description:	**Wordprocessor**
Author:	**Softwood, Inc., USA**

Title:	**CygnusEd Professional 3**
Description:	**Text editor. Ideal for software development. *Very* fast!**
Author:	**ASDG Inc., USA**
Contact:	**0101 608 273-6585**

Utilities, Data & File Management

Title:	**Directory Opus 4**
Description:	**Advanced File Management (Copying, Renaming and so forth)**
Author:	**InovaTronics Inc.**
Contact:	**0707 660992**

Title:	**AmiBack V2**
Description:	**Hard disk backup software with tape drive support.**
Author:	**Moonlighter Software Inc., USA**
Contact:	**0101 407 384 9484**

Communications

Title:	**NComm Version 2 and 3.**
Description:	**Terminal software. *Shareware*! Available for evaluation from most Amiga BBSs and PD libraries.**
Author:	**Torkel Lodberg**

Title:	**Lucy**
Description:	**Offline Reader for CIX. (*Whiskyware*!). Requires Workbench 2. Available from CIX for evaluation.**
Author:	**Toby Simpson**
Contact:	**toby@cix.compulink.co.uk**

Title:	**Term 3**
Description:	**Terminal software. *Giftware*! Requires Workbench 2. Available from most Amiga BBSs and PD libraries.**
Author:	**Olaf Barthel**

Art and Rendering

Title:	**Art Department Pro 2.5**
Description:	**Graphics Conversion and Image Processing.**
Author:	**ASDG Inc., USA**
Contact:	**0101 608 273 6585**

Title:	**Digi Paint 3**
Description:	**HAM-based paint package**
Author:	**Newtek Inc., USA**
Contact:	**0101 913 354 110584**

Title:	**Video Toaster**
Description:	**Video effects and 3D rendering system (hardware & software) NTSC only. Comes with the world-famous LightWave 3D rendering software, as used in Babylon 5 and Seaquest DSV on TV.**
Author:	**Newtek Inc., USA**
Contact:	**0101 913 354 1583**

Title:	**SCALA Multimedia MM200/MM300**
Description:	**Multimedia presentation software**
Author:	**Scala Inc.**
Contact:	**0920 444230**

Development

Title:	**SAS C 6.5**
Description:	**ANSI C/C++ Development System. Various component parts, such as the editor, support ARexx.**
Author:	**SAS Institute Inc., USA**
Contact:	**HiSoft 00525 718181 (UK Distributor and Support)**

Section D
ARexx in Real Applications

As we have discovered, ARexx is very useful as a standalone programming language. Not only can we write entire programs to perform useful functions, but it can be used to put together small routines to perform repetitive tasks. However, ARexx really starts to show its power when we use it to communicate with other applications, because it uses the features of those applications as if they were part and parcel of ARexx itself.

In this section we will demonstrate two small ARexx applications which communicate with other programs. The first uses the popular text editor "CygnusED" (or CED) to do a basic error check of a file, spotting such mistakes as failing to put a space after a full stop, and not capitalising the first word in a sentence for example. The second uses Art Department Professional (ADPro) to process a large number of picture frames.

You may be thinking, "I don't have the applications you're demonstrating." This isn't a big problem because there are many other ARexx-supporting programs which can perform the same sort of operations but by using different commands. So converting the error-checker to use TurboText instead of CygnusED, for instance, is relatively straightforward; and the pictures in the second program could be converted using any one of the many image-processing and art packages which support ARexx.

Using the Example Programs

If you do have the applications we are using as examples here, then the programs can be typed in and used immediately. If you don't, you'll need to do some conversion work. Section B and C contain the information you will need in order to adapt them.

Both examples will load and run the application used if it is not already running. They do this by using the AmigaDOS shell command "run" and the program name. For this to work, the program has to be available in one of the current paths (you can find information on the "path" command in any AmigaDOS shell reference guide). For example, if Art Department Pro was stored on your hard disk partition "Work", in the "ADPro" drawer, you could add a path to it from a shell window by typing:

```
path work:ADPro add
```

You will then be able to type "ADPro" and press Return to run it directly, without even having to go to the right drawer using the "cd" command. More advanced users might want to add such a path to their startup sequence, by adding the path line above to their "user-startup" file.

TOP TIP **Remember! If there is something boring and repetitive you do often on your computer, ask yourself whether ARexx couldn't do the job instead. It might be worth writing a program to solve the problem. It may take longer initially, but it will save you time in the long run.**

The Error Checker

When you are typing out a letter it is easy to make silly mistakes which a spell checker might not find. Programs like CygnusED do not have a built-in way of checking for this, but we can add such a feature by creating a suitable ARexx script. This program will work from within CygnusED, check a document for problems (even correcting them automatically if that's what you want) and then show you some useful information about the document, such as average word length.

There are lots of nice additional features too. If run from a shell window, it will load up CygnusED and prompt you to select a file to load. If the document which is to be checked has unsaved changes, you will be prompted to save it first in case something goes wrong.

The Program

This error checker is not infallible. If you type "fro" instead of "for" for instance it won't pick it up – any more than a decent spell checker would ("fro" exists as a word in the sense of "to and fro"!). And although it checks for spaces after full stops I have not included quotation marks (double or single) in the punctuation because they can have spaces before or after them.

However, the program does demonstrate a lot of the ARexx string-handling functions, particularly those used for deleting and inserting text.

It should be relatively easy to improve on it by adding new features for it to check, and you could even turning it into a spell checker in conjunction with a suitable wordprocessor with an ARexx port. Some of the techniques used are certainly worth discussing further, however.

As mentioned in Section C, you occasionally have to put quotes around commands you wish to be sent to an external program. The convention in this book is to use single ones for commands to external functions, with double quotes for all other uses. This is to help us to see at a glance which is a command, and which is a string.

In this example, some of the commands we send to CygnusEd contain ARexx reserved words, such as "End", for instance. To avoid ARexx showing an error, we put the entire command in single quotes to force it to be sent as one command without further parsing by the interpreter.

```
/* Quote problem example */

ADDRESS "rexx_ced"
END OF LINE

EXIT
```

This, when run will generate an error:

```
+++ Error 26 in line 4: Missing or unexpected END
Command returned 10/26: Missing or unexpected END
```

Although we wanted to send the command "END OF LINE" to CygnusED, ARexx detected the END keyword and generated the error. However, by simply changing the CED command to:

```
'END OF LINE'
```

... the problem is solved. In fact if you do get strange errors like this while developing applications which control external programs, this could be the answer. If you're at all unsure, put all the commands you intend to be sent out in single quotes. You won't do any harm, and at the same time you could make it easier to spot bugs in your program.

We will also be lowering the numeric accuracy from its default level to 2 by using the NUMERIC command:

```
NUMERIC DIGITS 2
/* This is for average word length precision */
```

... the reason for this is that one of our error checker's functions is to tell you the average word length when the program has finished. Setting the level to two prevents it giving us ridiculous answers like "2.3645374", when really

"2.4" would have been more than enough (unless you are particularly fussy!).

The single most important line in the program is also worth a brief mention:

```
OPTIONS RESULTS
```

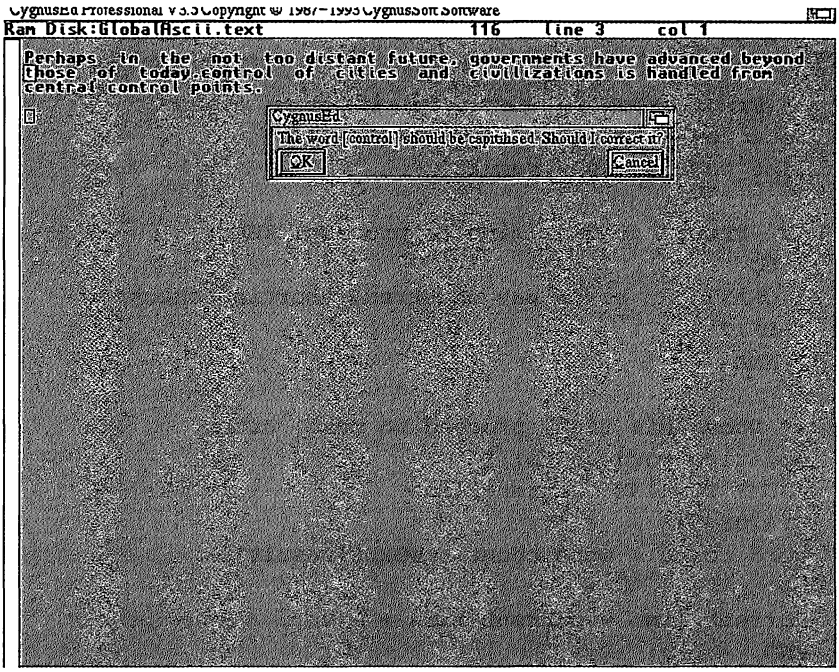
Without this, the program would not work because it would not be able to process information received back from CygnusED. This is another common problem encountered by beginners and experts alike, so do remember that you will need this statement at the start of your program if you wish to see data which is returned to you by an external function or command call.

Adapting it to work without CygnusED

There are two ways in which this program can be adapted to work without CygnusED. The first and potentially easiest way is to make it work with another text editor which supports ARexx, such as TurboText. If you do not have access to either editor, it is possible to make it work without an external application at all so that it operates as a stand-alone ARexx program. This does require some additional work, but the program has been designed to make this a little easier. The Procedure "ErrorCheckLine" makes only a couple of specific application calls, so the main program loop could be adapted to read files in from a text file, using the ARexx READLN function, and pass them directly to "ErrorCheckLine". The recommended way for doing it would be to have a separate output file, rather than modifying the existing one. At the end of "ErrorCheckLine" you would simply WRITELN out the line you have just processed to the new file.

Listing 1: the error checker

```
/*$Id: error.rexx 1.03 (17.7.94)**
**An error checker for CygnusED Professional (CED)**
**By Toby Simpson*/

OPTIONS RESULTS
/* This is for average word length precision */
NUMERIC DIGITS 2

/* If CED is not being run, then run it, and prompt for
a file to be checked */
IF ~SHOW(ports, "rexx_ced") THEN
  DO
    ADDRESS COMMAND "run ced"
    ADDRESS COMMAND "waitforport rexx_ced"

    ADDRESS "rexx_ced"

    OPEN

    /* If user selected CANCEL, abort: */
    IF ~result THEN
      DO
      OKAY1 "Operation aborted"
      EXIT
      END
  END

/* We have CED now, so set the port address: */
ADDRESS "rexx_ced"
CEDTOFRONT
/* CED Screen to the front */

/* Set up any important variables: */
LF = "0A"X
/* Line feed ASCII code */
/* Ask for user confirmation: */
OKAY2 "Are you sure you wish to continue?"
IF ~result THEN
  DO
  OKAY1 "Operation aborted"
  EXIT
  END

/* Ask if the user wants this file saved first if there
were any changes made since the last save: */
STATUS 18
/* Fetches changes since last save */
IF result > 0 THEN
  DO
  OKAY2 "Do you wish to make a back up¬
  copy"LF"before performing the check?"
  IF result THEN 'SAVE AS'

  /* If user selected CANCEL, abort: */
  IF ~result THEN
    DO
    OKAY1 "Operation aborted"
    EXIT
    END

  END

/* All ready to go, so start the check: */
'BEG OF FILE'
/* Move to first line in file */
current_line = 1
STATUS 17
/* Total lines in file */
total_lines = result
STATUS 16
```


CygnusEd Professional V3.5 Copyright © 1987–1993 CygnusSoft Software

Ram Disk:GlobalAscii.text 116 line 3 col 1

Perhaps in the not too distant future, governments have advanced beyond those of today.control of cities and civilizations is handled from central control points.

CygnusEd

The word [control] should be capitalised. Should I correct it?

Our ARexx error checker running in conjunction with CygnusEd

```
/* Total characters in file */
total_characters = result

corrections_made = 0
problems_found = 0
warnings = 0
word_count = 0
last_fs = 1
/* First word of document must be upper cased */

DO UNTIL (total_lines + 1)= current_line
  /* Fetch a line: */
  'JUMP TO LINE' current_line
  'BEG OF LINE'
  STATUS 55

  /* Error-check it (and abort if error was
  detected): */
  line_to_process = result

  IF ~ErrorCheckLine(line_to_process) THEN
    DO
    OKAY1 "An error has occurred, aborting check"
    EXIT
    END

  /* Done this line, proceed to next, after
  updating word-count: */
  'JUMP TO LINE' current_line
  STATUS 55
  word_count = word_count + WORDS(result)

  current_line = current_line + 1
  END

/* Operation finished, show statistics and information:
*/
string = "Problems Found:" || problems_found || LF
string = string || "Warnings:" || warnings || LF
string = string ||"Corrections Made:"¬ corrections_made
|| LF || LF
string = string ||"Characters in file:"¬
total_characters || LF
string = string ||"Lines in file:" total_lines || LF
string = string ||"Words in Document:" word_count¬ ||
LF

average_word_length = total_characters / word_count

string = string ||"Average word length:"¬
average_word_length
```

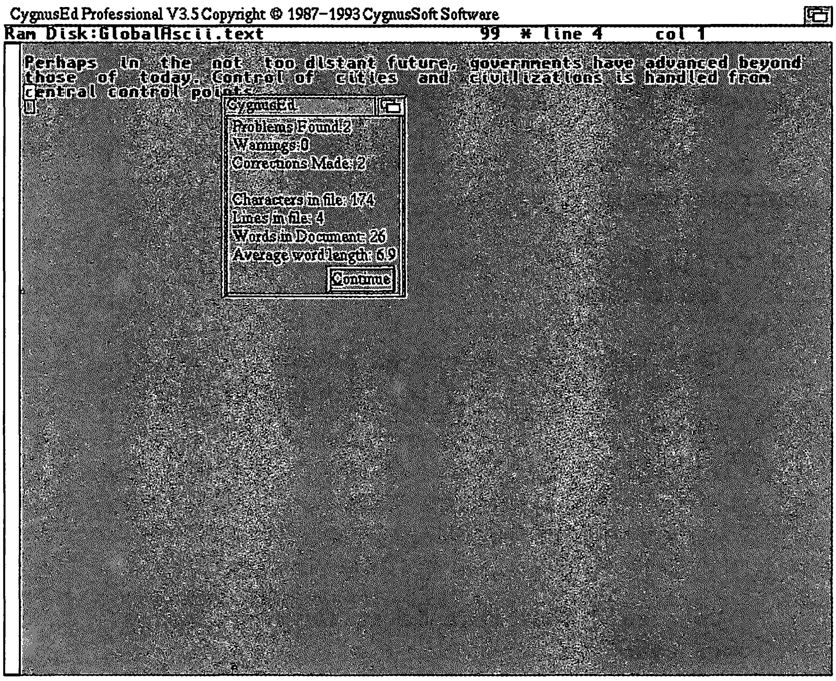
```
OKAY1 string

EXIT

/* Procedure to error check one line supplied as a
parameter. Two variables are exposed for global use,
"corrections_made" and "problems_found" - total error
corrections, and total problems found respectively.
This function returns FALSE if it has an error, or TRUE
if it succeeds.*/

ErrorCheckLine: PROCEDURE EXPOSE problems_found¬
corrections_made last_fs warnings

  /* Grab argument for processing: */
  PARSE ARG line_to_process

  /* If the line is empty, return now: */
  IF LENGTH(line_to_process) <= 1 THEN RETURN 1

  /* Initialise checker: */
  current_word = 1
  line_changed = 0
  punctuation = ":;.-,!?"
  capitalise_after_fs = ":?!."

  /* Loop through all words now: */
    DO UNTIL current_word >= WORDS(line_to_process)

    alter_word = 0

    /* Fetch the word to work on: */
    word = "" || WORD(line_to_process, current_word)

    /* Check for first word of sentence having a
    capital letter: */
```

```
IF last_fs & LowerCase(LEFT(word, 1)) THEN
  DO
  problems_found = problems_found + 1
  IF CapitaliseWord(word) THEN
    DO
    corrections_made = corrections_made + 1
    alter_word = 1
    END
  END

/* If "I" is used as a word, it must be
capitalised: */
IF word = "i" THEN
  DO
  problems_found = problems_found + 1

  IF CapitaliseWord(word) THEN
    DO
    corrections_made = corrections_made + 1
    alter_word = 1
    END
  END

/* Check for punctuation without a space after
it: */
IF VERIFY(RIGHT(word, 1), punctuation, "MATCH")¬
THEN
  DO

  IF VERIFY(RIGHT(word, 1),¬
  capitalise_after_fs, "MATCH") THEN
    DO
    last_fs = 1
    END

  ELSE NOP
```

```
    END

  ELSE

    DO

    punc_match = VERIFY(word, punctuation, "MATCH")
    last_fs = 0

    IF punc_match > 1 THEN
      DO
      problems_found = problems_found + 1

      IF AddSpaceAfterPunc(word, punc_match) THEN
        DO
        corrections_made = corrections_made + 1
        alter_word = 1

        /* Check for punctuation without a space
        on new word: */
        IF VERIFY(RIGHT(WORD(word, 1), 1), ¬
        capitalise_after_fs, "MATCH") THEN
           DO
           last_fs = 1
           END

        END

      END

    END

  /* Check for words with no vowels: */
  IF VERIFY(word, "AEIOUaeiou", "MATCH") = 0 THEN
    DO
    problems_found = problems_found + 1
```

```
      result_type = ProcessAbbrev(word)

      SELECT
        WHEN result_type = 1 THEN
          DO
          corrections_made = corrections_made + 1
          alter_word = 1
          END

        WHEN result_type = 2 THEN warnings =¬
        warnings + 1

        OTHERWISE NOP

        END

      END

    /* Make changes if required: */
    IF alter_word THEN
      DO
      /* Note word insertion point */
      insert_position = WORDINDEX(line_to_process,¬
      current_word) - 1

      /* Delete old word */
      SAY current_word
      line_to_process = DELWORD(line_to_process,¬
      current_word, 1)

      /* Insert new word */
      word = word || " "
      line_to_process = INSERT(word, ¬
      line_to_process, insert_position, ¬
      LENGTH(word))

      /* Mark this line as changed */
      line_changed = 1
      END

    current_word = current_word + 1

    END

  /* Insert new line: */
  IF line_changed THEN
    DO
    'DELETE LINE'
    TEXT line_to_process
    END

  RETURN 1

/* Function which returns TRUE if the character
supplied is lower case. */

LowerCase: PROCEDURE

  /*Get the argument: */
  PARSE ARG first_char
  IF first_char >= "a" & first_char <= "z" THEN¬
  RETURN 1

  RETURN 0

/* Function to capitalise the supplied word if required
*/

CapitaliseWord: PROCEDURE EXPOSE word

  /* Get user confirmation: */
  OKAY2 "The word [" || word || "] should be ¬
  capitalised. Should I correct it?"
  IF ~result THEN RETURN 0
/* Return without change */

  /* Make the correction: */
  character = LEFT(word, 1)

  character = UPPER(character)

  word = OVERLAY(character, word, 1, 1)

  RETURN 1

/* Process spaces after punctuation. Allows the user to
add spaces after punctuation where they should be. */

AddSpaceAfterPunc: PROCEDURE EXPOSE word

  word = ARG(1)
  offset_to_punc = ARG(2)

  SAY word offset_to_punc

  /*Get confirmation from user: */
  OKAY2 "Can I add a space after punctuation? :" ||¬
  word

  IF ~result THEN RETURN 0

  /* Make the correction: */
  word = INSERT(" ", word, offset_to_punc, 1)


  RETURN 1

/*Process abbreviations. Abbreviations should all be
upper case and should have a full stop at the end. This
function returns 0 for no action, 1 for a change and 2
```

```
for a warning. */

ProcessAbbrev: PROCEDURE EXPOSE word

  /* Get user confirmation of change: */
  corrections_made = 0

  OKAY2 "Is word [" || word || "] an abbreviation?"
  IF ~result THEN
    DO
    /* User said this is not an abbreviation: */
    OKAY1 "This word could be incorrectly spelt."
    RETURN 2
    END

  ELSE

    DO
    /* User said that this is an abbreviation: */
    lower_case = 0

    DO loop = 1 TO WORDLENGTH(word, 1)
      IF LowerCase(SUBSTR(word, loop, 1)) THEN¬
      lower_case = 1
      END

    IF lower_case THEN
      DO
      OKAY2 "Shall I capitalise the lower case in ¬
      this abbreviation?"
      IF result THEN
        DO
        word = UPPER(word)
        corrections_made = 1
        END
      END
```


CygnusEd Professional V3.5 Copyright © 1987–1993 CygnusSoft Software

Ram Disk:GlobalAscii.text 99 * line 4 col 1

Perhaps in the not too distant future, governments have advanced beyond those of today. Control of cities and civilizations is handled from central control points.

CygnusEd

Problems Found: 2
Warnings: 0
Corrections Made: 2

Characters in file: 174
Lines in file: 4
Words in Document: 26
Average word length: 6.9

We can get a report on our text which provides useful information.

```
IF RIGHT(word, 1) ~= "." THEN
  DO
  OKAY2 "Shall I add a full stop to the end?"
  IF result THEN
    DO
    word = INSERT(".", word, WORDLENGTH(word,¬
    1), 1)
    correction_made = 1
    END

  END

END
```

```
  RETURN corrections_made

/* End of File */
```

The Picture Modifier

One of the things that the Amiga is particularly good at is processing video information. Because of the poor quality and/or high price of many applications like 3D picture-renderers and landscape-generators, many Amiga users spend a lot of time dealing with large batches of video images, a boring, laborious task – especially if you are dealing with several hundred frames that all need one simple alteration. You could solve this problem by spending a quiet night in with DeluxePaint, or save yourself a lot of time by writing a short ARexx program to do the entire task for you. You can now spend a loud night out somewhere else and let your computer get on with the hard work – it's a lot less likely to get bored and make silly mistakes!

This example demonstrates some of the features of Art Department Professional (ADPro), from ASDG, in the field of image processing. ADPro allows the user to load in a picture file (from any number of different formats), process it in some way and then allow it to be saved back out to disk, again in one of many different graphics formats. A lot of users find it handy for converting GIF (Graphics Interchange File) picture files from IBM-PC compatible computers to IFF (Interchange File Format) for use on the Amiga. But it can do much more than that, it can change a picture to a different size or its number of colours – and this is where it starts to get useful for the more advanced user.

Murphy's Law states that the program that is generating all of your pictures will not able to generate them in the format you'd really like. A particularly nasty example might be if you had a video digitiser, which took running video from a household VCR and converted it into hundreds of individual 640x256 Amiga IFF screens in 256 colours. Unfortunately, you want them in Lo-Res 320x200 in 16 colours. Armed with DeluxePaint, ADPro, or another such application, you can solve the problem, but it is going to take hours. This is where the ARexx facility offered by some image programs, such as ADPro being demonstrated here, becomes invaluable. We are able to write a program, only a handful of lines long, which will do the entire job for us without any intervention at all.

The Program

This program will process a large number of pictures by loading them into ADPro, converting them to an alternative format, displaying them, and finally saving the new file out. The program makes a lot of assumptions about the operations performed, but it's easy to customise. It expects the files to be in the format:

```
framename.1
framename.2
framename.3
```

...and so on You can specify what the filename part is and, when numbering begins, how many frames there are.

A couple of the programming techniques used in this program are also useful for other ARexx programs. One of the problems you encounter when you're calling multiple external host commands is checking to see if each one returns an error. If external commands set the RC (return code – see Section C) to a value greater than the current fail-at level (normally 10), then the script will show an error.

We can deal with these individually, but an easier way of doing it is to use the SIGNAL ERROR interrupt (see the reference section, E, for more information on SIGNAL). This way, we can handle all errors returned by commands sent to ADPro in one handy routine:

```
/*This is the error interrupt when a command sent to
ADPro failed.*/

ERROR:

  /* If ADPro is running still, kill it: */
  IF SHOW(ports, "ADPro") THEN ADPRO_EXIT

  /* Now show error message: */
  SAY "Error: Command returned Error code at line ¬
  "SIGL

  EXIT
```

This is a simple case; it will terminate ADPro and show an appropriate message on the screen together with the line number in the ARexx program where the error occurred.

Notice also that we are using the built-in DATATYPE() function to confirm that the user enters a number when they are required to. You can never guarantee what the user of your ARexx program is going to type in. It's better to detect these and give them a chance to correct it, rather than having your ARexx program fail because of an error later on:

```
  IF DATATYPE(number) = "NUM" THEN
    DO
    number_valid = 1
    END
```

Of course, in this particular example we could have used ADPro's "GETNUMBER" command which shows a neat requester on the screen with a box for entering a number. ADPro will ensure that only a number is actually entered.

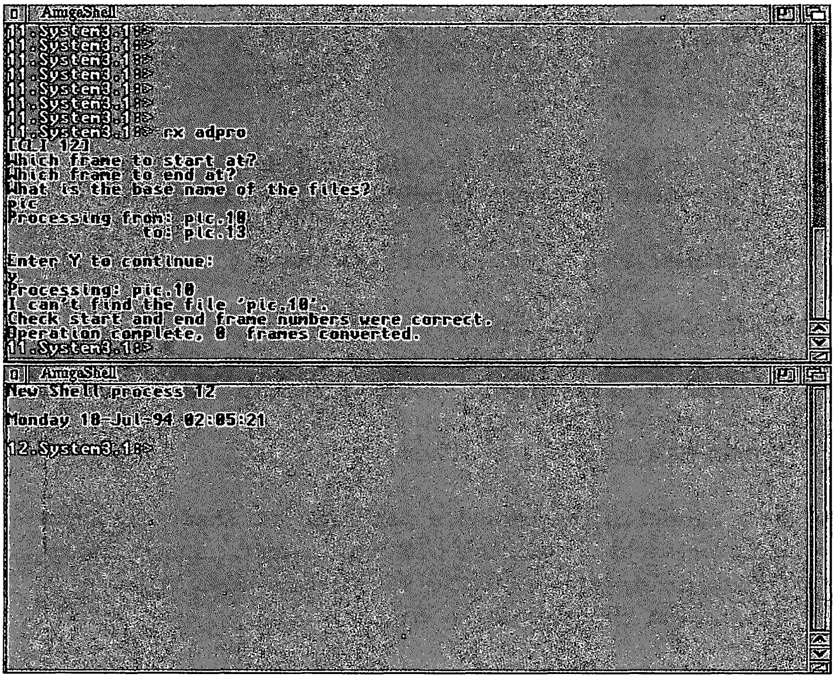
Here is how we could adapt the GetNumber function in the listing to use ADPro's function instead of keyboard entry on the shell window:

```
*/This function fetches a number. If the user enters
something which is not numeric (checked with the ARexx
DATATYPE function) then the user is prompted to enter
it again.*/

GetNumber: PROCEDURE

  ADPRO_TO_FRONT

  GETNUMBER "Enter a Number"

  ADPRO_TO_BACK

  RETURN ADPRO_RESULT
```

ADPro returns additional information in the ADPRO_RESULT variable, and in the case of the GETNUMBER function that is the number which was entered. We encompass the GETNUMBER call inside functions to bring the ADPro screen to the front so that the user can see the requester that they have to type into! If the user clicks on the number-requester's CANCEL button, ADPro will return an RC of 10, which will cause a jump to our error processing function "ERROR:".

Listing 2: the file processor

```
/* Art Department Pro picture processor */

/*$Id: adpro.rexx 1.01 (18.7.94) */
/*This is a "work-horse" program, one that performs a
large amount of otherwise repetitive work automatically,
using the ARexx abilities of ADPro */

OPTIONS RESULTS
NUMERIC DIGITS 2
/* This is for average word length precision */

SIGNAL ON ERROR

/*If ADPro is not being run, then run it, and prompt
for a file to be checked. */
IF ~SHOW(ports, "ADPro") THEN
  DO
    ADDRESS COMMAND "run adpro"
    ADDRESS COMMAND "waitforport ADPro"

    ADDRESS "ADPro"
  END

/*We have ADPro now, so set the port address: */
ADDRESS "ADPro"
ADPRO_TO_BACK

/* Set up the ASCII linefeed string */
LF = "0A"X

/* Fetch information from the user about the conversion
process: */
SAY "Which frame to start at?"
frame_start = GetNumber()

SAY "Which frame to end at?"
frame_end = GetNumber()

/*The frame end must be larger than start, so check
this: */
IF frame_end < frame_start THEN
  DO
  SAY "End frame cannot be large than start frame!"
  EXIT
  END

SAY "What is the base name of the files?"
PARSE PULL base_name

/* Show the user the information for confirmation: */
SAY "Processing from:" GetFileName(frame_start)
SAY "              to:" GetFileName(frame_end)

SAY LF || "Enter Y to continue:"
PULL confirm
IF UPPER(confirm) ~= "Y" THEN
  DO
  SAY "Operation aborted"
  EXIT
  END

/* Now we initialise things before the main loop: */
done = 0
current_frame = frame_start
frames_processed = 0

/* This is the main processing loop, it ends when the
job is complete or an error occurs. */
DO UNTIL done

  /* Generate correct file name: */
  file_name = GetFileName(current_frame)
  SAY "Processing:" file_name

  /* Confirm this file exists by trying to open it: */
  IF ~EXISTS(file_name) THEN
    DO
    SAY "I can't find the file '" || file_name || "'."¬
    || LF || "Check start and end frame numbers ¬
    were correct."
    done = 1
    END

  ELSE

    DO
    /* Perform processing operation on this file: */
    IF ~ProcessPicture(file_name) THEN
      DO
      /* An error occurred, ask if user wants to
      ignore it and continue anyway: */
      SAY "An error occurred processing '" || ¬
      file_name || "'."
      SAY "Continue anyway? Enter Y to continue:"
      PULL confirm
      IF UPPER(confirm) ~= "Y" THEN
        DO
        SAY "Operation aborted"
        done = 1
        END
      ELSE
        NOP

      END

    ELSE

      DO
      frames_processed = frames_processed + 1
      SAY ".. Complete"
      END

    END

  /* Proceed to next frame, and exit if we have
  just done the last: */
  current_frame = current_frame + 1
  IF (current_frame = frame_end + 1) THEN done = 1

  END
/* Operation complete! Inform user: */
ADPRO_EXIT

SAY "Operation complete," frames_processed " frames ¬
converted."

EXIT

/*This function fetches a number. If the user enters
something which is not numeric (checked with the ARexx
DATATYPE function) then the user is prompted to enter
it again. */

GetNumber: PROCEDURE

number_valid = 0

  DO UNTIL number_valid

    PULL number

    IF DATATYPE(number) = "NUM" THEN
      DO
      number_valid = 1
      END

    ELSE

      DO
      SAY "Please enter a numeric value, '" || ¬
      number || "' is not a number".
      END
    END

  RETURN number

/* This function builds a complete filename for the
picture number argument provided and returns it. */

GetFileName: PROCEDURE EXPOSE base_name

  PARSE ARG picture_number

  file_name = base_name || "." || picture_number

  RETURN file_name

/*This function processes a picture. It loads it into
ADPro, converts it to the GIF file format, and changes
the resolution from 640x256 16 colours down to 4
colours, and then saves it out. */

ProcessPicture: PROCEDURE

  /*Fetch the picture name argument: */
  PARSE ARG picture_name

  /* Bring ADPro to the front, set loader and load
  picture: */
  ADPRO_TO_FRONT

  LFORMAT "IFF"

  LOAD picture_name

  /* Set to 4 colours, redo picture and show it for
  1 second on screen: */
  RENDER_TYPE 4
```



```
11.System3.1B>
11.System3.1B>
11.System3.1B>
11.System3.1B>
11.System3.1B>
11.System3.1B>
11.System3.1B>
11.System3.1B> rx adpro
[CLI 12]
Which frame to start at?
Which frame to end at?
What is the base name of the files?
pic
Processing from: pic.10
              to: pic.13

Enter Y to continue:
y
Processing: pic.10
I can't find the file 'pic.10'.
Check start and end frame numbers were correct.
Operation complete, 0  frames converted.
11.System3.1B>
```

```
New Shell process 12

Monday 18-Jul-94 02:05:21

12.System3.1B>
```

Our file processor in action.

```
EXECUTE

ADPRO_DISPLAY
PAUSE 50

/* Set save format and save picture with .out
appended to it: */
SFORMAT "GIF"

output = picture_name || ".out"

SAVE output SCREEN

RETURN 1
```

```
/* End of file */

/* This is the error interrupt when a command sent to
ADPro failed. */

ERROR:

  /* If ADPro is running still, kill it: */
  IF SHOW(ports, "ADPro") THEN ADPRO_EXIT

  /* Now show error message: */
  SAY "Error: Command returned Error code at line ¬
  "SIGL

  EXIT
```

Other Applications

There are many ways to use ARexx. This section has just suggested a couple, with examples of the sort of time-saving program that will perform repetitive tasks automatically. Here are a few other possible uses of ARexx that might give you a few ideas for other uses.

Communications

Users of modems who regularly call bulletin boards could find ARexx invaluable. It can be very tedious having to remember what to type to connect to any particular service, especially since some require you to type in all kinds of stuff, including a username and a password, before you get started This is just the job for a computer, because the sequence for connecting (or "logging on") to a bulletin board is the same every time for that given service.

If you use a terminal program with an ARexx port, then you can program ARexx to dial the right phone number for you and "type in" all the necessary information. Then all you have to do is write an ARexx script for each bulletin board or service that you use regularly.

Here is an example which uses the popular shareware terminal software NCOMM. It can be adapted to work with "Term" as well. It's a program for really lazy people who can't even be bothered to move the mouse far enough to select a phone book entry! For it to work, a file called "PhoneBook" must be created consisting of lines containing phonebook entries, for example:

```
cix 0813901255 ogin:|qix user|<my_username>¬
assword|<my_password>
```

Respectively: the first parameter is the service name, followed by the telephone number to dial, and up to three wait and response strings ("ogin" and "assword" avoid any problems with upper or lower case "l"s and "p"s) . The script, once it has dialled the number, will go through each of the wait-response sequences. If you need less than three, put a "-" for unused ones, for example:

```
a_bbs 123456789 Name:|fred Password:|juggle -¬
```

The | symbol is used to separate the wait string and response. NCOMM will wait for the text left of the | and then respond with the text on the right of the |.

In communications, ARexx is an eternal asset. We have shown above how to use it to log on to a service. It could then perform a number of operations automatically, fetching information, then disconnecting and allowing you to view it at your leisure – and while no longer connected to the phone line and racking up the bill.

```
/* Auto Dialler Program by Toby Simpson*/

/* $Id: bbs.rexx 1.02 (17.7.94) */
/* A simple program which dials up and connects to a
bulletin board using the popular shareware NCOMM
program. /*

OPTIONS RESULTS

/* If NCOMM is not being run, then run it, and prompt
for a file to be checked. */
IF ~SHOW(ports, "ncomm") THEN
  DO
    ADDRESS COMMAND "run ncomm"
    ADDRESS COMMAND "waitforport ncomm"
    END

/* We have NCOMM loaded now, so set the port address:
*/
ADDRESS "ncomm"

/* Reset the modem:*/
HANGUP

/* Ask user for the service to dial: */
BEEP  /* Brings screen to front and alerts user */

STRINGREQ "Enter service to dial"
IF RC ~= 0 THEN
  DO
  SAY "Operation aborted"
  EXIT
  END

service = UPPER(RESULT)
```

```
/* Look it up in the phone book: */
IF ~OPEN("InFile", "ncomm:My_Phone_Book") THEN
  DO
  SAY "Can't open phone-book"
  EXIT
  END

success = 0
done = 0

DO WHILE ~done

  book_line = READLN("InFile")

  IF UPPER(WORD(book_line, 1)) = service THEN
    DO
    done = 1
    success = 1
    END

  END

CLOSE("InFile")

/* Exit if we could not find it in the phone book: */
IF ~success THEN
  DO
  SAY "Can't find information for" service "in¬
  phonebook."
  EXIT
  END

/* Strip phone number and commands from phone book: */
phone_number = WORD(book_line, 2)
command1 = WORD(book_line, 3)
command2 = WORD(book_line, 4)
command3 = WORD(book_line, 5)
```

```
/* Now dial the service: */
DIALNUMBER "081 390 1255"

IF command1 ~= "-" THEN ProcessCommand(command1)
IF command2 ~= "-" THEN ProcessCommand(command2)
IF command3 ~= "-" THEN ProcessCommand(command3)

/* Operation complete, show a simple requester to say
so: */
SIMPLEREQ "Connected"

EXIT

/* A function to wait for a given message to arrive
from the modem and respond with a set command: */

ProcessCommand: PROCEDURE

  PARSE ARG command

  offset = INDEX(command, "|")

  wait_string = LEFT(command, offset - 1)
  respond_string = RIGHT(command, LENGTH(command) -¬
  offset) || "\n"

  WAIT wait_string
  SEND respond_string

  RETURN 0
```

Word Processing and Text Editing

Wordprocessing can also involve repetitive tasks like reformatting documents or making simple changes. Again, an ARexx script can help, not only to perform simple operations, like counting the number of occurrences of a list of words, but you can add new features to a product.

It would be possible, for example, to write an ARexx program which generated real indexes for documents. It would be easy to add an index generator for CygnusED. All you would have to do is write a program in ARexx which searched a document for the word you wish to put in the index, note which page(s) it appeared on, and generate a suitable output file. If you were really smart, you could allow a list of words to be specified and automatically sort the index into alphabetical order, creating an output which might look like this:

```
INDEX

apples ............................................................. 2, 3, 8
oranges .................................................. 1, 3-7, 100, 200
pears ............................................................ 4, 45-60
```

For wordprocessing and text applications in particular, the uses of ARexx are highlighted. It is easy to cobble together programs which will perform a number of complex operations in the blink of an eye. Files can be manipulated in many ways, they can be chopped into sections, searched, organised, updated, reformatted... the list goes on.

Multimedia

This is a common buzzword these days and something the Amiga is particularly good at – the processing of audio/visual information from interactive experiences to complex presentations. Amiga programs such as AmigaVision and Scala both support ARexx and it is possible to control both of these applications – creating visual presentations which can be changed entirely with a couple of tweaks to an ARexx program.

Section E
Reference Section

This section contains reference information about ARexx and starts with the basic specification and features of the ARexx language before moving on to the ARexx keywords reference. Each keyword in the ARexx language is discussed and there is an example of its use. Further, more-detailed examples can be found in Section B. The function reference which follows lists all the built-in functions provided for the programmer and those present in the rexxsupport.library function library, and also mentions a few other external function libraries. Finally, there is a description of all the utility commands that are supplied with ARexx in the RexxC drawer.

Language Specification

ARexx is the Amiga version of REXX, an interpreted language written by Mike Cowlishaw of IBM and designed to be easy to learn and use, yet both powerful and flexible. The Amiga port was done by William Hawes in 1987. The basic features of ARexx are as follows:

Interpreted language

Although slower than compiled languages such as "C", an interpreted language has the advantage that you do not need to compile a program in order to run it and, because the interpreter is always resident, you can create and run programs from programs. For more on the advantages of interpreted languages, see the "interpret" command later in this reference section and in Section B.

Typeless data

Most high-level computer languages require that you state what type a variable is before you use it; that is, tell the language what sort of data you will be storing there, like floating point numbers, strings or integer numbers. ARexx sorts all this out for you when you use the variables.

Resource tracking

This is a powerful feature. In the case of ARexx, the interpreter tracks all the resources your program uses and ensures that they are released for you at the end, even if you don't remember to release them. It cannot, however, track non-ARexx resources, so if you were to allocate memory for your program using a system library call, ARexx would not be able to free it for you. Resource management of this sort ensures that unnecessary memory allocations are freed.

Tracing, trapping and debugging features

ARexx comes with built-in debugging features that allow the programmer to see and control the execution of a program step by step. This helps to reduce development time considerably. See Section B for further debugging information, and also the "Utility commands reference" later in this section.

Function libraries

This allows external programs to add to the functionality of ARexx, by providing additional functions for a program to use. This is one of the key features of ARexx. See Section C for further information on ARexx's ability to talk to other applications, and the discussion of Functions in section B.

ARexx Keywords Reference

There are 30 keywords in the ARexx language. This reference section details each of these in alphabetical order, together with its correct syntax, a discussion about its use and one or more examples. Here are the conventions used for describing the syntax:

All keywords are shown in upper case, all information to be supplied to the keyword is shown in lower case.

|	(vertical bar) alternative selections are separated by this.
{}	(braces) required alternatives are enclosed by braces.
[]	(square brackets) used to separate optional instruction parts.

To retain continuity, these syntax conventions are the same as those used by Commodore in their documentation. The best example of this convention in use is shown by the CALL keyword:

```
CALL {symbol | string} [expression] [,expression, ...]
```

This shows that first a symbol or string is required, then several optional expressions can also be provided.

ADDRESS

Syntax:

```
ADDRESS [[symbol | string] | [VALUE] [expression]
```

Specifies the host address for commands, which are not ARexx keywords, to be sent to.

This is a flexible command which can be used to change the host address. ARexx remembers two hosts, the current one and the previous one. Issuing ADDRESS on its own swaps between the two. This is useful when you change host for a few commands and then wish to revert to the previous one.

You can also specify the name of the host address you wish to change to. It's important to remember that these are case sensitive, and that if you do not put quotes around them then ARexx will convert to upper case, which may not be the desired result.

If the VALUE keyword is present, then the result of the expression decides the host address that will be selected. If the first part of the expression is neither a symbol nor a string, the VALUE keyword can be omitted. It's best to put this keyword in when it is intended, just in case. For example:

```
ADDRESS        /* Toggles between current and previous ¬
host */
ADDRESS Lucy_OLR    /* Selects host "LUCY_OLR" */
ADDRESS "Lucy_OLR"  /* Selects host "Lucy_OLR" */
ADDRESS VALUE variable_name     /* Selects host to ¬
the contents of variable_name */
```

ARG

Syntax:

```
ARG [template] [,template...]
```

Gets argument strings for the program and assigns them to the variables specified in the template.

This command is a shorthand form of "PARSE UPPER ARG". It is used to recall arguments which were specified when running the script and to place them into variables. ARG returns upper case letters. When the program is invoked as a command there is only one argument string, but programs run as a function can have up to 15. (See PARSE for further information.) For example:

```
/* Arg test */
ARG first,next
SAY "Argument1:"first "Argument2:"next
```

This produces the result:

```
7.System3.1:> rx arg fred john
Argument1:FRED JOHN Argument2
```

Since this example was run as a command, there is just the one argument string so both specified arguments went to the variable "first" and nothing was put in "next". We could now, of course, split this single argument up into component parts using a small program.

BREAK

Syntax:

```
BREAK
```

Exit from a DO instruction or from within an INTERPRET'd string.

BREAK is used to allow you to exit prematurely from a DO loop. This can be quite handy. It only BREAKs from one DO loop, so if you are within several DO loops, you'll only exit from the current one. For example:

```
/* Break example */
DO loop = 1 TO 10
  IF loop = 3 THEN BREAK
  SAY loop
  END
```

This produces the result:

```
7.System3.1:> rx break
1
2
```

The IF causes the loop to terminate when the variable "loop" reaches 3, and before the SAY keyword gets to show it.

CALL

Syntax:

```
CALL {symbol | string} [expression] [,expression...]
```

Call an internal or external function with optional arguments.

Any result from a function called using "CALL" is placed in the special RESULT variable. This can then be optionally interrogated by the program to see what happened. Not getting a result is not treated as an error because some functions perform actions without returning results. The expressions are evaluated and become the arguments to the function. For example:

```
/* Call example */
CALL ADDRESS()
SAY RESULT

test = ADDRESS()
SAY test
```

This produces the following:

```
System3.1:> rx call
REXX
REXX
```

Note that in this case, the function called does return a result and, as shown, the CALL example. CALL is not actually required to use the function.

DO

Syntax:

```
DO [[variable=expression] | [expression] [TO
expression] [BY expression]] [FOR expression] [FOREVER]
[WHILE expression | UNTIL expression]
```

Marks the start of a group of statements to be executed as a block. This group is ended using the END keyword.

DO is a flexible way of looping in ARexx. There are several kinds of loops which can be executed from a DO keyword. The simplest is a DO FOREVER loop in which execution of the loop continues forever, or until a BREAK keyword is issued to exit from the loop. There are several other simple ways in which DO can be used apart from DO FOREVER. For instance, DO UNTIL loops continue until the specified expression becomes true, and DO WHILE loops continue while the expression is true.

You can also specify the start and end points of a loop and the increment by which the loop increases. This is the "DO variable = expression TO expression [BY expression]" type of loop. Using DO by itself, followed by a group of statements and an END, means that that group will be executed only once. This is usual with IF statements so that a whole load of statements can be executed rather than just one. See IF for an explanation. For example:

```
/* Do examples */

/* Loop WHILE loop is smaller than 12, printing 12x
tables as we go */
loop = 1
DO WHILE loop < 13
  SAY loop "x 12 = " loop*12
  loop = loop + 4
  END

/* Loop from 1 to 12 in steps of 4, with some 12x
tables too */
DO loop = 1 TO 12 BY 4
  SAY loop "x 12 = " loop*12
  END
```

Both of the above DO loops achieve the same result, using slightly different methods. The different DO loops can be used for different purposes, in this case since we have fixed start, end and increment values, the second version is far better. The results of the above script is as follows:

```
7.System3.1:> rx do
1 x 12 =  12
5 x 12 =  60
9 x 12 =  108
1 x 12 =  12
5 x 12 =  60
9 x 12 =  108
```

DROP

Syntax:

```
DROP variable [variable...]
```

This tells your program to forget about the values in the named variables.

DROP allows you to reset one or more variables to their uninitialised (empty) states, which is the name of the variable itself. If you DROP a variable which does not exist, an error will not occur. For example:

```
/* Drop example */
a = "Lobster"
b = "Thermidore"

DROP b
SAY a b
```

The results of this script are as follows:

```
7.System3.1:> rx drop
Lobster B
```

We DROP'd the B variable, which reset it to its own name, in upper case.

ECHO

Syntax:

```
ECHO [expression]
```

ECHO is the same as the SAY keyword. See SAY for further information and an example.

ELSE

Syntax:

```
ELSE [;] [conditional statement]
```

Provides the alternative condition for an IF statement.

ELSE is only valid from within an IF structure. IF allows you to make a statement happen if an expression is evaluated to TRUE. By using the ELSE instruction you can also make a statement happen if the expression did not evaluate to TRUE. The ELSE is only relevant to the nearest previous IF statement. For example:

```
/* IF example */
PULL age
IF age > 30 THEN SAY "You're old"
  ELSE SAY "You're young"
```

Running this script twice with the ages 20 and 40 specified produces the results:

```
7.System3.1:> rx if
20
You're young
7.System3.1:> rx if
40
You're old
```

END

Syntax:

```
END [variable]
```

Terminates a group of statements started with a DO instruction.

See DO for examples. If the optional variable part is specified then the END refers to the DO loop using that variable as a counter, of the type "DO i=1 TO 10...". It's not normally necessary to specify a variable name.

EXIT

Syntax:

```
EXIT [expression]
```

Terminates the execution of a program.

When the EXIT instruction is found the program ends immediately and the optional expression is passed back to the caller of the program as a result. In the case of a script run from the shell this will be a return code or, if the program was called as a function, a secondary result – normally used to specify an error. For example:

```
  EXIT 20
  EXIT
```

IF

Syntax:

```
IF expression [THEN] [;] [conditional statement]
```

If the supplied expression evaluates to TRUE the conditional statement is executed.

IF is a key decision-making instruction in ARexx. By using the ELSE instruction you are also able to execute an instruction if the expression evaluated to FALSE (see ELSE). Only one conditional statement is allowed but ,by using a

DO END structure, you are able to execute a group of statements. For example:

```
/* IF example */
PULL age
IF age > 30 THEN DO
    SAY "You're old"
    SAY "Well, not THAT old."
    END
  ELSE SAY "You're young"
```

Which produces the results:

```
7.System3.1:> rx if
40
You're old
Well, not THAT old.
7.System3.1:> rx if
10
You're young
```

Note the use of the DO END structure to allow us to execute more than one statement after the THEN part of our IF.

INTERPRET

Syntax:

```
INTERPRET expression
```

Treats the expression as though it was actual ARexx program source. The expression is first evaluated and then executed. The source can be an entire separate ARexx script if you want. A BREAK statement can be used to terminate the INTERPRET'd statements and return control to the main program. The only limitation is that you cannot define labels within the interpreted source. Although ARexx won't object to this, only labels in the main program are looked at if you attempt to go to one using SIGNAL, for example. If your INTERPRET'd source has more than one statement in it, you can separate them using the ; (semi-colon)

INTERPRET is an extremely powerful instruction and one which is found almost only in ARexx. It is one of the benefits gained from ARexx being an interpreted rather than a compiled language in that, at all times, the interpreter has to be present when running ARexx, so INTERPRET can easily work by treating its parameters as an extension to the source itself. You can write ARexx programs which write their own small sub-programs and run them themselves. On the surface this might not appear to be that useful, but you'll encounter a number of applications where this can be put to good use. In Section B we saw two examples used, one to show how an advanced calculator program can be constructed using only a few lines of ARexx; plus the extremely handy ARexx shell, where you can just type in ARexx statements and see the results immediately – very good for testing things, controlling host applications and viewing results quickly. For example:

```
/* INTERPRET example */
PARSE PULL instruction
INTERPRET instruction
```

If this script is run, you can type any valid ARexx statement in and view the result of it's execution immediately.

ITERATE

Syntax:

`ITERATE [variable]`

Terminates the current iteration of a DO statement and begins the next.

This is a useful instruction. If you are in a long DO..END statement block and, near the beginning, you have a check which requires you to skip that loop and proceed to the next, you would normally have to build complex IF blocks to do the job. The ITERATE instruction effectively skips all following instructions to the END associated with the current DO loop and goes straight to it. If there are more loops to do, they will then be continued as normal. The optional variable parameter is used to specify directly which DO range is to be left. This is particularly important if there are a number of nested DO..END loops and you wish to state which DO is to be affected.

Errors occur if the variable did not match a currently valid DO..END block, or if the program is not currently within a DO statement block. For example:

```
/* ITERATE example */

SAY "We don't like odd numbers"
DO loop = 1 TO 20
  /* Skip the number 10 always */
  IF loop = 10 THEN ITERATE

  /* We divide our counter by 2 and if the remainder is
greater than 0 then it must be an odd number, so
ITERATE */
  IF (loop // 2 > 0) THEN ITERATE

  /* If it's odd we won't get here */
  SAY loop
  END
```

In the above example we use the remainder operator to decide if the loop counter is odd or even, and then call ITERATE if it is odd, thus getting a list of even numbers only, except the number 10 – which is also skipped.

LEAVE

Syntax:

```
LEAVE [variable]
```

Exits the current DO..END statement block immediately.

This works a little like the ITERATE instruction except that it terminates the current DO..END statement block without proceeding with any additional iterations. The optional variable can be used to specify which DO..END block is affected by the command. This can be useful for exiting from a particular range from within a nested group of DOs.

Errors occur if the variable did not match a currently valid DO..END block, or if the program is not currently within a DO statement block. For example:

```
/* LEAVE example */

DO FOREVER

  /* Get a number from the user */
  SAY "Enter a number, or 99 to exit"
  PULL number

  /* If it's 99 leave this DO block immediately */
  IF number = 99 THEN LEAVE

  SAY "Well done, you entered "number
  END
```

```
/* We got here, user must have typed 99 */
SAY "Goodbye!"
```

NOP

Syntax:

```
NOP
```

No Operation. Instruction does nothing.

The NOP instruction has only one useful application – for dealing with the confusion of ELSE statement in compound IF statements. If you have more than one IF from within an IF, then specifying an ELSE clause could easily become confused with the wrong IF block. For example:

```
/* NOP */

SAY "Enter a number from 1 to 20"
PULL number

IF number > 10 THEN

  IF number = 12 THEN SAY "It was a twelve"
    ELSE NOP
/* Ensure the next else does not get confused with this
if statement */

  /* This else now talks to the first IF */
  ELSE SAY "It was smaller or equal to 10 but not the
magic number!"
```

If you miss out the NOP statement in the above example, the message saying whether the number entered was less than or equal to 10 will never be shown.

NUMERIC

Syntax:

```
NUMERIC {DIGITS | FUZZ} expression
NUMERIC FORM {SCIENTIFIC | ENGINEERING}
```

Sets options relating to the way in which ARexx deals with numeric precision and format.

It is often necessary to change the way in which numbers are dealt with in ARexx, particularly if you are dealing with floating point values. The NUMERIC statement is a flexible way of telling ARexx how to look at numbers.

The DIGITS and FUZZ options are concerned with the numeric precision and accuracy of internal calculations. You can use DIGITS to specify how many numbers of precision are used with arithmetic calculations, and FUZZ can be used to say how many digits are ignored in numeric comparisons. When dealing with extremely small differences between numbers, it's often useful to be able to make small differences count as the same in a comparison operation. The FUZZ setting must be less than the current DIGITS setting, and in both cases the expression which gives both values should evaluate to a positive whole number. The default for DIGITS is 10, and the maximum is 14.

The two FORM options are concerned with numbers which will require exponential notation to be expressed, and tell ARexx which format to use – either engineering or scientific. In scientific notation the exponent is adjusted so that the mantissa (the bit to the right of the decimal point) for non-zero numbers is between 1 and 10, whereas in engineering notation, the number is normalised so that its exponent is a multiple of 3 and the mantissa, if non-zero, is between 1 and 1,000. SCIENTIFIC is the default. For example:

```
/* NUMERIC example */

NUMERIC DIGITS 3
NUMERIC FORM ENGINEERING

/* Enter a number */
SAY "Enter a number:"
PULL number

/* This forces the above to be converted to a number so
that the NUMERIC statement's effects occur */
number = number + 0;

SAY "That number, through the current NUMERIC settings
is" number
```

You can change the NUMERIC settings at the start of the example to show the effect of other options when dealing with the resultant number format and precision.

OPTIONS

Syntax:

```
OPTIONS [FAILAT expression]
OPTIONS [PROMPT expression]
OPTIONS [ [NO] RESULTS]
OPTIONS [ [NO] CACHE]
```

Sets a number of internal options relating to the way in which scripts are executed.

There are a number of options dealing with the way in which your programs are run. The OPTIONS statement allows you to change these values. Using OPTIONS by itself will restore all the options to their default values, which are;

CACHE ON; NO RESULTS; FAILAT, the script caller's "failat" value (normally 10); and PROMPT "" (empty string).

The CACHE option is used to switch the internal instruction caching system to either on or off. This defaults to ON and there is no real reason why you should wish to change this. ARexx caches instructions to speed up program execution and, since it keeps the previous few instructions executed in memory, cacheing means loops can execute very quickly.

Often when you call an external program to perform a function you will expect a result back. By default ARexx does not ask for results from external programs. If you wish to get information back you have to switch RESULTS on by using the statement OPTIONS RESULTS. When you are writing scripts to control other applications, it is common for the only bug to be the fact that you have omitted to switch RESULTS on and none of the functions are returning any information. External function hosts can still return an error code in the RC variable. See Section C for further examples of RESULTS and the RC return code.

The PROMPT expression is evaluated and assigned to the text prompt shown before a PARSE PULL response is required. Normally this is set to a blank and you get no prompt for entering information. For example:

```
/* OPTIONS example */

OPTIONS PROMPT "Well? Eh?>"
PARSE PULL string
SAY "You said" string

/* Switch results on before calling something */
OPTIONS RESULTS
ADDRESS "ProWrite"
```

```
ScreenToFront
StyleBold
Type "This is in Bold (but only if you have ProWrite
running!)"
```

OTHERWISE

Syntax:

```
OTHERWISE [;] [conditional statement]
```

Can only be used in connection with a SELECT statement, and its conditional statement is run only if none of the WHEN statements were executed.

When using a SELECT statement you can deal with a number of options with WHENs. Often, however, it is necessary to be able to deal with the default case, which happens if none of the WHENs succeeded. OTHERWISE allows you to do this. It can only be used from within a SELECT block. Although it is not necessary to have an OTHERWISE in your SELECT/WHENs, if you don't have one and none of your WHENs are executed, ARexx will generate an error. For example:

```
/* OTHERWISE example */

SAY "Enter a number"
PULL number

SELECT
  WHEN number < 10 THEN SAY "Smaller than 10!"
  WHEN number > 50 THEN SAY "Greater than 50!"
  OTHERWISE SAY "Something between 10 and 50"
  END
```

PARSE

Syntax:

```
PARSE [UPPER] inputsource [template] [,template ...]
```

Extract one or more sub-strings from a string defined in the input source and assign them to named variables as defined in the optional templates.

PARSE is a very complex instruction. For a full discussion on how it works see "The PARSE statement" in Section B. PARSE allows you to take a string and split it into a number of sub-strings using a template to define how they are divided. As well as splitting your own string variables, PARSE has a number of different options for the input source to define where the main string comes from. The optional UPPER keyword before the input source makes ARexx convert the entire source string to upper case characters before applying any template options to it. The templates can be omitted if the only intention is to create a single string from the input source.

	TABLE OF VALID INPUT SOURCES
ARG	The input string comes from arguments supplied to the ARexx program when it was initially executed. (See the ARG() internal function for further methods of extracting program arguments)
EXTERNAL	Works like the PULL option but, instead of the input coming from the current input source (normally the shell window from which the script was run), it comes from the STDERR input stream – if it is defined. If you open a global tracing window using the TCO utility, this will be treated as STDERR. All input that comes through the STDERR method does not affect any data which has been PUSH'd or QUEUE'd (see the

descriptions of these two instructions). This is particularly useful for inputting debugging information. You can open your own STDERR console window using the ARexx OPEN() function. Here is an example:

```
/* PARSE EXTERNAL example */
CALL OPEN("STDERR", "CON:0/60/640/100/STDERR Window!")
SAY "Enter a number"
PARSE EXTERNAL number
SAY number
CLOSE ("STDERR")
```

This will open a new window called "STDERR Window!" and will get the input from there instead of the main shell window. A prompt is given for strings inputted using EXTERNAL as defined by OPTIONS PROMPT.

NUMERIC The input string is made up of the current values for DIGITS, FUZZ and FORM, in that order, each separated with a single space.

PULL The input string comes from the input console, which is normally the shell window from which the script was run. This is the primary method of inputting information from the keyboard into ARexx programs (see the "PULL" keyword). A prompt is given for strings inputted using PULL as defined by OPTIONS PROMPT. (The default prompt is blank.)

SOURCE Gets information about the script (including how it was invoked), whether a result is expected or not, whether the program was run as a function or a program, what the program name is, and full filepath to the program name of the script. The format of the result is:

```
{COMMAND | FUNCTION} {0 | 1} CALLED RESOLVED EXT HOST
```

A result might look like this:

```
COMMAND 0 shell Development:ARexx/shell.rexx REXX REXX
```

We are first told that this was evoked as a COMMAND rather than a function, and the 0 means that no result is expected from us. We were called as a program name of "shell" and the next string is the full file path to the program. Finally we have the current file extension used for searching for ARexx scripts (normally REXX because ARexx programs use the extension .rexx), and the initial host address for commands (in this case the REXX host).

VALUE expression WITH template The input string comes from the evaluated expression. The WITH is necessary to separate the expression from the template. It is possible to PARSE the result of the expression multiple times using more than one template, although it is only evaluated once.

VAR The input string comes from a variable. Be warned that if multiple templates are used then each one will use the current value for variable – so if the first template affects it then any following templates will use the updated result of the variable.

VERSION Gets information on the machine configuration on which the script was run. The format of the resultant input string before going through optional templates is:

```
ARexx VERSION CPU FPU VIDEO FREQ
```

This might result in a string like this:

```
/* PARSE VERSION example */
PARSE VERSION a_version
SAY a_version

ARexx V1.15 68030 68881 PAL 50HZ
```

The information supplied is: the version of the ARexx interpreter itself; the microprocessor currently in use; the name of the floating point unit in use (or NONE); the video type, PAL or NTSC; and the clock frequency of the current system, usually either 50 or 60Hz.

Don't be alarmed if ARexx reports your 68882 maths co-processor as a 68881, this is because ARexx 1.15 (the current version) cannot differentiate between the two. With some modern processor combinations it can guess the processor type incorrectly as well. Future versions may fix these problems.

A template is made up of a selection of target variables and markers. Markers define the starting position for each of the targets. Markers can either be absolute, relative or pattern-based. Absolute markers are fixed positions for a target set by the number of characters from the start of the input string. Absolute markers are prefixed by an = operator. The = is optional if the absolute marker is not a variable but a fixed value. By prefixing an absolute marker value with a + or a - sign instead of an =, it becomes a relative marker and specifies a positive or negative offset from the current marker position specified in characters. A pattern marker is an actual string, is enclosed in quotes, and is matched in the input string to determine where the target comes from. All markers can be either fixed symbols, such as "45" or "HELLO" or variables. For pattern markers, the variable name must be enclosed in ()s (round brackets). If multiple templates are specified, they are separated with commas. For example:

```
/* PARSE Templates example 1 */
string = "HELLO WORLD"

offset1 = "LO"
/* A pattern marker */
offset2 = 2
/* An absolute marker */

/* Our string is divided up using the template
specified. Three targets are used, two which we shall
pick using the above markers, while anything remaining
gets thrown into the variable "rest" which we will be
ignoring */
PARSE VAR string (offset1) result1 =offset2 result2
rest
/* Note the use of the hexadecimal character 0A
inserted, which is the ASCII code for LINEFEED and puts
our results on separate lines for us (see "Advanced
Programming" in Section B) */
SAY result1 "0a"x result2

EXIT

/* PARSE example 2 */

/* Set the prompt for any PARSE PULL"s */
OPTIONS PROMPT "Enter a time in the format HH:MM:SS>"

/* System version */
```

```
PARSE VERSION version_string
SAY version_string

/* Details about this script */
PARSE SOURCE source_string
SAY source_string

/* Fetch a time of day and split it into HRS MINS and
SECS using a template */
PARSE UPPER PULL string
PARSE VAR string hours ":" mins ":" secs

/* Show the result */
SAY "Hours =" hours " Minutes = " mins " Seconds = "
secs

EXIT
```

PROCEDURE

Syntax:

```
PROCEDURE [EXPOSE variable [variable ...]]
```

Used to define an internal function as having a separate symbol table, thus avoiding variable collisions with the main program.

Normally internal functions share the same variables as the program with which they were called. By using the PROCEDURE statement we cause that function to have its own variables created. These variables are totally separate from the caller's. Even if they happened to share the same name with the caller's variables, they would not affect each other. Variables defined in a PROCEDURE are therefore local variables. Under some circumstances it might be desirable for a function defined as a PROCEDURE to have access to some global variables, defined by the caller. In this case EXPOSE is used. Any variables specified after an EXPOSE keyword are taken from the caller's variable table rather than being created as local variables. This provides an additional way of getting information in and out of a function.

If a stem symbol is defined using EXPOSE, all compound symbols possible from that stem are exposed.

Although the PROCEDURE statement is valid anywhere inside a function, it is normal to place it immediately after the function's label. You cannot use PROCEDURE other than inside an internal function and it is not valid more than once in any single function. For example:

```
/* PROCEDURE example */

/* Our copy of input_value will remain untouched by the
function because it will create a new local variable */
input_value = 99
SAY DivideByTwo(10)
SAY "Callers input_value is" input_value

EXIT

/* A simple function that takes one argument, divides it
by two and returns the result */
DivideByTwo: PROCEDURE
  ARG input_value
  RETURN input_value/2
```

PULL

Syntax:

```
PULL [template] [,template...]
```

Reads a string from the standard input stream (STDIN) and assigns it to variables according to the template specified.

This command is a shorthand version of "PARSE UPPER PULL". In its simplest form PARSE can be used to input a value from the console and assign it to a variable. By using templates it is possible to PARSE the string input in a number of ways (see PARSE for further information). All results given using PULL are in upper case. To get an accurate copy of the inputted string, including upper and lower case characters, use the longer PARSE PULL version instead. For example:

```
/* PULL example */

OPTIONS PROMPT "Yes?"
PULL string
SAY "You entered "string
```

PUSH

Syntax:

```
PUSH [expression]
```

Evaluates the expression and pushes it into the input stream.

PUSH works almost as the opposite of PULL. Instead of fetching data from the input stream (STDIN), it pushes data back in to be read by later accesses to STDIN – such as a PULL instruction. PUSH works like a stack in that the first line PUSH'd in is the last one PULL'd out. PUSH'd information acts as though it was entered by the user from the keyboard so it can be used to pre-enter strings to be read by ARexx or the shell window. For example:

```
/* PUSH Example */
```

```
PUSH "dir"
PUSH "cd ram:"
PUSH "dir"
PUSH "cd sys:"
```

This shows that PUSH'd information can be read by anything which uses the standard input stream (STDIN) for input, such as the shell. Since strings are in a first-in-last-out stack, the last one we PUSH'd (cd sys:) will be the first to be run.

PUSH does not work under Workbench 1.3 unless you have a special console handler. The authors of ARexx (Wishful Thinking Development Corp.) have an excellent shareware console manager called "ConMan". This should be available from all good PD software libraries and on large bulletin boards. Wishful Thinking's commercial shell replacement, called WShell, will allow you to use PUSH under 1.3.

QUEUE

Syntax:

```
QUEUE [expression]
```

Evaluates the expression and QUEUEs it into the input stream.

QUEUE works almost the same way as PUSH except that the strings put back into the input stream are in a queue instead of a stack. This means that the first string QUEUE'd is the first one pulled back out again, when read from the standard input stream (STDIN). Strings QUEUE'd are indistinguishable from those entered interactively by the user from the keyboard and thus can be used to prepare lines to be read from ARexx using PULL, or from the shell. For example:

```
/* QUEUE Example */

QUEUE "This will be entered first"
QUEUE "And this will be in the middle"
QUEUE "And this last"

PULL first
PULL second
PULL third

SAY first "0a"x||second "0a"x||third "0a"x
```

This shows QUEUE being used to prepare strings to be read from PULL statements. Note the use of hexadecimal ASCII codes to give new lines and the string concatenation operator "||" to avoid spaces where we don't want them (see Advanced Programming in Section B for more information).

Like PUSH, QUEUE does not work under Workbench 1.3 unless you have a special console handler. The authors of ARexx (Wishful Thinking Development Corp.) have an excellent shareware console manager called "ConMan". This should be available from all good PD software libraries and on large bulletin boards. Wishful Thinking's commercial shell replacement, called WShell, will allow you to use QUEUE under 1.3.

RETURN

Syntax:

```
RETURN [expression]
```

Leaves the current function and RETURNs the evaluated expression to the function caller.

RETURN is used to exit from a function. If no RETURN value is supplied, the caller might object if he or she was expecting one. If RETURN is used from within the main body of a script and not in a function, it acts as if the EXIT statement was used and stops the script. A script can find out if a RETURN value is expected from it by using PARSE SOURCE.

It is not possible to know whether the caller will make use of the value RETURN'd or not. For example:

```
/* RETURN example */

SAY TimeFunc()

/* Since we're now at the base level this next return
will act like EXIT */
RETURN

SAY "This will never be run"

/* Simple function to return the time of day. Returns
the value RETURNED by the internal TIME function! */
TimeFunc: PROCEDURE
  RETURN TIME("C")
```

SAY

Syntax:

```
SAY [expression]
```

Evaluates the result and displays the output to the current output stream (STDIO) – normally the shell window.

SAY is the primary way of showing output to the screen in ARexx. The string shown automatically has a new-line character appended to the end, so multiple SAYs will always happen on separate lines on the display. By inserting ASCII characters it is possible to do some basic string formatting, including moving the cursor around and inserting new lines in the middle of SAY'd results. Since a space anywhere is shown as a space on the screen, it is often useful to be able to concatenate two strings without any spaces. This can be done with the concatenation operator "| |". For example:

```
/* SAY example */

Hello = "Howdy"

SAY " Hello " || Hello   ||  " Hello " Hello ¬
"0a"x || "On another line!"
```

SELECT

Syntax:

```
SELECT
```

Starts a SELECT sequence, consisting of one or more WHEN conditional statements and optionally one OTHERWISE.

SELECT is used to make multiple conditional checks and then act upon them. Each conditional check is done with a WHEN statement. If none of the conditions succeed the instructions following the OTHERWISE statement are executed. If no OTHERWISE is present, an error is generated. A SELECT sequence is terminated by an END statement. Only one of the WHENs, or the OTHERWISE conditional code will be executed. See both OTHERWISE and WHEN for further information. For example

```
/* SELECT example */
```

```
SAY "Enter a number between 1 and 5"
PULL number

SELECT
  WHEN number = 1 THEN SAY "You entered 1"
  WHEN number = 2 THEN SAY "You entered 2"
  OTHERWISE SAY "You entered something other than 1 ¬
or 2"
  END
```

SHELL

Syntax:

```
SHELL [symbol | string] | [[VALUE] [expression]]
```

Specify the host address for commands which are not ARexx keywords.

A direct equivalent to the ADDRESS instruction. See "ADDRESS" for a full description.

SIGNAL

Syntax:

```
SIGNAL {ON | OFF} condition
SIGNAL [VALUE] expression
```

Controls the state of the internal interrupt flags, or transfers program control directly to a named label.

SIGNAL is used to control the internal interrupt flags. When certain events occur, it is possible to cause an interrupt and to transfer program control to another place in the program for it to be dealt with. There are a number of these conditions and, when one occurs, ARexx jumps directly to a label with the same name as the interrupt itself – as long as you have switched that signal on using SIGNAL ON interrupt_name. SIGNAL can also be used as a unconditional transfer of control statement to jump directly to another position in the program. However, using it in this way is bad programming practice (see Section B for a discussion), but there are circumstances where it could be useful – such as recovering after an error. See the example.

VALID INTERRUPT NAMES

BREAK_C	A Control-C sequence was detected in the STDIN stream. This is usually used to stop execution of a program script with the message "Execution Halted".
BREAK_D	A Control-D sequence was detected.
BREAK_E	A Control-E sequence was detected.
BREAK_F	A Control-F sequence was detected.

(BREAKs D,E and F differ from BREAK_C in that they do not default to anything; that is default produces no result unless you first switch them on and process them yourself.)

ERROR	A host command returned a non-zero return code. If FAILAT is set this can be used to trap all errors up to but not including the FAILAT threshold. If it is not set, then this can be used to deal with all non-zero host command return values. The RC variable contains the error code level.
HALT	An external HALT request was detected. This is useful to prevent being shut down by the utility program HI. Although this can be used in this way it is best used only to clean up neatly, and close any

	resources that you might have allocated before exiting.
IOERR	Traps errors from the Input/Output system, such as attempting to write to a read-only device like a CD-ROM unit. After an IOERR, the RC variable contains an error number.
NOVALUE	Happens when an uninitialised variable is used. Normally, ARexx does not object to uninitialised variables, it simply creates them as you use them and assigns to the variable the variable name itself but in capital letters. This is not the case with many other computer languages where using an uninitialised variable is illegal. By setting this interrupt, any attempt to use an uninitialised variable will result in a trap.
SYNTAX	A syntax or execution error was detected. This includes malformation of program lines, errors like "Unbalanced Parentheses", and execution errors like running out of memory. After the interrupt occurs, RC contains the actual syntax error number which can be converted into a string using the built in ERRORTEXT() function.

To make any of the above interrupts work, they must be first switched on using SIGNAL ON interrupt_name, for example: SIGNAL ON NOVALUE

If the interrupt then occurs, program control will immediately be transferred to the label NOVALUE:. The label name must be defined or a syntax error will occur. The label called when an interrupt occurs is the same as the interrupt name itself. After an interrupt the special variable SIGL contains the line number where the interrupt occurred.

Signals are particularly useful for trapping errors and dealing with them yourself. This can be very important in a large ARexx application when a sudden failure would look unprofessional and confuse the user. Often you may not necessarily want errors to be shown on the shell. If your ARexx script was talking to a text editor you might prefer to have the errors popping up in a window on the Editor's screen. SIGNAL can allow you to do this and customise the way you deal with the error. For example:

```
/* Signal Example */

reset:

SIGNAL ON syntax

hello = ((((1 + 2)

EXIT

/* Deal with Syntax Error */
syntax:
  SAY "A syntax error occurred. Error ID" RC ","
ERRORTEXT(RC)
  SAY "Type Y and press return to continue, any other
key to halt program"
  PULL check

  /* Re-Starts program if we typed Y */
  IF UPPER(check) == "Y" THEN SIGNAL reset

  EXIT
```

This example sets up a signal to trap syntax errors. Straight after we switch on the interrupt, we cause an "Unbalanced Parentheses" syntax error. This is then trapped by our "syntax:" label, where we show the error number and description text before asking the user if they wish to re-start the program and try again, or exit. An unconditional branch then happens to the "reset" label at the start of the program. Of course, if you do type Y and re-start this example, the syntax error will simply re-occur.

TRACE

Syntax:

```
TRACE   [mode] | [[VALUE] expression] | [num]
```

Sets the trace mode. Tracing is a debugging facility where the interpreter allows you to see information about the way the program is progressing. ARexx also offers interactive tracing where you can single-step through the program and see what is going on as you work your way through. While interactive tracing is on, any further changes to the trace mode using the TRACE statement are ignored, although the built-in TRACE() function will still allow you to alter tracing mode (see Function reference). See Section B for a full description of tracing.

	AVAILABLE TRACE MODES
ALL	All Clauses are traced
BACKGROUND	The program runs with no tracing information, and cannot be forced into interactive tracing using the TS/TCO command utilities.
COMMANDS	Command clauses are traced. Command clauses are clauses which are sent to an external host for execution. Non-Zero return codes are displayed on the console. (Normally your shell window or the global trace console)

ERRORS	All commands that generate a non-zero return code are traced after the clause has been executed.
INTERMEDIATES	All clauses are traced and intermediate results are displayed during the evaluation of expressions. This is a lot of information and generates a considerable quantity of trace information. This data includes values of variables.
LABELS	Only label clauses are traced. Labels are specific points in the program which control can be directly transferred to. They are traced after such a transfer takes place.
NORMAL	This is the default tracing mode, and nothing is traced unless an error occurs in which case the failed clause is shown with an error message.
OFF	Tracing is switched off.
RESULTS	A very useful form of tracing. All clauses are traced before execution and the final result of every expression is displayed. This is one of the best tracing operations because it shows the values assigned to variables from PARSE and other such statements.
SCAN	All clauses are traced and checked for errors, but nothing is executed. This is a program dry-run facility.

All of these can be abbreviated to one character. If the new tracing mode is preceded by a ? then it will toggle backwards and forwards to the interactive tracing mode. Note that with interactive tracing on, the TRACE statement will be ignored, so once on – it's on!

Interactive tracing allows you to make the program stop at certain points and provide a prompt so that control commands can be inputted. Pressing Return at this point will proceed to the next stop point, pressing "=" and then return re-executes the current statement.

The TRACE statement also accepts a numeric parameter. This is useful when you are in interactive tracing and wish to suspend tracing for a period of time, or allow more than one instruction to pass before execution stops and waits for you to respond. If the numeric parameter, N, is positive it takes this to mean: "Continue tracing for N clauses before stopping again". If negative, it takes it to mean: ""Don't do any tracing for N clauses".

While in interactive tracing mode it is possible to examine the contents of variables and execute program statements by simply typing them in. The command utilities TS, TE, TCO and TCC (see utility command reference below) are also related to the control of tracing programs and allow you to trace and stop "runaway" programs.

WHEN

Syntax:

```
WHEN expression [THEN [;] [conditional statement]]
```

Used like IF to execute conditional statements, but from within a SELECT..END structure.

See SELECT for further information. WHEN behaves like an IF. The expression is evaluated and, if TRUE, the conditional statement is executed. You can execute more than one statement if the condition is true by enclosing them in a DO...END. Once one of the WHEN statements has been successfully evaluated to TRUE and executed, control is passed directly to the instruction after the SELECTs matching END. Only one of the WHENs will ever be executed. If none evaluate to TRUE, ARexx looks for the OTHERWISE and executes any conditional statement there. For example:

```
/* WHEN example */

SAY "Enter a number"
PULL number

SELECT
  WHEN number < 100 THEN SAY "Smaller than 100"
  WHEN number // 2 THEN SAY "Odd"
  OTHERWISE SAY "I didn't catch that"
  END

EXIT
```

This demonstrates that only one of the WHENs is run. If you type in a number that is both less than 100 and odd – 17 for example – you would expect both WHENs to be run. Only the first WHEN, for which the expression evaluates to TRUE, occurs.

Function Reference

The function reference section describes the usage of all the built-in functions and the additional functions made available when using the rexxsupport.library. Section B has a full description of how to use functions, with plenty of examples, and also describes how to load additional support libraries.

Built-in Functions

ARexx has nearly 90 built-in functions which you can use to find, or process, information in some way or other. This reference lists them all in alphabetical order, together with an example of what the function does. Built-in functions are not case sensitive; they can be in lower or upper case characters. We use upper case here to help differentiate them from our own functions and variables used in this book. In addition to the function names themselves not being case sensitive, some functions have optional keywords to tell them how to behave. These are also case insensitive. Instead of full, working-program examples, only one instance of a function being used, and the result, are shown. In some cases, the examples are shown as if they had been typed into the ARexx shell (listed in "Advanced Programming Techniques" in Section B). In this case the command we typed in is shown along with the result given, on a separate line. For example:

```
>SAY "Hello World!"
Hello World
```

For each function, the function name, the syntax, a description and finally an example are given. The syntax line clearly shows how the function might be used, with all of its parameters and what type of value is returned. As for the keyword reference, optional parameters are given in square brackets – "[" and "]". Valid return value types include:

`Boolean`	`0 or 1, TRUE or FALSE.`
`string`	`A string value is returned.`
`number`	`Numerical value`
`result`	`The result of the specified operation is returned.`

ABBREV()

Syntax:

```
Boolean = ABBREV(string1, string2 [, length])
```

Returns a Boolean TRUE if string 2 is an abbreviation of string 1. The optional length parameter specifies how many characters long string 2 must be before the function will check it against string 1. The default for length is 0 so, if you do not specify it, then a null string for string 2 would match as an abbreviation of string 1. This function is particularly useful when you have a menu of options for the user to choose from and you want to allow abbreviations to be valid. For example:

```
 /* Abbrev example */

SAY "Menu:"
SAY "   HELLO   Say hello"
SAY "   GOODBYE Say goodbye"

PULL choice

SELECT
  WHEN ABBREV("HELLO", choice, 2) THEN SAY "Hello"
  WHEN ABBREV("GOODBYE", choice, 2) THEN SAY "Goodbye"
  OTHERWISE SAY "Unknown command"
  END
```

```
EXIT
```

In this example we specify the optional length parameter, meaning that we have to enter at least two characters of the command before it can be recognised.

ABS()

Syntax:

```
number = ABS(number)
```

Returns the absolute value of the single argument. This function effectively strips off the minus sign if present and returns a positive value every time, for example:

```
>SAY ABS(-1.234)
1.234
>SAY ABS(9876)
9876
```

ADDLIB()

Syntax:

```
Boolean = ADDLIB(name, priority [, offset, version])
```

Adds a function library or host to the current ARexx library list and makes it available to ARexx programs (see Section B for further information on function hosts, and the RXLIB command in "Utility Programs Reference" below). To use ADDLIB, several parameters must be specified. These are the name of the library itself, the priority that it will have in the search list and, optionally, an offset to the library's query point and a minimum version for the library.

Priorities can be from -100 to 100. 0 is the usual value. If you had two libraries which had functions with identical names, then you could specify which one would be used by giving that library a higher priority.

If this function succeeds, it returns a Boolean TRUE. If the function fails, because the library did not have a new enough version number,or it could simply not be loaded or found, FALSE is returned. For example:

```
/* Add the rexxsupport.library to the library list */
IF (ADDLIB("rexxsupport.library", 0, -30, 0) = 0) THEN
  SAY "I could not add the rexxsupport library"
ELSE
  SAY "Rexxsupport.library added"
```

ADDRESS()

Syntax:

```
string = ADDRESS()
```

This returns the name of the current host address, which is where host commands are sent. This can be particularly useful, because a program is able to find out which address is currently selected and is then able to return to that address after changing it. For example:

```
/* ADDRESS() example */

old_address = ADDRESS()
SAY "Old address = " old_address

ADDRESS "ProWrite"
SAY "Current address = " ADDRESS()

/* In the ProWrite Word Processor, this sets the style
```

```
to bold */
StyleBold

ADDRESS VALUE old_address
SAY "And now, address = " ADDRESS()

EXIT
```

This example remembers the current address, then changes it, using the ADDRESS keyword, issues a host command, and then sets it back to the previous one.

ARG()

Syntax:

```
result = ARG([number][,"EXISTS"|"OMITTED"])
```

The ARG function can be used to find out information about the arguments supplied to the current function or script. It can be used in three basic forms:

1 With no parameters at all. This will return the number of arguments present as a positive integer.

2 With a single numeric parameter specified for "number". This will return that argument, if present. If it is not present, a null string is returned.

3 With an argument number specified and either the EXISTS or OMITTED keyword. This allows you to find out the status of a particular argument. When used in this fashion, ARG() returns a Boolean result, depending on whether the named argument EXISTS or has been OMITTED. The keywords can be abbreviated and are not case sensitive.

Examples:

```
IF ARG(1, "OMITTED") THEN SAY "Argument 1 was omitted"
SAY ARG(2)
/* Shows the 2nd argument on the screen */
SAY ARG()
/* Shows the total number of arguments present. */
```

B2C()

Syntax:

```
result = B2C(string)
```

Converts a binary string to characters. Spaces are permitted in the string of binary digits only at byte boundaries to make it easier to read. For example:

```
>SAY B2C("1000010")
B
```

BITAND()

Syntax:

```
result = BITAND(string1, string2 [,pad])
```

Logical ANDs string 1 and string2 (see end of Section B for more information on logical operations such as AND). The optional pad character is used to pad the shorter of the two strings up to the length of the longer one. If no pad character is provided, the OR operation terminates at the end of the smallest string and the remainder of the longer one is appended to the result. For example:

```
>SAY C2X(BITAND("123456"x, "ff00"x, "ff"x))
120056
>SAY C2X(BITAND("123456"x, "f0f0f0"x))
103050
```

BITCHG()

Syntax:

```
result = BITCHG(string, bit)
```

Changes the state of the specified bit in string. If it was a binary 0 then it becomes 1 and vice versa. Bit 0 is the least significant bit, the one on the far right of the first argument. For example:

```
>SAY BITCHG("01100001"b,5)
A
>SAY BITCHG("a", 5)
A
```

BITCLR()

Syntax:

```
result = BITCLR(string, bit)
```

Clears the specified bit in string. Bit 0 is the least significant bit, the one on the far right of the first argument. For example, this clears bit 0, the bit on the far right, turning the supplied binary number from 67 to 66. It produces the result "B" on the screen, the ASCII character 66:

```
>SAY BITCLR("01000011"b,0)
B
```

For another example see the BITSET() function example below.

BITCOMP()

Syntax:

```
result = BITCOMP(string1, string2 [,pad])
```

Does a bit comparison of the two arguments, starting at bit 0 (the bit at the far right of the number and the least significant). The returned value is the bit number at which the two strings first differed, or -1 if they were identical. The optional pad character is used to pad the smaller string to the same length as the larger one. If no pad character is provided, the default is a space. For example:

```
>SAY BITCOMP("89123456"x, "123456"x, "99"x)
28
>SAY BITCOMP("123456"x, "3456"x)
17
>SAY BITCOMP("  3456"x, "3456"x)
-1
```

BITOR()

Syntax:

```
result = BITOR(string1, string2 [,pad])
```

Logical ORs string 1 and string2 (see end of Section B for more information on logical operations such as OR). The optional pad character is used to pad the shorter of the two strings up to the length of the longer one. If no pad character is provided, the OR operation terminates at the end of the smallest string and the remainder of the longer one is appended to the result. For example:

```
>SAY C2X(BITOR("123456"x, "ff00"x, "ff"x))
FF34FF
>SAY C2X(BITOR("123456"x, "f0f0f0"x))
F2F4F6
```

BITSET()

Syntax:

```
result = BITSET(string, bit)
```

Sets the specified bit in string to 1. Bit 0 is the least significant bit, the one on the far right of the first argument. For example, this will set bit 0 to 1. Bit 0 is the units column, so this will have the effect of making even numbers odd, but not changing numbers which are already odd:

```
/* BITSET() example */

SAY "Enter a number"

PULL number
number = BITSET(number,0)
SAY "Number is " number

EXIT
```

By replacing the BITSET in the above example to BITCLR, you can produce the opposite effect, turning odd numbers into even, but not affecting even ones.

BITTST()

Syntax:

```
Boolean = BITTST(string, bit)
```

Returns TRUE if the specified bit in string is set to 1, or FALSE if it is set to 0. Bit 0 is the least significant bit, the one on the far right of the argument "string"'. An example of its use is a modification of the example program for BITSET, which shows whether a number entered is even or odd:

```
/* BITTST() example */

SAY "Enter a number"

PULL number
IF BITTST(number, 0) = 1 THEN
  SAY "Odd"
ELSE
  SAY "Even"

EXIT
```

BITXOR()

Syntax:

```
result = BITXOR(string1, string2 [,pad])
```

Logical XORs (exclusive OR) string 1 and string2 (see end of Section B for more information on logical operations such as XOR). The optional pad character is used to pad the shorter of the two strings up to the length of the longer one. If no pad character is provided, the OR operation terminates at the end of the smallest string and the remainder of the longer one is appended to the result. For example:

```
>SAY C2X(BITXOR("123456"x, "ff00"x, "ff"x))
ED34A9
>SAY C2X(BITXOR("123456"x, "f0f0f0"x))
E2C4A6
```

C2B()

Syntax:

```
result = C2B(string)
```

String to binary conversion. Converts the characters in "string" to their equivalent binary bytes. The opposite of this function is B2C(). For example:

```
>SAY C2B("A")
01000001
```

C2D()

Syntax:

```
result = C2D(string [,n])
```

String to decimal conversion. Converts the argument into its corresponding decimal number, assuming each character in the string is a one byte value. Can be used to find the ASCII value of a single character:

```
>SAY C2D("A")
65
```

The optional n parameter is the amount of bytes from the left hand side of the string which C2D will use in its conversion. For example:

```
SAY C2D("ABCD",1)
68
```

C2X()

Syntax:

```
result = C2X(string)
```

String to hexadecimal conversion. Converts the argument into its corresponding hexadecimal number, assuming each character in the string is a one byte value. Can be used to find hex ASCII values of single characters.

```
SAY C2X("ABCD")
```

The above example shows 41424344 on the screen. Each two-number pair is the HEX ASCII for the corresponding character, 41 is "A", 42 is "B" and so on.

CENTRE()

Syntax:

```
result = CENTRE(string, length [,pad])
```

Centres the string argument in a string of length specified by the length argument. The optional pad character is used instead of the default SPACE to pad out each end of the string to centre it. Please note –both spellings are provided for, American and British! For example:

```
SAY CENTER"Introduction", 40, "-")
```

... will centre the word "Introduction" in a line 40 characters in length. The ends are padded using a "-" character producing a result like this:

```
——————Introduction——————
```

CLOSE()

Syntax:

```
success = CLOSE(file)
```

Closes a file previously opened using the OPEN() function. If the operation fails for some reason (the file didn't exist, for example), then this function will return FALSE. If the file was successfully closed, it returns TRUE:

```
SAY CLOSE("MyFile")
/* Will show 1 if "MyFile" was successfully closed */
```

See the OPEN() function for a further example of using CLOSE().

COMPARE()

Syntax:

```
result = COMPARE(string1, string2 [,pad])
```

Compares the two strings supplied and returns the offset, in characters, from the left-hand side, where the strings first differ – or returns 0 if the strings are identical. If the optional pad character parameter is given the shorter string is padded out the length of the longer one with the specified character, or blanks are used if none was specified. Pad characters are added to the right-hand side of the shorter string. For example:

```
SAY COMPARE("tobyAAA", "toby", "B")
/* Produces the result 5 */

/* Result of 0, strings identical. SHorter is padded
using As making it identical to the longer one: */
SAY COMPARE("tobyAAA", "toby", "A")

SAY COMPARE("the quick brown fox", "the quick green
fox")
/* result is 11 */
```

COMPRESS()

Syntax:

```
string = COMPRESS(string [,removelist])
```

COMPRESS() removes any of the characters specified in the optional parameter from the string specified. The default for the removal list, if not provided, is SPACE – which will remove any spaces from the string. Can be particularly useful for removing special characters and punctuation from text before performing text-based operations, such as word counting. Here is an example which strips vowels from strings entered by the user:

```
/* COMPRESS() example */

SAY "Enter a string, and I'll take the vowels out of
it"
PARSE PULL string

string1 = COMPRESS(string)
string2 = COMPRESS(string, "aeiou")

SAY "Output of Compress with no removal parameter:"
string1
SAY "Vowel-less version!:" string2

EXIT
```

COPIES()

Syntax:

```
result = COPIES(string, copytimes)
```

Makes a number of copies of the supplied string, by appending the string to itself the number of times specified by the "copytimes" numeric parameter. A value of 0 for this is valid, and COPIES() will return an empty string.

```
>SAY COPIES("Hi ", 3)
HiHiHi
```

D2C()

Syntax:

```
string = D2C(number)
```

Converts the specified number into a character string. This routine is particularly useful for converting single ASCII code values into their appropriate number. Works a byte at a time through the number in the conversion process. The parameter must be a positive integer from 0 to 2,147,483,647 (7FFFFFFF in hexadecimal).

```
>SAY D2C(65)
A
```

D2X()

Syntax:

```
string = D2X(number)
```

Decimal to hexadecimal conversion. Converts the specified number into a hexadecimal string. The parameter must be a positive integer from 0 to 2,147,483,647 (7FFFFFFF in hexadecimal).

```
>SAY D2C(65535)
FFFF
```

DATATYPE()

Syntax:

```
result = DATATYPE(string)
```

Or:

```
Boolean = DATATYPE(string, type)
```

Determines or checks for the datatype of a particular string. In its first mode of operation the function attempts to identify what datatype has been supplied and returns either NUM or CHAR, depending on whether the string is a valid number or consists of characters.

The second mode of operation allows checking of a string to see if it is a particular datatype. It will return a Boolean value, TRUE, if the string did consist of the datatype required, or FALSE if it did not. This can be used to check values entered by the user before processing them to avoid errors during the running of a program.

	VALID DATATYPES FOR STRINGS
ALPHANUMERIC	String is alphanumeric characters, A-Z, a-z or 0-9.
BINARY	String is a valid binary number, consisting of 0s and 1s.
LOWERCASE	String consists of lower case alphabetic characters only.
MIXED	String consists of any alphabetic characters, upper or lower case.
NUMERIC	String is a valid decimal number.
SYMBOL	String is a valid AREXX symbol.

UPPER	String consists of upper case characters only.
WHOLE	String is a valid integer number, positive or negative.
X	String is a valid hexadecimal number.

... and the DATATYPE() function will return TRUE if the appropriate condition specified is satisfied. All of the above options can be abbreviated to one character. For example:

```
SAY DATATYPE("Hello")
/* Shows CHAR */
SAY DATATYPE("0101Z", "BINARY")
/* Shows 0, as the Z is not a valid binary character */
```

For a further example of DATATYPE() in action, see the description for the SOURCELINE() function below.

DATE()

Syntax:

```
result = DATE([option] [,date] [,format])
```

In its default form, with no parameters, the option is assumed to be "NORMAL" and this function returns the current date in the form DD MMM YYYY – for example, 10 Nov 1993.

	VALID OPTIONS FOR DISPLAYING DATES
BASEDATE	The number of days passed since 1 Jan, 0001.
CENTURY	Days passed since 1 Jan, this century.
DAYS	Days passed since 1 Jan, the current year.

EUROPEAN	Date in the form of DD/MM/YY – for instance, 23/01/94
INTERNAL	Internal system days - for example,days since 1 Jan, 1978.
JULIAN	Date in the Julian Date (year and days elapsed this year) format, YYDDD., eg 94/123
MONTH	Current month (in mixed case – for instance, "January")
NORMAL	Date in the AmigaDOS format DD MMM YYYY – for instance, 10 Jan 1994
ORDERED	The date in the format YY/MM/DD – for instance, 94/01/23
SORTED	The date in the format YYYYMMDD – for instance, 19940123
USA	The date in the American format MM/DD/YY – for instance, 01/23/94
WEEKDAY	The day of the week, in mixed case – for instance, "Monday"

These options can be abbreviated to one character.

The second and third parameters allow you to find information about a specified date. For example, it is possible to find the day of the week for any date you like, so long as it is after 1 Jan 1978. The second parameter is the date itself, in one of two formats: system days (the default); or sorted date. For system days, the number of days since 1 Jan 1978 is specified. For sorted, the date is specified in the format YYYYMMDDD (the "SORTED" option above). Here is an example:

```
SAY DATE("W",19780101,"S")
/* Shows "Sunday", the day of 1st Jan, 1978, start of
internal time */
SAY DATE("MON")
/* Shows the current Month */
```

DELSTR()

Syntax:

```
result = DELSTR(string, n [,length])
```

Deletes the sub-string beginning with the "n"th character in from the left hand side of the string, and length characters in length from the supplied string. The default value for n is the remaining length of the string and can be therefore be used to abbreviate strings:

```
SAY DELSTR("Hello There", 3)
/* Shows "He" */
```

When used with the optional length parameter, selective deleting can take place:

```
SAY DELSTR("Hello There", 3, 2)
/* Shows "Heo There" */
```

DELWORD()

Syntax:

```
result = DELWORD(string, n [,length])
```

Deletes "length" number of words from the supplied string beginning with the "n"th word in from the left. The default value for length is the remaining number of words in the string, and can thus be used to shorten strings to a fixed number of words. White space characters, such as SPACE and TAB separate words. An example:

```
/* Produces the result "This was amazing": */
SAY DELWORD("This is, or was amazing", 2, 2)

/* Deletes the third word, producing the result "first
second" */
SAY DELWORD("first second third", 3)
```

DIGITS()

Syntax:

```
numeric = DIGITS()
```

Returns the current DIGITS setting, as set by the ARexx statement NUMERIC DIGITS. DIGITS is the number of digits of precision used for arithmetic operations. Useful for recalling what the current setting is so that it can be changed back after a temporary alteration.

```
old_digits = DIGITS()
/* Stores the current DIGITS() setting in the
old_digits variable */
```

EOF()

Syntax:

```
Boolean = EOF(file_handle)
```

Returns TRUE if the End of File marker has been reached for the specified open file, otherwise returns FALSE. If reading in a file of unknown length for processing, such as a text file for a word count, for instance, this function can be used to determine when there is no more data to be read. For example:

```
IF EOF("MyFile") THEN SAY "End of file reached"
```

ERRORTEXT()

Syntax:

```
result = ERRORTEXT(error_number)
```

Returns the actual error message associated with the error_number specified. See Appendix 1 for a complete list of ARexx error numbers and messages.

EXISTS()

Syntax:

```
Boolean = EXISTS(filename)
```

Returns TRUE if the specified file exists, or FALSE if it does not. Complete paths can be specified allowing programs to determine if a file about to be used is available. For example:

```
/* This example runs the text editor in your C: drawer
if it exists */
IF EXISTS("C:ED") THEN DO
  ADDRESS COMMAND ED
  SAY "Editor run"
  EXIT
```

```
SAY "Could not run editor"
EXIT
```

EXPORT()

Syntax:

```
result = EXPORT(address [,string] [,length] [,pad])
```

Copies the data from the optional string parameter to an area of memory specified with the address parameter. The address value normally comes from the GETSPACE() function, used to allocate memory for your own usage. The length parameter, if specified, defines the number of characters to be copied. The default value for length is the actual length of the string. If the specified length is longer than the actual string itself, then the remaining area is padded out with the pad value, or the byte value 0 if no pad is specified. The returned value is the number of bytes actually copied. For example:

```
bytes_copied = EXPORT("00500000"x, "SomeText")
```

WARNING **This can be a very dangerous function. You can crash your computer, or worse, corrupt data on your hard disk if you don't use it properly. While EXPORTing is taking place, ARexx will disable multi-tasking. If you EXPORT large quantities of data, system performance might be adversely affected.**

FIND()

Syntax:

```
result = FIND(string, phrase)
```

FIND() attempts to find a smaller phrase of words in a larger string, and returns the word number at which the phrase occurred in string, or 0 if it could not find it. This function does not detect parts of words in strings, phrase is assumed to be a number of words with white spaces between them. For example:

```
>SAY FIND("This is a sentence", "sente")
0
>SAY FIND("This is a sentence", "sentence")
4
```

FORM()

Syntax:

```
result = FORM()
```

Returns the current NUMERIC FORM setting, either ENGINEERING or SCIENTIFIC, depending on the value set. The NUMERIC FORM setting specifies the format in which numbers requiring exponential notation are shown. This example shows the current setting:

```
>SAY "Current setting of NUMERIC FORM is "FORM()
Current setting of NUMERIC FORM is SCIENTIFIC
```

FREESPACE()

Syntax:

```
  Boolean = FREESPACE(address, length)
  result = FREESPACE()
```

Used to return memory allocated using the GETSPACE() function to the interpreter's internal memory pool. The "address" parameter should be the same that was returned from GETSPACE(). It is good practice to free any memory you allocate as soon as you have finished with it, but if you do forget, ARexx will automatically free memory allocated using GETSPACE() when your program exits. When actually freeing memory, this function should return a Boolean success result, but the current release version of ARexx, 1.15 actually returns the amount of memory in ARexx's internal memory pool – a fairly useless number. In the function's second form, with no parameters, the amount of memory in the internal memory pool is returned. Again, this is a reasonably useless value, because the pool simply grows if you attempt to allocate more than it contains – assuming you have the memory available, that is.

```
/* FreeSpace example */
memory = GETSPACE(2000)
SAY FREESPACE()
SAY FREESPACE(memory, 2000)
EXIT
```

The above allocates 2,000 bytes of memory, shows the space left in the pool, and then frees our block of memory showing the new free space in the pool - this example takes into account a "misbehaviour" in ARexx 1.15.

FUZZ()

Syntax:

```
numeric = FUZZ()
```

Returns the current value of NUMERIC FUZZ, which is the number of digits to ignore in numeric comparison operations. It is a positive whole number.

```
SAY FUZZ()
/* Shows the current NUMERIC FUZZ setting */
```

GETCLIP()

Syntax:

```
result = GETCLIP(clip_name)
```

Attempts to find a value in the global clip list by the supplied name, and returns the contents of that clip. The clip list is globally available to all applications and is an easy method of sharing data. SETCLIP() is used to set clip values. The clip_name is case sensitive and, if no clip of that name is found, this function returns an empty string. Issuing RXSET as a command from the shell will list all current clips and their values (see Utility Command Reference, below, for information on the RXSET command).

```
/* GETCLIP() example: */

SAY "Enter a value for the clip:"
PARSE PULL clip

SETCLIP("test_clip", clip)

SAY "Clip value is now: " GETCLIP("test_clip")

EXIT
```

Typing "RXSET" from the shell, after running the above example, will show the name "test_clip" and the value you typed in assigned to it.

GETSPACE()

Syntax:

```
result = GETSPACE(length)
```

Allocates a block of memory, "length" bytes long, and returns the address of that memory – or NULL if the memory allocation failed. There is no guarantee as to the location of the memory, it could be in either fast or chip memory, and it is not initialised – so it is likely to contain random data. Memory allocated with GETSPACE() is automatically freed when the program exits, but it is good programming practice to free memory when you no longer need it using the FREESPACE() function. For example:

```
/* This allocates 1000 bytes of memory, and shows the
address of this memory in hex */
SAY C2X(GETSPACE(1000))
```

HASH()

Syntax:

```
numeric = HASH(string)
```

Hashing is a process whereby a mathematical calculation takes place on the string, deriving a numeric value which is (in ideal cases) different for every string, and can therefore be used to quickly find items in tables. Complex hashing algorithms are developed to perform this operation to try to avoid "collisions" – where the hash value for two different strings is identical. The ARexx HASH() function is very simple though, and returns an integer value from 0 to 255. This example generates and shows the hash value for a string entered from the console:

```
/* HASH() example: */

SAY "Enter a string:"
PARSE PULL hash
```

```
SAY "Hash value for this string is "HASH(hash)

EXIT
```

IMPORT()

Syntax:

```
result = IMPORT(address [,length])
```

Reads bytes from the specified address. If no length parameter is specified, the reading will take place until a zero byte is found, otherwise the result will be of "length" bytes. This function is used to read values out of memory which has been allocated by the user.

It is possible to crash your computer with this function, as you can read data from memory in special write-only locations. IMPORT is intended for occasions where you have allocated memory using GETSPACE(), or need to read a special system variable such as the ExecBase. For example:

```
/* Show Execbase value */
SAY C2X(IMPORT("00000004", 4))
/* My system showed 07C007F8 */
```

Programmers using a MMU (Memory Management Unit) and development tools like Enforcer, will get enforcer hits from the above example. This is because ARexx reads memory a byte at a time using IMPORT, not four bytes at a time.

INDEX()

Syntax:

```
numeric = INDEX(string, pattern [,start])
```

Searches for the sub-string "pattern" in the larger "string" parameter, and returns an offset, in characters, from the start of the string of the first occurrence. The optional parameter "start" allows the user to specify an initial start position, as an offset in characters at which searching will begin. The default value for start, if not specified, is 1. This function returns 0 if the sub-string did not occur in the main string. It is identical to the POS() function, except the parameter ordering (the "string" and "pattern" parameters) is reversed. For example:

```
SAY INDEX("Contoured with touring", "to")
/* Returns 4, the first occurrence of "to" */
SAY INDEX("Contoured with touring", "to", 10)
/* Start at position 10, so returns 2nd occurrence of
"to", which is 16 */
```

INSERT()

Syntax:

```
result = INSERT(new, old [,start] [,length] [,pad])
```

Inserts a new string into an older one. The optional parameters offer control over where the new string is inserted, how many characters are inserted, and what character to pad the new string with if it is shorter than the supplied "length" parameters. The default pad character is a blank space. If the "start" position is longer than the string, the new string is inserted at the end of the old. The default value for "start" is zero, and the default for "length" is the length of the new string to be inserted. This function, in conjunction with the opposite DELSTR(), and INDEX(), can provide quite complex editing functions including full search and replace facilities.

```
/* INSERT() example: */

SAY "Enter a string:"
PARSE PULL string

offset = 1
expand_to = "expanded"

DO WHILE new_offset > 0

  new_offset = INDEX(string, "z", offset)

  IF new_offset > 0 THEN DO
    SAY "Expanding one reference.."
    string = DELSTR(string, new_offset, 1)
    string = INSERT(expand_to, string, new_offset - 1)
    offset = new_offset + LENGTH(expand_to)
    END

  END

SAY "Result:" string

EXIT
```

When run, it could produce this result:

```
Enter a string:
this is the z string of zness
Expanding one reference..
Expanding one reference..
Result: this is the expanded string of expandedness
```

LASTPOS()

Syntax:

```
numeric = LASTPOS(pattern, string [,start])
```

Searches for the first occurrence of "pattern" in "string", but starts from the end of the string and works backwards – unlike POS() and INDEX() which start at the beginning.

The optional "start" parameter is the character offset to begin the search. The default is the last character in the string. The result is 0 if no match was found, or the character offset from the beginning of the string where the match occurred (assuming 1 is the first character in the string). This is particularly useful for dealing with strings where you might need to find the end section, such as the formatting of times and dates, and stripping the filename off a fully qualified AmigaDOS path. For example:

```
SAY LASTPOS(":", "10:54:23")
/* Shows 6, the furthest right : found */
```

The next example finds the start position of the filename itself. Using the RIGHT or SUBSTR functions you could strip off the file part, or just the path part from a full filename:

```
SAY LASTPOS("ram:directory/another_directory/file.test",
"/")
```

LEFT()

Syntax:

```
result = LEFT(string, length [,pad])
```

Returns a sub string "length" characters long from the left hand side of the parameter "string". If length is larger than string, it is padded with the optional pad character. If no pad character is specified, a blank space is used as default. The opposite of this is the RIGHT() function. For example:

```
>SAY LEFT("two words", 3)
two
```

LENGTH()

Syntax:

```
numeric = LENGTH(string)
```

Returns the length of the string in characters. For example:

```
>SAY LENGTH("Hello")
5
```

LINES()

Syntax:

```
numeric = LINES(file_handle)
```

This function returns the number of lines typed ahead, PUSH'd, or QUEUE'd (see the description of the ARexx QUEUE and PUSH statement for further information), or 0 if we are up to date. For example:

```
/* LINES() function example */

PUSH "echo Hello"
PUSH "echo Goodbye"

SAY LINES(STDIN)

/* A routine to remove everything which has been
```

```
PUSH'd/QUEUE'd or TYPE'd ahead */
DO WHILE LINES(STDIN) > 0
  PULL string
  SAY "I removed "string" from the input stream!"
  END

EXIT
```

TOP TIP **LINES() will not work with the normal Amiga shell, which does not support the information types required for LINES(). From the Amiga shell, this will always return 0, regardless of what has actually been QUEUE'd. Some commercial shells, such as WShell by William Hawes' (the author of ARexx), do work, using new console handlers.**

MAX()

Syntax:

```
numeric = MAX(number, number [,number, ...])
```

Returns the largest of all the numeric parameters supplied. At least two parameters must be specified. For example:

```
SAY MAX(10,20,50,4)
/* Shows 50, the largest of the four parameters */
```

MIN()

Syntax:

```
numeric = MIN(number, number [,number, ...])
```

Returns the smallest of all the numeric parameters supplied. At least two parameters must be specified. For example:

```
SAY MIN(10,20,50,4)
/* Shows 4, the smallest of the four parameters */
```

OPEN()

Syntax:

```
success = OPEN(file, filename [,"APPEND"|"READ"|"WRITE"])
```

Opens a new file for access specified by the optional operation type of APPEND, READ or WRITE. The default, if none is specified, is READ. A file opened in WRITE mode is created as a new file and the opener has exclusive access to the file while it remains open. If a file already exists of that name, then it will be erased and the new file will replace it. READ and APPEND are almost the same, except that in APPEND mode the start position in the file is set to the end, rather than the beginning. READ and APPEND files are opened as existing files, and you can write to them as well as read from them. READ, WRITE and APPEND may be abbreviated to one character. There is no limit to the number of files you can have open at any one time, and they are automatically closed when your program exits. It's good practice, however, to always close your files when you have finished with them using the CLOSE() function, particularly in the case of files you have written to to ensure that the copy on disk is up to date. You can use OPEN() to open console windows.

The opener specifies the name by which this file will be referred to (the File Handle), and the filename itself. The File Handle is our "Magic Cookie" (or special name, in the case below it is "MyFile"), which we specify when using a file command to access that file, so that ARexx knows which file we wish to talk to.

```
/* Open/Close example */
```

```
IF OPEN("MyFile", "ram:test", "W") = 0 THEN DO
  SAY "I could not open my file!"
  EXIT
  END

written = WRITELN("MyFile", "A Grand Piano would be
fine, thanks.")

SAY "I wrote "written" characters out to the file"

IF CLOSE("MyFile") = 0 THEN SAY "File didn't close
right!"

EXIT
```

The above example creates a new file in the RAM disk containing one line, which can be shown from the shell using the Type command:

```
8.System3.1:> rx openclose.rexx
I wrote 37 characters out to the file
8.System3.1:> type ram:test
A Grand Piano would be fine, thanks.
```

Here is an example which opens a window on the screen:

```
SAY OPEN("MyWindow", "CON:0/0/640/200/TobysWindow")
```

Information can be put in the window using the normal file reading and writing functions, such as READLN() and WRITELN(). The window can be closed using the CLOSE() function.

OVERLAY()

Syntax:

```
result = OVERLAY(new, old [,start] [,length] [,pad])
```

Overlays a new string over an existing one.

The optional parameters allow control over where the overlay is going to take place: an offset in characters, "start", the amount of characters to be overlaid, "length", and the pad character to be used to pad out the new string if it is less than the length parameter. The default pad character is a blank space. This function is very useful for direct overwriting one part of a string with another of the appropriate length. For example:

```
/* This example replaces the like in I Like Chocolate
with hate */
SAY OVERLAY("hate", "I like chocolate", 3, 4)
```

POS()

Syntax:

```
numeric = POS(pattern, string [,start])
```

Searches for the sub-string "pattern" in the larger "string" parameter, and returns an offset from the start of the string of the first occurrence, in characters. The optional parameter "start" allows the user to specify an initial start position, defined as an offset in characters. This is the same as the INDEX() function, except that the pattern and string parameters are reversed.

PRAGMA()

Syntax:

```
result = PRAGMA(option [,value])
```

PRAGMA() is a special function designed to allow the user some control over the system environment in which their script is running. It is quite an advanced function and allows you, for example, to disable AmigaDOS requesters such as "Insert Volume ... in any drive", or change the priority at which their task is run. There are 6 PRAGMA options: one returns information while the others allow you to set certain attributes.

	VALID OPTION KEYWORDS
DIRECTORY	Sets a new current directory. If the "value" item is omitted, then the current directory will be returned, and no change will be made.
PRIORITY	Changes task priority. Your ARexx script normally runs with a priority value of 0. Priorities can run from -128 to +127. The return value is the old priority. You cannot fetch the current priority value without specifying a new one. Although the possible range is quite large, it is not wise to set it to anything higher than 4 which is the priority at which the ARexx interpreter runs. Typical uses for this are changing the priority temporarily while you call an external host function which requires more processor time.
ID	Returns the task ID. This is actually the address of the Process structure for the current script. It is an 8-digit hexadecimal string. It is possible to read further information about the task when you have this information using the IMPORT() function. But never, ever write data to your task structure unless you are

	quite sure you know what you are doing, otherwise you could crash your computer.
STACK	Changes the stack size. This function changes the current stack size for your script's task and returns the old value. One of the most useful applications for this is when you are calling an external application or host command which requires a larger stack value in order to operate. You can set a new stack, store the old stack value in a variable and then set it back after calling the appropriate program/host command.
W	Enables or disables DOS requesters. Sometimes it is better to deal with errors like "No Disk Present" yourself, rather than having requesters pop up on the workbench screen. The W option allows AmigaDOS requesters to be switch on or off, depending on the "value" parameter, which can either be NULL to disable requesters, or WORKBENCH to enable them. These two parameters can be abbreviated to one character.
*	Changes the current default console handler. Without the "value" parameter this simply sets the default console handler back to the default one for the current process. This can be used to open a new window and use that for both input and output, like STDIN and STDOUT, the file handles for the normal console, which also share a window. Particularly useful for scripts launched without a shell window for input, like those started with RUN RX rather than just RX.

The option keywords can be abbreviated to one character, as with other ARexx options. For example:

```
SAY PRAGMA("I")
```

```
/* Returns task ID as hexadecimal address */

/* Sets the stack and task priority to new values
before running a program, and then resets them: */
old_pri = PRAGMA("PRI", 3)
old_stack = PRAGMA("STA", 15000)

/* Run the application which required the 15K stack and
priority of 3 */
ADDRESS COMMAND "application"

/* Now return stack and priority to original values */
CALL PRAGMA("PRI", old_pri)
CALL PRAGMA("STA", old_stack)

EXIT
```

RANDOM()

Syntax:

```
numeric = RANDOM([minimum] [,maximum] [,seed])
```

Returns a pseudo random number. The number returned is greater than or equal to the "minimum" parameter, and smaller or equal to the "maximum" parameter. If omitted, the default minimum and maximum are zero and 999. The difference between minimum and maximum must be no greater than 1,000. Pseudo random numbers use a mathematical calculation to get the next number. This calculation must be seeded with a suitably random value itself to avoid getting the same sequence of numbers every time you run your program. The easiest way to achieve this is to seed from the TIME("S") function (seconds elapsed):

```
CALL RANDOM(,,TIME("S"))
```

Larger ranges of numbers can be achieved by scaling the results from RANDOM(). For example:

```
/* Random Day of Week */

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

CALL RANDOM(,,TIME("S"))

picked = RANDOM(1,7)

SAY "Here is a random day of the week:" day.picked

EXIT
```

RANU()

Syntax:

```
numeric = RANU([seed])
```

Returns a pseudo random number between 0 and 1.

The precision of the number returned will depend on the current NUMERIC DIGITS setting (see keyword reference earlier in this section for more information). By multiplying this number out, random numbers of any range can be generated easily. The optional parameter can be used to seed the generator. See the RANDOM() function above for a suitable method of seeding using the TIME() function. The seed values for RANDOM and RANU are shared, so changing one will have the effect of re-seeding the other. For example:

```
SAY RANU()
/* Might show 0.234764734 depending on NUMERIC DIGITS
setting */
```

READCH()

Syntax:

```
result = READCH(file_handle, length)
```

Reads a number of characters specified using the "length" parameter from the specified file_handle. It is possible that the required length might not be read. This can be checked using the LENGTH() function on the returned string, reasons for this might include end of file reached, or an error occurred. End of file conditions can be checked with the EOF() function. For example:

```
string = READCH("MyFile", 30)
/* Read 30 characters from "MyFile", previously opened
with OPEN() */
```

READLN()

Syntax:

```
result = READLN(file_handle)
```

Read characters from the specified file handle until a new-line character is reached. The result is the entire line as read. The amount of bytes eventually read can be checked by using the LENGTH() function. The returned string does not include the new-line character itself. This function is typically used for processing text files off disk. For example, this program will count the number of lines and characters that are in your S:startup-sequence file:

```
/* READLN() example */
IF ~OPEN("ReadFile", "S:startup-sequence", "R") THEN DO
  SAY "I could not open file!"
  EXIT
  END

lines = 0
characters = 0

/* Loop through startup sequence a line at a time */
DO WHILE ~EOF("ReadFile")
  string = READLN("ReadFile")

  characters = characters + LENGTH(string)

  lines = lines + 1

  END

/* Now show file statistics */
SAY "File contained" lines "lines."
SAY "File contained" characters "characters."
SAY "Average line length = "characters/lines

CLOSE("ReadFile")

EXIT
```

REMLIB()

Syntax:

```
Boolean = REMLIB(name)
```

Removes a library from ARexx's list of libraries. This is the opposite to the ADDLIB() function. This function returns TRUE if the library was removed, or FALSE if not. The primary reason why REMLIB() might return FALSE is if the library specified did not exist. For example:

```
SAY REMLIB("rexxsupport.library")
/* Will show 1 if the library was removed, 0 if not */
```

REVERSE()

Syntax:

```
result = REVERSE(string)
```

Reverses a string. The result is the string parameter in reverse order. For example:

```
SAY REVERSE("?syas siht tahw tuo dnif ot rehtob enoyna
lliW")
```

RIGHT()

Syntax:

```
string = RIGHT(string, length [,pad])
```

Returns a sub string "length" characters long from the right hand side of the parameter "string". If length is larger than the string parameter, it is padded with the optional pad character. If no pad character is specified, a blank space is used as default. The opposite of this is the LEFT() function. For example:

```
SAY RIGHT("two words", 3)
/* Will produce the result "rds" */
```

SEEK()

Syntax:

```
result = SEEK(file_handle, offset ¬
[,"BEGIN"|"CURRENT"|"END"])
```

SEEK() is a function for moving the position pointer around in a file relative to either the current position, the beginning, or the end of the file. It expects a file handle, an offset from the anchor point, and the anchor point itself. The anchor point can be abbreviated to one character, and the default is CURRENT. The value returned is the old position. This function can therefore be used to find the current position in a file without changing anything using SEEK(file_handle, 0), which will move 0 bytes relative to the current position and return the old one. Example:

```
/* SEEK() example */
IF ~OPEN("WriteFile", "ram:test.txt", "W") THEN DO
  SAY "I could not open file!"
  EXIT
  END

WRITELN("WriteFile", "This is a string")

/* Move to the start of the file */
SEEK("WriteFile", 0, BEGINNING)

WRITECH("WriteFile", "Rope")

CLOSE("WriteFile")

EXIT
```

The above writes the line "This is a string" out to a new file called "ram:test.txt", then seeks back to the beginning, and writes over the "This" with "Rope". The results can be seen by TYPE'ing it from the shell:

```
TYPE ram:test.txt
```

SETCLIP()

Syntax:

```
Boolean = SETCLIP(clip_name [, value])
```

Alters an existing clip value, or creates a new one in the global clip list. The clip list is a list of special variables which can be accessed by all ARexx programs, and is maintained by the ARexx interpreter. It is a convenient way of sharing data values with other ARexx programs. Clip names are case sensitive. If a value is not specified, the named clip is deleted. This function returns TRUE if it succeeded, or FALSE if it did not.

```
SETCLIP("My_Clip", "This is the value")
/* Sets a value to the clip "My_Clip" */
```

See GETCLIP() for a further example of the SETCLIP instruction in action.

SHOW()

Syntax:

```
result = SHOW(option, [,pad])
Boolean = SHOW(option, name)
```

This function is used to examine information about the current resource list. It can be used in two basic ways, one, which returns a string to find out what items are available in a particular resource, and the other to check if a certain named value exists in a particular resource.

	AVAILABLE RESOURCE OPTIONS
CLIP	Examines the contents of the Global Clip List – see GETCLIP() and SETCLIP()
FILES	Examines the current files which have been opened using OPEN()
INTERNAL	Examines ports opened using OPENPORT (see section on the rexxsupport.library reference below)
LIBRARIES	Examines the names in the library list, those added using ADDLIB()
PORTS	Examines the public system port list.

All of the above options can be abbreviated to one character. In its first form SHOW returns a list of any items in the named resource and separates them with the optional pad character. If omitted, the pad character is assumed to be a space. In its second form SHOW can be used to confirm if a particular item is present. Examples of SHOW():

```
SAY SHOW("PORTS", "toby")
/* Shows 1 if the port "toby" was opened using OPENPORT
*/
SAY SHOW("F",,"0A"x)
/* Shows any files currently open, separating them with
the ASCII character 0A, which is Linefeed. This will
normally show STDIN, STDOUT and any other files you've
opened */
```

SIGN()

Syntax:

```
result = SIGN(number)
```

Returns -1 if the specified number is a negative one, or +1 if it is a positive one. Returns 0 if the number was 0. For example:

```
SAY SIGN(-100)
/* Produces the result -1 */
SAY SIGN(100)/* Produces the result +1 */
```

SOURCELINE()

Syntax:

```
result = SOURCELINE([line])
```

Returns the text with the named line in the currently-running ARexx program. If the optional line parameter is omitted, this returns the number of lines in the current program. This is very useful for error trapping using the SIGNAL command, because you are able to show: the line where the error occurred, the error text using ERRORTEXT(), and the actual line of source code causing the problem. For example:

```
/* SOURCELINE() example: */
SAY "Total lines in program:" SOURCELINE()

number = "false start"

DO WHILE DATATYPE(number, "NUMERIC") = 0

  SAY "Enter a line to display, from 1 to "SOURCELINE()
  PULL number
  END

IF number > SOURCELINE() | number = 0 THEN DO
  SAY "Line was illegal!"
  EXIT
  END

SAY "Line "number" was ""SOURCELINE(number)""" 

EXIT
```

SPACE()

Syntax:

```
string = SPACE(string, n [,pad])
```

This function reformats the string parameter so that there are exactly "n" spaces between each word. If the optional pad character is specified, it is used instead of spaces. Specifying 0 for n will result in all spaces being removed from the string. Spaces are not added before the first word and after the last, only between pairs of words. For example:

```
SAY SPACE("This is a test", 3, "-")
/* Results in: This---is---a---test */
```

STORAGE()

Syntax:

```
result = STORAGE([address] [,string] [,length] [,pad])
```

STORAGE() is a function to write information directly to memory. It behaves similarly to EXPORT but with two basic differences. First, unlike EXPORT() which returns the number of bytes written, STORAGE() returns the previous contents of that memory, allowing you to store this and set it back at a later point in your program. Secondly, STORAGE(), with no parameters, returns the total amount of free memory in your system.

WARNING **Like EXPORT(), this can be a very dangerous function. You can crash your computer or, even worse, corrupt data on your hard disk. While STORAGE is copying data to memory, ARexx will disable multi-tasking. So if you use STORAGE() to write large quantities of data, your system's performance might be adversely affected.**

An example of using Storage():

```
SAY STORAGE()
/* Shows total memory available in bytes */
previous_memory = STORAGE("00500000"x, "SomeText")
```

STRIP()

Syntax:

```
result = STRIP(string [,{"B" | "T" | "L"}] [,pad])
```

STRIP() is used to remove leading and/or trailing spaces from a string. If just a string is specified as a single parameter, the function removes both leading and trailing spaces from it. The second parameter can be used to specify whether leading, "L", trailing, ""T", or both, "B", are to be removed. The optional pad parameter allows you to specify the character to be remove. It will not remove characters from the middle of a string, just the beginning or end. Here is an example:

```
SAY STRIP("    hello    ", "B")
/* Returns "hello" */
```

```
SAY STRIP("    hello    ", "L")
/* Returns "hello    " */
```

SUBSTR()

Syntax:

```
result = SUBSTR(string, start [,length] [,pad])
```

Returns the specified sub-string of string. The "start" parameter specifies how many characters in to start, the "length" parameter is the number of characters to be included in the sub-string. If this is omitted, the default is the remainder of the string from "start" onwards. If the resultant sub-string is shorter than the required length, it is padded with blank spaces, or the optional pad character if specified. For example:

```
SAY SUBSTR("Hello",2,2)
/* Shows the result "ll", 2 characters in, 2 characters
in length */
```

SUBWORD()

Syntax:

```
result = SUBSTR(string, n [,length])
```

Returns the specified sub-string of string starting from the "n"th word, and "length" words long. If "length" is not specified, it is assumed to be the remaining words in the string from n onwards. For example:

```
SAY SUBWORD("This is a sentence", 3, 2)
/* Shows "a sentence" */
```

SYMBOL()

Syntax:

```
result = SYMBOL(name)
```

Checks to see if the supplied parameter is a valid ARexx symbol or not. Returns BAD if it is not, VAR if it is a valid variable with an assigned value, or LIT if it is an uninitialised variable. This example shows VAR LIT and then BAD on the screen:

```
/* SYMBOL() example */

symbol1 = "Hello"

SAY SYMBOL("symbol1")
SAY SYMBOL("symbol2")
SAY SYMBOL("+!&#E")

EXIT
```

TIME()

Syntax:

```
result = TIME([option])
```

With no parameter, this returns the current time in 24-hour mode, in the form HH:MM:SS. There are several option keywords which allow different information to be extracted, either related to the current time of day, or the elapsed timer which counts how long the current program has been running in seconds.

	OPTION KEYWORDS FOR DISPLAYING TIME
CIVIL	Normal time, in 12-hour format with an AM or PM appended to the end.
ELAPSED	Returns the total time in seconds that the current program has been running.
HOURS	Current time, hours since midnight.
MINUTES	Current time, minutes since midnight.
NORMAL	Current time in 24-hour mode. This is the default, if no option keyword is specified.
RESET	Resets the ELAPSED timer to zero.
SECONDS	Current time, seconds since midnight.

All of these options can be abbreviated to one character. Examples:

```
SAY TIME()
/* Current time, for example: 17:43:00 */
SAY TIME("CIVIL")
/* Current time, for example: 11:52PM */
SAY TIME("EL")
/* Elapsed time, for example: 32.22 */
```

TRACE()

Syntax:

```
result = TRACE([option])
```

Sets the current trace mode, and returns the old one, useful for restoring the old trace mode after a short change. If no new trace mode is specified, the old one is returned and no change is made to the current trace mode. The TRACE() function works even if interactive tracing is on and normal TRACE statements in the program are ignored. See Section B for further information on tracing and trace modes. See the trace statement for a list of trace modes and a discussion on interactive tracing. Example:

```
/* TRACE() example */

SAY "Trace mode is default at this point"

/* Switch interactive tracing on, ALL mode */
old_trace = TRACE(?A)

SAY "Trace mode is "TRACE()" currently."

CALL TRACE(old_trace)

EXIT
```

TRANSLATE()

Syntax:

```
result = TRANSLATE(string [,output] [,input] [,pad])
```

TRANSLATE() is an advanced function for replacing selected characters in a string with others. In its simplest form, with just the "string" parameter, it converts the string to upper case – behaving identically to the UPPER() function (although it's not as fast).

This is very easy to use. Two translation strings are provided, one containing the characters to replace in the "string" parameter, while the other contains the characters which will replace them. An example:

```
>SAY TRANSLATE("this is interesting", "a", "i")
thas as anterestang
```

All the "i"s in the string have been replaced with "a"s. TRANSLATE() scans the string character by character. When it comes across a character which is present in the "input" parameter, it replaces it with the corresponding character in the "output" parameter. Here is another example, which capitalises just the vowels in a string:

```
>SAY TRANSLATE("this is interesting", "AEIOU", "aeiou")
thIs Is IntErEstIng
```

The optional pad character, when specified, is used to pad out the shorter of the "input" or "output" parameters. The default is a space, for example:

```
>SAY TRANSLATE("this is interesting", "AEIOU",
"aeioust", "1")
1hI1 I1 In1ErE11Ing
```

TRIM()

Syntax:

```
result = TRIM(string)
```

Removes trailing spaces from the supplied string argument. For example:

```
SAY TRIM("   hello   ")
/* Produces the result "   hello" */
```

TRUNC()

Syntax:

```
numeric = TRUNC(number [,places])
```

Truncates the number supplied to the optional "places" number of decimal places. If not specified the default for "places" is 0, which removes any decimal fraction of the "number" parameter. If there are less decimal places than the value requested, it will be padded with "0"s as necessary. For example:

```
SAY TRUNC(10.123,8)
/* Shows "10.12300000" */
```

UPPER()

Syntax:

```
string = UPPER(string)
```

Converts the supplied string parameter to upper case characters. This function behaves like TRANSLATE() with no parameters, but it is more efficient. It understands foreign characters and correctly converts them to upper case also. For example:

```
SAY UPPER("This is an upper case sentence")
/* Shows "THIS IS AN UPPER CASE SENTENCE" */
```

VALUE()

Syntax:

```
result = VALUE(name)
```

Returns the value associated with the symbol specified. If the supplied parameter is not a currently initialised symbol, it will be created with a default value of the symbol name itself in upper case characters. For example:

```
/* VALUE() example */

a_symbol_1 = "Howdy"
a_symbol_2 = 10.123

SAY VALUE(a_symbol_1)
SAY VALUE(a_symbol_2)
SAY VALUE(a_symbol_3)

EXIT
```

This produces the following result when run:

```
8.System3.1:> rx value
HOWDY
10.123
A_SYMBOL_3
```

VERIFY()

Syntax:

```
numeric = VERIFY(string, list [,"MATCH"])
```

This function works in two ways, depending on whether the optional MATCH keyword is specified or not. If it is specified the function returns the offset, in characters from the start of the string, at which any of the characters in the list parameter occur. For example:

```
SAY VERIFY("Stegosaurus", "ur", "MATCH")
```

... will return 8, which is the first occurrence of any of the characters in "list" in the specified string. Without the MATCH parameter, the function works in the opposite way, returning the offset, in characters from the start of the string, at which any of the characters in the list parameter didn't occur. For example:

```
SAY VERIFY("Stegosaurus", "ur")
```

... will return 1, because S is not in the list of accepted characters.

WORD()

Syntax:

```
result = WORD(string, n)
```

Returns the "n"th word of the supplied string. In conjunction with the WORDS(), and other word-based functions listed below, this allows you to go through each word in a sentence successively, and could be useful for sentence processors such as those found in text adventure games. This function returns an empty string if word n does not exist. A word in ARexx is a group of non-white space characters separated by a white space, or bounded with the start or end of a string. For example:

```
/* Example of some of the WORDS based functions */
SAY "Input a sentence"
PARSE PULL sentence

word_index = 1

SAY "Sentence has "WORDS(sentence)" words in it."

DO WHILE WORDLENGTH(sentence, word_index) > 0
```

```
  SAY "Information on word "word_index", which is
""WORD(sentence, word_index)"""
  SAY "Word Length:" WORDLENGTH(sentence, word_index)
  SAY "Offset of Word from start of Sentence:"
WORDINDEX(sentence, word_index)
  word_index = word_index + 1
  END

SAY "Operation Complete!"

EXIT
```

The above takes a sentence and then provides information about each word in turn.

WORDINDEX()

Syntax:

```
numeric = WORDINDEX(string, n)
```

Returns the offset in characters from the beginning of the string of the "n"th word. Returns 0 if there are less than n words in the string (see WORD() for a full program example of the ARexx word functions in action).

```
SAY WORDINDEX("Here are some words", 4)
/* Produces the result "15", the offset in characters of
the fourth word from the start of the string */
```

WORDLENGTH()

Syntax:

```
numeric = WORDLENGTH(string, n)
```

Returns the length, in characters, of the "n"th word in the string. This function returns 0 if the word does not exist in the string (see WORD() for a full program example of the ARexx word functions in action). Example:

```
SAY WORDLENGTH("Here are some words", 4)
/* Produces the result "4", the length of the fourth
word in the string */
```

WORDS()

Syntax:

```
numeric = WORDS(string)
```

Returns the number of words in the supplied string (see WORD() for a full program example of the ARexx word functions in action). Example:

```
SAY WORDS("Here are some words")
/* Produces the result "4", the total words in the
string */
```

WRITECH()

Syntax:

```
numeric = WRITECH(file_handle, string)
```

Writes the supplied string out to the file specified. The result of the function is the number of characters that were successfully written. This function is very similar to the WRITELN() function, except that it does not automatically add a new line to the end of the string in the same way that WRITELN() does. For example:

```
SAY WRITECH("stdout", "Hello")
```

```
/* Writes the string to stdout, normally the shell
window. The result is "5", the number of characters
written */
```

WRITELN()

Syntax:

```
numeric = WRITELN(file_handle, string)
```

Writes the supplied string out to the file specified and appends a new line to the end of it. The result of the function is the number of characters that were successfully written. This function is almost identical to WRITECH() except it adds a new line to the end of the string outputted to the file, for example:

```
SAY WRITELN("stdout", "Hello")
/* Writes the string to stdout, normally the shell
window. The result is "6", the number of characters
written, including the new-line character which was
added */
```

X2C()

Syntax:

```
result = X2C(string)
```

Conversion from hexadecimal to character representation. This function is often used to convert hex ASCII codes into the appropriate character representations. Spaces are permitted in the string at byte boundaries only. For example, if you use the AmigaDOS command "TYPE" with "OPT H" to get a hex dump, you can use X2C to convert the hex characters shown into the characters themselves, here is the first line of my startup-sequence:

```
say x2c("3B202456 45523A20 53746172 7475702D")
```

Which produces the result:

```
; $VER: Startup-
```

Using this and the file access functions it is possible to write your own TYPE command complete with hex dump facilities.

X2D()

Syntax:

```
result = X2D(hex_number [, digits])
```

Hexadecimal to decimal conversion. The result is the decimal of the hex number string supplied. The optional "digits" parameter is a numeric stating how many of the characters in the hex number we should pay attention to during the conversion. The default is all. The "digits" are counted from the right hand side of the number, the least significant part. For example:

```
SAY X2D(ffff)
/* Produces the result 65535, the decimal of FFFF in
hex. */
SAY X2D(ffff, 4)
/* Shows -1, the actual number of a four digit FFFF
value */
SAY X2D(12345678, 2)
/* Shows 120, the decimal of the two digits furthest to
the right, 78 */
```

XRANGE()

Syntax:

```
result = XRANGE([start] [,end])
```

This function generates a sequence of characters from the start ASCII value to the end ASCII value. If the "start" value is omitted, it is assumed to be zero, and likewise if "end" is not supplied it is assumed to be 255 - the highest printable ASCII character. If no parameters are supplied, the entire range from 0 to 255 is generated (a string 256 characters in length). See the discussion on ASCII in Section B for further information. This function can be particularly useful for building translation tables for the TRANSLATE() function. For example:

```
SAY XRANGE("41"x, "48"x)
/* Result is: ABCDEFGH */
SAY LENGTH(XRANGE())
/* Result is: 256 */
SAY XRANGE("a", "z")
/* Result is all lower case characters from a to z
inclusive */

/* This next example uses TRANSLATE(). It converts
upper case to lower case: */
SAY TRANSLATE("THIS IS INTERESTING", XRANGE("a","z"),
XRANGE("A","Z"))
```

rexxsupport.library

The functions given in rexxsupport.library tend to be more advanced so be careful when using them, otherwise you could crash your computer. They are intended for the advanced programmer and allow you to allocate blocks of memory and deal with intertask message communications, for example. Before you can use these functions, however, you will need to add the library to ARexx's search list. There are a couple of ways of doing this. One is from within an ARexx program by using the ADDLIB function, like this:

```
/* Add rexxsupport.library to the ARexx Library List:
*/
IF ~ADDLIB("rexxsupport.library",0,-30,0) THEN DO
  SAY "Cannot add this library"
  EXIT
  END
SAY "Library added. Ready for use"
```

...or directly from your shell using the ARexx utility command RXLIB:

```
8.System3.1:> rxlib rexxsupport.library 0 -30 0
```

For a more detailed description of these two methods, see the appropriate reference section.

Each function is shown with its name, its syntax, a description and example, in the same format as the built-in functions listed above.

ALLOCMEM()

Syntax:

```
memory_address = ALLOCMEM(length [, attributes])
```

Allocates "length" bytes of memory and returns a pointer to where that memory is. If the allocation failed, the returned result is zero. The optional attributes allow you to specify which type of memory you would like. Unfortunately, it is not possible to specify simple keywords for memory types, you have to specify them in the format that the Amiga's operating system memory allocation routines require. A full list of memory types can be found in the *Rom Kernel Manuals: Libraries, Edition 3* (published by Addison Wesley). The default is "PUBLIC", which means that you are not fussed what sort of memory you get, but it must be public RAM. It is possible to ask specifically for your allocated memory to be cleared for you, or to be "Chip RAM (Graphics Memory)".

THE COMMONEST TYPES OF MEMORY SPECIFICATION

MEMF_CHIP	"00000002"X	Graphics memory (Chip RAM)
MEMF_FAST	"00000004"X	Fast memory
MEMF_PUBLIC	"00000001"X	Public memory (The default)
MEMF_CLEAR	"00010000"X	An addition flag which, when added to any of the above, ensures your allocated memory is cleared first.

In the above list, the exec name for the type is shown first, followed by the exact value (specified in hexadecimal) to put in the attributes argument, and then the action of that flag. If you wanted Cleared Chip RAM, you would therefore specify "00010002"X as the attributes argument.

WARNING **Unlike most other ARexx functions – including the built-in function GETSPACE() – ALLOCMEM is not resource tracked, which means that it is *your* responsibility to free any memory allocated with ALLOCMEM by using the FREEMEM() function. If you do not do this then your machine could run out of memory very quickly and crash. ALLOCMEM() should not be confused with the built-in function GETSPACE() which is less advanced, and does not allow you to specify memory types.**

Here is an example which allocates 1,000 bytes of chip RAM, cleared, shows us where it was (in hexadecimal), and then frees it before exiting:

```
/* Memory Allocation example */
memory = ALLOCMEM(1000, "00010002"X)

IF memory = 0 THEN DO
  SAY "Could not allocate memory"
  EXIT
  END

SAY "Memory was at $" C2X(memory)
FREEMEM(memory, 1000)

EXIT
```

BADDR()

Syntax:

```
result = BADDR(bptr)
```

Converts a BCPL (BPTR) pointer into a standard C pointer.

AmigaDOS was originally written in the BCPL language, which did not use standard memory pointers like C does. A simple conversion is required to turn BCPL pointers into C pointers. Unless you are working directly with the Amiga's dos.library you are unlikely to need this function. (In fact a BPTR points to a long-word aligned memory block divided by four. This means that, to convert a BPTR to a CPTR, you simply multiply the BPTR by four.)

The other BCPL nasty is the BSTR, which is a BCPL pointer to a string. Normal strings inside the Amiga are a collection of ASCII codes with a trailing zero to mark the end of the string. BCPL strings differ in that the first byte of the string contains the length, and the following bytes contain the string itself. There is no terminating zero. For example:

```
cptr = BADDR(bptr)
```

CLOSEPORT()

Syntax:

```
Boolean = CLOSEPORT(name)
```

Closes a message port previously opened using OPENPORT. If there were any messages pending on the port ARexx will automatically return them to their sender with an error code of 10. Returns TRUE if the port was successfully closed, or FALSE if there was an error. Here is an example:

```
CALL CLOSEPORT("DinosaurMessagePort")
SAY CLOSEPORT("FredPort")
/* Would show 1 if the port was closed, 0 if not */
```

DELAY()

Syntax:

```
DELAY(ticks)
```

Delay for the specified number of ticks. A tick is 1/50th of a second, so to delay for 1 second, you pass the parameter "50". For example:

```
/* Delay for 1.5 seconds: */
CALL DELAY(75)
```

DELETE()

Syntax:

```
Boolean = DELETE(filename)
```

Attempts to delete the specified file from disk. Returns a Boolean success variable, TRUE if the file was deleted, or FALSE if the deletion failed (which it might if the file was protected from deletion, or did not exist). For example:

```
>SAY DELETE("ram:test")
1
>SAY DELETE("ram:test")
0
```

The above example fails the second time around, because the file has already been deleted at that point.

FORBID()

Syntax:

```
result = FORBID()
```

Stops task switching. This has the action of disabling multi-tasking. Calls to FORBID() can be nested but, for every FORBID() called, there must be a matching call to PERMIT(), otherwise the machine is likely to freeze. The returned result is the current nest count of FORBIDs.

There are occasions when you will need to use FORBID in general programming, although under some complex circumstances it is used to ensure other tasks do not interfere with public information while you alter it.

Do not FORBID() for any length of time because other tasks running in your computer will also stop. In addition, you must always match every call to FORBID with one to PERMIT.

When you have called FORBID() your ARexx script will have all of the available processor time.

FREEMEM()

Syntax:

```
Boolean = FREEMEM(address, length)
```

Frees a block of memory which you had previously allocated using the ALLOCMEM() function. You must specify the address and length of the block you are freeing.

WARNING **You must ensure that you free the same amount of memory that you originally allocated, and at the same place. If you get this wrong your computer is likely to crash very rapidly indeed – or at best run out of memory. This function should not be confused with the built-in functions GETSPACE() and FREESPACE(). They are not interchangeable so, for instance, you cannot free memory allocated with GETSPACE() with FREEMEM().**

For an example of FREEMEM(), see ALLOCMEM() above.

GETARG()

Syntax:

```
result = GETARG(packet, [n])
```

Gets a command, function name or argument string from the specified message packet. This is one of a collection of functions which are especially designed to give the ARexx programmer access to lower level ARexx functionality. This, in conjunction with other packet handling routines, allows you to write your own basic host interface in ARexx. See Section F for an example. The specified packet is an address obtained from a previous call to GETPKT(). The optional parameter "n" specifies which slot the information should be extracted from. The command or function name is always in slot 0, and any additional parameters are in slots 1 to 15. The maximum number of arguments permissible is 15. In addition, "n must be equal to or less than the actual number of arguments for the packet. If not specified, "n" defaults to 0, so the function/command name is read from slot 0 of the packet. For example:

```
command = GETARG(my_packet)
arg_1 = GETARG(my_packet, 2)
```

GETPKT()

Syntax:

```
result = GETPKT(port_name)
```

Returns a four-byte address of a packet on the specified port_name, or "00000000"x if no packets were available. The port_name parameter is the name of a port which has been opened with a previous call to OPENPORT() within the current ARexx program. It is normal practice to use WAITPKT() to wait for a packet to actually arrive at a port before using GETPKT() to fetch it, this way your program is not wasting CPU time as it would if it was making constant calls to GETPKT() and checking the return value to see if a packet has arrived. See Section F for a proper example of the packet handling functions of ARexx in action. For example:

```
/* Wait for a packet and then get it on port
"toby's_port" */
CALL WAITPKT("toby's_port")
packet = GETPKT("toby's_port")

/* Cope with the possibility that the packet
disappeared */
IF packet = NULL() THEN DO
  SAY "Weird, I thought a packet was available, but it
isn't!"
  EXIT
  END

/* Show command/function name and first argument */
SAY "Command was:" GETARG(packet)
SAY "Argument 1 was: " GETARG(packet, 1)

EXIT
```

MAKEDIR()

Syntax:

```
Boolean = MAKEDIR(directory_name)
```

Attempts to create the specified directory. This function returns a Boolean success value, TRUE, if the directory was created, and FALSE if it was not (which could mean that the supplied path was invalid, or the directory already existed). For example:

```
>SAY MAKEDIR("ram:my_directory")
1
>SAY MAKEDIR("ram:my_directory/another_directory")
1
>SAY MAKEDIR("ram:my_directory/another_directory")
0
```

The last one fails because we've attempted to create a directory which now already exists.

NEXT()

Syntax:

```
result = NEXT(address [,offset])
```

Returns the four-byte value at the specified address. If the optional offset is specified this is added to the address before the value is read. This function is intended mainly for use with exec list structures, for following them forwards and backwards, although it has other uses. For example:

```
/* Go to the next node and say whether it was the last
in the list */
next_node = NEXT( current_node )
IF next_node = NULL() THEN SAY "Reached end of list"
```

NULL()

Syntax:

```
result = NULL()
```

Returns a null pointer as a four-byte string in the form "0000 0000"X. This is primarily for use with functions which return pointers to items as four-byte addresses, because these functions will often return a null pointer if they failed. For example:

```
>IF GETPKT("my_port") = NULL() THEN SAY "No message
available"
No message available
>SAY C2X(NULL())
00000000
```

OFFSET()

Syntax:

```
result = OFFSET(address, displacement)
```

Returns a new address having added the displacement parameter to the address parameter. The address is a four-byte string and the displacement is a numerical value. This function is particularly useful for calculating the address of certain fields within a system structure. Normally calls to C2D and D2C would be required to add four-byte string addresses together:

```
SAY D2C(C2D(address) + displacement)
```

... is functionally identical to the call:

```
SAY OFFSET(address, displacement)
```

... which is more straightforward, so you are less likely to make a mistake. For example, if you wanted to find the version of a library, whose library base is "07F45210", you can calculate the final address of the version (which is at offset "00000014"), thus:

```
>SAY C2X(OFFSET("07F45210"X,20))
07F45224
```

OPENPORT()

Syntax:

```
result = OPENPORT(port_name)
```

Creates a new public message port with the provided name. This function returns the four-byte address of the new port, or "00000000"x if the port could not be opened (the port opening would fail if a port with the given name already existed) or initialised. If you open lots of ports for one ARexx program you are likely to run out of signal bits. For more information on signal bits, consult the *Amiga Rom Kernel Manuals: Libraries, Edition 3;* published by Addison Wesley, which contains additional information on ports in general, though not in the context of ARexx.

Ports opened with OPENPORT() are automatically closed when the program exits and any pending messages are returned to their sender. However, it is good programming practice to free any resource when you have finished using it, to avoid using memory and other machine resources that you no longer need. See Section F for an example of OPENPORT in action, in a host interface written entirely in ARexx. An example of OPENPORT():

```
/* Open a public message port */
my_port = OPENPORT("toby's_port")
```

```
/* Check for the possibility that the port wasn't
created */
IF my_port = NULL() THEN DO
  SAY "I can't open the port"
  EXIT
  END

/* Put our port handing code here */
SAY "A-Ok"

/* Now close port and exit */
CALL CLOSEPORT("toby's_port")

EXIT
```

PERMIT()

Syntax:

```
result = PERMIT()
```

Re-enables multi-tasking which was previously disabled using a call to FORBID(). Calls to FORBID() and PERMIT() are nested and only succeeds when the final nested FORBID() has a corresponding call to PERMIT(). PERMIT returns the current nest count, or -1 if this call to PERMIT() finally re-started multi-tasking.

WARNING **Disabling multi-tasking degrades system performance. Only use it if strictly necessary. Every call to FORBID() has to have a matching PERMIT() call. See FORBID() for more information.**

For example:

```
/* Forbid() and Permit() example */
SAY FORBID()
SAY "The system is all ours"
SAY PERMIT()

EXIT
```

... produces the result ...

```
0
The system is all ours
-1
```

RENAME()

Syntax:

```
Boolean = RENAME(old_file, new_file)
```

Attempts to rename the file "old_file" to the name "new_file". Returns a Boolean success value of TRUE if the rename succeeded, or FALSE if it did not (which might happen if the new file was invalid, or the file did not exist). Although you can rename a file to another directory, you cannot move it to a different device using rename. For instance:

```
>SAY RENAME("libs:rexxsupport.library",
"libs:rexxsyslib.library")
0
```

This fails because the new filename specified already exists.

REPLY()

Syntax:

```
REPLY(packet, rc)
```

Replies to the sender of the supplied packet, setting the result field to the value given by the "rc" parameter (return code). The secondary result is cleared. The packet must be a four-byte address as supplied from GETPKT(), and rc must be an integer.

REPLY() is used in connection with other port and packet handing routines such as OPENPORT(), WAITPKT() and GETPKT(). These functions can be used to write your own host interface entirely in ARexx. See section F for a detailed example of this. For example:

```
/* Reply to the packet, with a return code of 10 */
CALL REPLY(my_packet, 10)
```

SHOWDIR()

Syntax:

```
result = SHOWDIR(directory, ["ALL" | "FILE" | "DIR"],
[pad])
```

Returns a list of files, directories, or both, in the directory specified with the "directory" parameter. Each entry in the list is separated with the optional pad character. The default for pad, if not specified, is a blank space. The second parameter can be used to filter the list so only files, or only directories are shown. If omitted, this parameter defaults to "ALL". These option keywords can be abbreviated to one character. For example:

```
/* Show files and sub-directories in the rexx: assign
separated with a - */
SAY SHOWDIR("rexx:", "a", "-")

/* Show a list of directories only in the SYS: assign,
with a new line between each entry */
```

```
SAY SHOWDIR("sys:", "D", "0A"x)
/* "0A"X is the ASCII code for LINEFEED */
```

SHOWLIST()

Syntax:

```
result = SHOWLIST(list_name, [name], [pad],
["ADDRESS"])
```

This is a useful function that allows you to check various system lists, such as a list of ports or assigns for example, and find specific information about that list – such as whether a certain entry exists. If the "name" parameter is not specified, then the result is all items in the appropriate list, separated with the optional pad character. The default pad character, if not specified, is a blank space.

If the "name" parameter is provided, then SHOWLIST() will return a Boolean result depending on whether "name" is present in the list specified. Name matching in this way is case sensitive. If the "ADDRESS" keyword is specified as well as a name, then the result of the function is a four-byte address of that item.

If you are finding the addresses of system resources in this way, be very careful what you do with the information – you could easily crash your computer and cause loss of data.

A table of valid values for the list_name parameter is given on the following page.

	VALID VALUES FOR LIST_NAME PARAMETER
ASSIGNS	Lists all assigns in the system, such as "DEVS:"
DEVICES	Lists devices present, such as "serial.device" and "timer.device"
HANDLERS	AmigaDOS handlers, such as CON, PIPE, SER, DF0.
INTRLIST	Nodes on the exec IntrList of interrupts.
LIBRARIES	Available libraries, such as "intuition.library" and "dos.library"
MEMLIST	Nodes on the exec MemList, such as "expansion memory" and "chip memory"
PORTS	Lists public message ports on the exec PortList structure.
RESOURCELIST	Lists resources in the system, such as "potgo.resource"
SEMAPHORELIST	Lists semaphores, – used to avoid conflicts between more than one task requiring access to the same resource.
TASKREADY	A list of tasks in a READY state.
VOLUMES	A list of volumes available (disk names, such as "System3.1")
WAITING	A lists of tasks in a WAIT state.

All of these can be abbreviated to one character.
SHOWLIST() also allows you to look at lists which it cannot deal with normally, by specifying a four-byte address of a

list header, rather than one of the above list names, for the first parameter. It is unlikely, however, that you'll ever want to do that in normal circumstance. Examples:

```
/* Show a list of volumes, separated with a new line */
SAY SHOWLIST("V",, "0a"X)

/* Say "Yes" if "lucyldb.library" is present */
IF SHOWLIST("L", "lucyldb.library") THEN SAY "Yes"
```

STATEF()

Syntax:

```
result = STATEF(filename)
```

Returns a string containing information about a file on disk. This information includes the length of the file, protection information and the date when it was last updated. The exact format of the returned string is as follows:

```
{DIR | FILE} length blocks protection days minutes
ticks comment
```

The first keyword tells you if the file specified was a directory name or an actual file. Then you get the length of the file in bytes, and then in blocks (normally 512 bytes), the protection details, and the last date the file was updated in days since 1st January 1978 followed by the number of minutes since midnight, and the number of ticks in the minutes. Finally the file's comment, if any. For example:

```
>SAY STATEF("libs:rexxsyslib.library")
FILE 33392 66 ----ARW-D 5552 898 1530
```

WAITPKT()

Syntax:

```
Boolean = WAITPKT(port_name)
```

Waits for a packet to arrive at the specified port_name. The result is a Boolean value, TRUE, if a valid packet is waiting at that port, or FALSE if not. Normally this will return TRUE because it waits until a packet is available before returning. However, in some error conditions it may return FALSE, so always check the result. Having waited for a packet, it can be fetched with GETPKT() and finally replied to using REPLY().

WAITPKT() is used in connection with other port and packet handing routines such as OPENPORT(), REPLY() and GETPKT(). These functions can be used to write your own host interface entirely in ARexx. See section F for a detailed example of this. For example:

```
>SAY WAITPKT("TestPort")
1
```

Other libraries

Other function libraries are also available. Many can be found in PD libraries and on bulletin boards. Each offers different features. For example, some of the popular libraries for generating windows with neatly laid out buttons come with rexx function libraries. Check any supplied documentation for proper instructions on usage.

Utility Programs Reference

ARexx comes with a number of special utilities that provide various control functions. These can be found in the Rexxc directory. The RX program which actually runs ARexx scripts can be found here. Apart from RX the others are not strictly necessary in order to use ARexx, but since they are only a few hundred bytes each, they are worth looking at. Several are very useful when debugging scripts. These utilities are run from the shell, or from ARexx scripts after issuing "ADDRESS COMMAND".

In the following reference section, optional parameters are specified in square brackets [], and after each command there is a summary of its purpose plus a detailed description.

HI

Stop all running scripts

This causes all active ARexx scripts to stop immediately. It does this by setting a global flag which causes the HALT interrupt to occur within each script. Scripts which intercept and process this interrupt cannot resist being terminated. Once all programs have been interrupted, the flag is reset. This can be used to stop run-away ARexx scripts.

RX name [arguments]

Run an ARexx Script

Runs the named ARexx script. If no path is specified, only the rexx: drawer and current directory are searched for the named script. Unless specified, RX adds ".rexx" to the end of the filename. Any arguments will be passed to the script for processing.

RXSET [name [[=] value]

Set a global variable value

Sets the value of an ARexx clip or, if no arguments are specified, lists any currently-defined clips in the list. An ARexx clip is like a global variable, a value that is available to all scripts which are run. This can be very useful because, for instance, you could set a value to a word directory which all ARexx scripts could then access to decide where to store information. There is a built-in function, called "GETCLIP" which allows ARexx scripts to read clips. From the shell, you could set a clip value like this:

```
8.System3.1:> rxset toby author_of_book
8.System3.1:> rxset
toby=author_of_book
8.System3.1:>
```

This sets the value of the clip "toby" to be equal to the string "author_of_book". We could then read this from within an ARexx script using getclip:

```
/* Clip reading Demo */
SAY "Value of Clip "toby" is:" GETCLIP("toby")
```

When run this produces the response:

```
8.System3.1:> rx toby
Value of Clip "toby" is: author_of_book
8.System3.1:>
```

RXC

Terminate ARexx

This causes the interpreter to stop accepting new scripts and terminate as soon as the last currently-running script finishes. It is the opposite of RexxMast. When RXC is run, the REXX public port is withdrawn. This prevents any new scripts from starting up. All memory occupied by the ARexx interpreter (about 45-50k) is released as soon as the last script finishes. Freeing this memory is the only reason for using RXC – and if you have a lot of RAM in your Amiga then this is not much use to you.

TCO

Opens global tracing console

This opens a special window on the workbench screen. All tracing information is directed into it. In addition, when using interactive tracing, any input required is typed into this window. Normally when using TRACE (as described in Section B), all program-tracing information is directed to the default console, which is often the shell window from which the script was run. When TCO has been run, all TRACE information from currently-running scripts is directed to this window. A script can override this by specifying a STDERR output channel, in which case tracing information will be directed there instead.

As TCO intercepts tracing information from every script running, it is best not to run too many at once because the trace window can become very busy and it will be harder to spot specific information.

TCC

Closes global tracing console

This closes the Global Tracing Console window after it has been opened using TCO. The window will only close when all scripts using it have indicated that they are finished. If any scripts are waiting for something to be input from the tracing window, then that must also be completed before it will close.

TS

Set global tracing flag

This sets the global tracing flag and switches on tracing operations in any currently-running scripts. It also means that any further ARexx scripts run will also have tracing on. This command is very useful as a debugging tool because you can regain control of programs that are going wrong, or stuck in an infinite loop, and find out what is happening.

TE

Clears global tracing flag

This is a global clear tracing command. It will switch tracing off in any ARexx scripts which currently have tracing on, and they will then run normally. It also acts as a reversal of the TS command.

WaitForPort [port-name]

Waits for a port to become active

This command waits for up to 10 seconds for the named Arexx port to become available. In other examples in this book we have used the Show() function to find out if a port is present, like this:

```
DO UNTIL Show(Ports, "rexx_ced")
  END
```

This would wait for the "rexx_ced" host address to become available. We could also use the WaitForPort command:

```
ADDRESS COMMAND
WaitForPort "rexx_ced"
```

This would have had the same effect but has the added advantage of only waiting for 10 seconds before failing with a return code value of 5. This is not normally enough to halt operation of an ARexx script, unless OPTIONS FAILAT has been set to 5 or lower. It means that you can check the return code value, as in this example:

```
/* WaitForPort Demo */

ADDRESS COMMAND
WaitForPort "Stegosaurus"

SAY "Return Code was:" RC
IF RC >= 5 THEN SAY "Which means that it didn't find the
port"
```

Assuming, of course that you don't have a program with an Arexx host address of "Stegosaurus" then this program will produce the result shown in Figure E-1, which also demonstrates TCO and TS in action to show tracing information in the global tracing console window.

RXLIB [name priority [offset] [version]]

Add a new library of functions to the library list.

This adds a new function library to those already available to ARexx. Function libraries are explained in more detail in the function reference in this section and in Section B. The RXLIB command is of particular use if you want to ensure that certain function libraries are always pre-loaded because you can add lines into your user-startup file like so:

```
RXLIB rexxsupport.library 0 -30 0
```

This would load the rexxsupport.library and make all of its functions immediately available to ARexx scripts. You can also load libraries from within scripts using the built-in Arexx function ADDLIB. For a library to be loaded, it must be present in your LIBS: drawer. Typing RXLIB by itself and pressing return lists all libraries currently in the library list, for example:

```
8.System3.1:> rxlib rexxsupport.library 0 -30 0
8.System3.1:> rxlib
rexxsupport.library (library)
REXX (host)
8.System3.1:>
```

In the above shell session, the rexxsupport.library was loaded and then RXLIB issued by itself to show currently loaded libraries.

Section F
Adding ARexx Support to Your Own Programs

This book would not be complete without a discussion on how you can add ARexx support to your own programs. As we have seen in Sections C and D, the ability to communicate with other applications and expand the functionality of ARexx is one of its biggest advantages over other languages. There will come a time when you write an application, be it in ARexx, or in another programming language such as "C" or Assembly, when you will want to be able to allow other programs to access some of its features and information. This is when you will need to consider adding proper ARexx support.

How you do this depends on the programming language you are using. The easiest way is to use ARexx itself. With the advanced functions provided in the rexxsupport.library, it is possible to set up a simple ARexx port, accept host commands from other applications, and act on them. The catch is that you can only implement command hosts, you cannot implement full function hosts (this ability may or may not be sorted in later versions of ARexx, we'll have to wait and see).

Another major disadvantage with implementing ARexx support in ARexx itself is the speed. ARexx is not one of the the fastest languages in the world, but its flexibility and ease of use normally compensate. Adding support this way is relatively easy and all the information you need is in this book. Other advantages of this method are that it is fast to write, easy to debug, and a quite neat way of prototyping a major application in another language.

If you wish to add ARexx support to an application which is not written in ARexx, then it becomes a little more complex – particularly if you are using Assembly. The Amiga's operating system isn't really designed to be programmed in Assembly, all of the functions provided for dealing with everything from opening windows to processing keyboard information were designed with "C" in mind. This is great if you are a "C" programmer, but can be a little daunting if not. Having said that, regardless of whether you program in "C", a basic "reading" knowledge of the language is very useful. By reading knowledge, I mean knowing enough "C" to be able to look at a listing and get a rough idea of what is going on. This makes Amiga developer documentation (the *Amiga Rom Kernal Manuals*, see Appendix 2) quite handy!

In this part of the book will show you how to add an ARexx port to an ARexx program, together with a working program example. In addition a brief introduction to adding ARexx support to other applications is discussed in "C", together with an example.

Adding ARexx Support in ARexx

The procedure for implementing a command host in ARexx itself is simple. First we have to open a port which will be our host address. Other applications can then "tune in" to us using the ARexx ADDRESS statement and by specifying our port name. Having opened a port, we simply wait for messages to arrive at it, deal with them, reply to them, and continue until we have finished; at which point the port is closed, and the program finishes.

The example program acts as a simple passive command host. It accepts two commands, QUIT and DINOSAUR, and is passive because it does not communicate with any other applications. It is a very simple prehistoric animal enquiry program. The QUIT command makes it exit and close its ARexx port. The DINOSAUR command, followed by a single parameter of the dinosaur name in question, will print information about that dinosaur. In this example, any additional code to show something a little more meaningful has been left for you to add (or at least imagine).

You could create a file of dinosaurs and data as a kind of "reference manual". This would allow you to create a fairly powerful program; in fact you could make it far more useful by interfacing it with an ARexx picture program to display a graphic to accompany the information. You could take this even further and play sampled, narrated text to talk you through each entry. We might be getting a little carried away here, but at least you can see how easy it would be to link other applications into even the simplest of programs to add more powerful features.

So, how do we use our command host? Well, if you type in and run the command host in one shell window, open another shell and then type in and run this small program, you will soon see! Our ARexx port name is "dinosaur":

```
/* Find info about a dinosaur */

IF ~SHOW(ports, "dinosaur") THEN
  DO
  SAY "The Dinosaur Enquiry program is not¬
  running."
  EXIT 10
  END

ADDRESS "dinosaur"

SAY "Enter a dinosaur name:"
PULL dino_name

DINOSAUR dino_name

EXIT
```

Clearly, it's easy to use and just like talking to any other command host such as the ProWrite word processor or Art Department Pro (see Section D). Now for how it all works…

Our command host does a few things to initialise itself. First it ensures the rexxsupport.library is available in the library list or, if not, it adds it. Then we check to see if we are already running – you can't create two of our command hosts at once. Most major ARexx-supporting programs, such as the CygnusED Professional text editor, will create unique port names by appending a different number to the end. CygnusED (CED), for example, calls its first port "rexx_ced" and then successive CEDs get the names "rexx_ced1", "rexx_ced2" and so on.

```
/*Check if a copy of this program is already running:
*/
IF SHOWLIST("PORTS", "dinosaur") THEN
  DO
  SAY "Port already exists, you can't run this¬
  program twice!"
  EXIT 10
  /* Return an error code */
  END
```

If it does already exist, the program shows an error and exits with an error code. If not, it moves on to create our ARexx port itself using the OPENPORT function. Once we have opened the port the actual procedure of receiving and processing messages is very easy and goes like this:

1 Wait for a message to arrive at our port
2 Confirm that it is a valid message
3 Extract the command and any parameters from the command message
4 Process the command
5 Reply to it
6 Repeat until done!

Note that when a message packet arrives at our port, it is not necessarily a valid one. This is common in a lot of cases on the Amiga, so we have to check that it is not null. We do this not by comparing the return value to zero, but to an empty address "00000000"X. The easiest way of doing this is by using the NULL() function:

```
/* If its not NULL, process it: */
IF packet ~= NULL() THEN
  DO
  /* Got one, it's great, process it... */
  END
```

In addition we have to reply to the packet when we have finished with it. When we reply, we also get to set the return code value. The program returns 10 if there is a serious error or 0 if everything is fine:

```
CALL REPLY(packet, error_level)
```

... we set the "error_level" variable to our desired value.

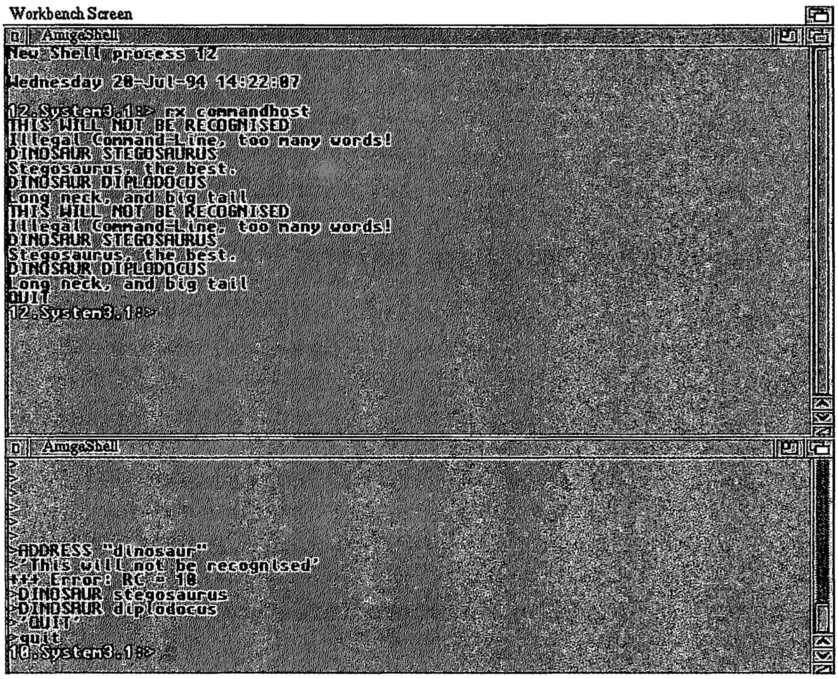
For more information on the functions used in this example look up the functions in the "rexxsupport.library", in Section E; Reference Section.

```
/* Simple Command Host (Passive) example in ARexx */

/*Load up the rexxsupport.library, we'll need it: */
IF ~SHOW("L", "rexxsupport.library") THEN
  DO
  IF ~ADDLIB("rexxsupport.library", 0, -30, 0) THEN
    DO
    SAY "Could not open ARexx support library!"
    EXIT 10
    /* Return an error code */
    END
  END
```

```
/*Check to see if a copy of this program is already
running: */
IF SHOWLIST("PORTS", "dinosaur") THEN
  DO
  SAY "Port already exists, you can't run this
  program twice!"
  EXIT 10
  /* Return an error code */
  END

/*Attempt to create our port: */
IF ~OPENPORT("dinosaur") THEN
  DO
  SAY "Can't create port"
  EXIT 10
  /* Return an error code */
  END

/*Now loop around waiting for messages: */
finished = 0

DO UNTIL finished

  /* First, wait for a packet: */
  CALL WAITPKT("dinosaur")

  /* Get the packet: */
  packet = GETPKT("dinosaur")

  /* If its not NULL, process it: */
  IF packet ~= NULL() THEN
    DO

    /* Reset error code level */
    error_level = 0

    /* Fine, now grab the full command sent to us: */
```

```
host_sent = UPPER(GETARG(packet))

IF WORDS(host_sent) > 2 THEN
  DO
  SAY "Illegal Command-Line, too many words!"
  error_level = 10
  END

ELSE

  DO

  command = WORD(host_sent, 1)
  /* Just the command */
  parameter = WORD(host_sent, 2)
  /* The parameter    */

  /* Process QUIT command */
  IF command = "QUIT" THEN

    finished = 1

  ELSE

    DO

    IF WORD(command, 1) ~= "DINOSAUR" THEN
      DO
      SAY "Unknown command:" WORD(command, 1)
      error_level = 10
      END

    ELSE

      DO
      /*Success, fetch information on this
      dinosaur! */
```

```
          CALL DinosaurInformation(parameter)
          END

        END

      END

    CALL REPLY(packet, error_level)

    END

  END

/*Close our port and exit: */
CALL CLOSEPORT("dinosaur")

EXIT

/*Now for the function to return information about a
specified dinosaur! */

DinosaurInformation: PROCEDURE

  PARSE ARG dino_name

  SELECT

    WHEN dino_name = "STEGOSAURUS" THEN SAY¬
    "Stegosaurus, the best."
    WHEN dino_name = "DIPLODOCUS" THEN SAY "Long¬
    neck, and big tail"
    OTHERWISE SAY "Unknown dinosaur, try another"
    END

  RETURN
```



```
New Shell process 12

Wednesday 28-Jul-94 14:22:07

12.System3.1> rx commandhost
THIS WILL NOT BE RECOGNISED
Illegal Command-Line, too many words!
DINOSAUR STEGOSAURUS
Stegosaurus, the best.
DINOSAUR DIPLODOCUS
Long neck, and big tail
THIS WILL NOT BE RECOGNISED
Illegal Command-Line, too many words!
DINOSAUR STEGOSAURUS
Stegosaurus, the best.
DINOSAUR DIPLODOCUS
Long neck, and big tail
QUIT
12.System3.1>
```

```
>
>
>
>
>
>
>
>
>
>ADDRESS "dinosaur"
>'This will not be recognised'
+++ Error: RC = 10
>DINOSAUR stegosaurus
>DINOSAUR diplodocus
>'QUIT'
>quit
10.System3.1>
```

Our prehistoric animal enquiry program in action.

ARexx Support: non-ARexx Applications

Adding an ARexx port to an application written in another language is more complex than in ARexx, but you benefit from additional features which would be impossible to get in ARexx without additional function libraries. We will discuss how to set up a passive ARexx port simply to receive messages and act on them.

However, most major applications supporting ARexx will want to create an active ARexx port and communicate with the ARexx resident process to control the clip-list, start

ARexx macros, modify an ARexx programs variable list, or communicate with other applications. Unfortunately, since this is such a large subject, it is beyond the scope of this book to provide all the information needed, including a break-down of the ARexx resident process and all of the examples that you'll need. There is a reference guide called *Amiga: Programmer's Guide to ARexx – Amiga Technical Reference Series*, but it is not, unfortunately, available from book-shops – only from Commodore. The book requires at least a reading knowledge of "C".

For the remainder of this section, we'll look into some of the basic information for adding ARexx support to "C" programs. If you are an Assembly programmer, the job will be harder, but is certainly possible. It's worth asking yourself though: "Do I really have to program in Assembly?" For most programs it's not usually necessary these days, and you just make work for yourself. Even some modern computer games are written in "C"!

The first thing we need to do when adding ARexx support to an application is create a unique ARexx port. This can be anything you want, but it makes sense for it to be simple, and correspond to the nature of the application. Since the Amiga is a multi-tasking operating system, you have to bear in mind that several of your applications could be run simultaneously. If this is a problem, then you can prevent your program running multiple times by checking for the presence of a particular message port. If it is not present, then yours will be the first copy run, so go ahead and create it. If it *is* present, show a message to say "You can't do this" and then exit the program.

We use a procedure similar to this to create our ARexx port. If the application can be legally run several times, each ARexx port has to be unique. The easiest (and standard) way of doing this is to add a number to the end of the port for each successive run, for example:

```
my_port.1
my_port.2
```

... and so on. Creating an ARexx port is easy, we simply use the CreatePort function:

```
#define PORT_NAME "fred"

void main(void)
{
struct MsgPort *my_port = NULL;

Forbid();
/* Prevent messing with ports while we're checking */

/*Check to ensure the port doesn't already exist: */
if (FindPort(PORT_NAME) == NULL)
  port = CreatePort(PORT_NAME, 0);
else
  {
  /* Port existed, we could add ".1", ".2", etc, on
  the end and try again here...*/
  }

Permit();

/* Great stuff, we have a port... */
}
```

A better way to achieve this might be to put the port creation routine into a separate function, so that it can be called in a loop to find a unique port, simply adding ".1", ".2" and so on to the end of the name each time, until the port is successfully created. Once we have created our port we are ready to use it to create an ARexx interface. The procedure for receiving messages on this port is simple:

1 Use the Wait() or WaitPort() function to wait for something to arrive
2 Use GetMsg() to get the message structure
3 If this is a reply to a message we have sent, deal with it. If not, it is a command, so look at the command and arguments in question
4 Reply to the message having stored any results that will be needed.

Sending messages is similar, although we won't be covering that in any detail here. However, the general procedure involves creating a special RexxMsg structure (used for both sending and receiving messages), setting it up as required, sending it direct to the appropriate message port (which we can find using FindPort) using PutMsg(), and then waiting for the reply to come back before re-allocating any resources used (such as the RexxMsg structure itself, for example).

You will notice that ARexx expects replies to all messages sent. If you are sending messages, you have to wait for the reply before freeing any resources used and, likewise, if you are receiving messages you have to remember to reply to them when you have finished.

WARNING **It is very important to remember this – failing to reply to messages can cause a dramatic loss in performance of your machine, and memory could vanish at a fair old rate because the sender of the message is not able to free memory. This could eventually crash your computer, so be careful.**

Now then, how do we set up our port to receive messages? Let's have a closer look at exactly what happens. When ARexx comes across a command which is sent to the current host, it is evaluated first and them dumped straight out. It is up to the receiving application to then decode that and deal with the arguments as it sees fit. Take this for example:

```
/* Send a command to the dinosaur command host program
example listed at the start of this section */
ADDRESS "dinosaur"
fred = 1
john = 2
lobster = "Thermidore"
ADDRESS "dinosaur"
DINOSAUR fred lobster john "hi"

EXIT
```

If we now examine what it is we receive at our port (you can easily modify the example ARexx command host program to do this) we'll see that we get this:

```
DINOSAUR 1 Thermidore 2 hi
```

You can see that the ARexx interpreter has fully evaluated the expression, substituting variables where required, and then sent the whole line to us for processing. This was easy to achieve in ARexx itself and it's just as easy to achieve in "C". Having evaluated the expression into a single string, the interpreter allocates a RexxMsg structure, then sets it up as a command message and gives us a pointer to the string. A RexxMsg structure looks like this:

```
struct RexxMsg {
  struct Message rm_Node; /* EXEC message structure */
  APTR      rm_TaskBlock; /* global structure ¬
  (private) */
  APTR      rm_LibBase; /* library base (private) */
  LONG      rm_Action; /* command (action) code */
  LONG      rm_Result1; /* primary result (return code)
*/
  LONG      rm_Result2; /* secondary result */
  STRPTR    rm_Args[16]; /* argument block ¬
  (ARG0-ARG15) */

  struct MsgPort *rm_PassPort; /* forwarding port */
  STRPTR    rm_CommAddr; /* host address (port name) */
  STRPTR    rm_FileExt; /* file extension       */
  LONG      rm_Stdin; /* input stream (filehandle) */
  LONG      rm_Stdout; /* output stream (filehandle) */
  LONG      rm_avail; /* future expansion */
  }; /* size: 128 bytes */
```

The RexxMsg structure and all of the other ARexx related header file information is found in the rexx/ drawer with the standard Amiga "includes" (supplied with any commercial compiler or assembler).

The interpreter sets the rm_Action field of the structure to RXCOMM (which means "Rexx Command Message"), and the rm_Args[0] to point to the string. Having done this, it sends the message to the command host for processing.

We would simply check the rm_Action field to see what type of message it was and if it is RXCOMM we could then parse the rm_Args[0] field to find the command and any arguments, perform the appropriate action, and then reply to it. Replying to the message is simple, we set up any return code information which needs to be sent back to the ARexx interpreter (such as a result, or error code), and then use the ReplyMsg() function. The interpreter would then pick up the reply and free the resources used to send you the command, and the ARexx program which caused the message to be sent could continue running.

We can find more information from the rm_Action field. An additional flag (RXFF_RESULT) is set if the interpreter expects a result to be sent back (for instance, if OPTIONS RESULTS was set then a result would be expected).

As well as the RXCOMM action for commands, we could also implement a function host (or indeed a combination of the two), by receiving and processing RXFUNC messages also (see below). Let's have a brief look at how we can receive and display what is sent to our ARexx port. We will assume that our port has already been opened, using the example code fragment shown above:

```
WaitPort(my_port);

while ((msg = (struct RexxMsg *) GetMsg(my_port))¬
!= NULL)
  {
  /* Got one, check if its a RXCOMM message */
  if ( (msg->rm_Action & RXCODEMASK) == RXCOMM)
    {
    /* Process our command now! */
    printf("Command is [%s]\n", msg->rm_Args[0]);

    /* Now deal with results */
    msg->rm_Result1 = RC_OK;

    if (msg->rm_Action & RXFF_RESULT)
      {
      /* Result string expected: */
      msg->rm_Result2  = (LONG)¬
      CreateArgstring("HELLO!", 6);
      }
    }

  /* All done, reply to it */
  ReplyMsg((struct Message *) msg);
  }
```

MAKE A NOTE!

We check to see if the RXFF_RESULT flag is set. If so, then we are expected to return something; if we have nothing to return, then we set the msg->rm_Result2 field to NULL. We must do this if there is no result to return, and RXFF_RESULT was set.

We can return results using the CreateArgstring() function which has two parameters, the string itself, and the length in characters. (The CreateArgstring function is present in the "rexxsyslib.library", which we must have open before we can use CreateArgstring.) This result is then copied directly to the RESULT variable in ARexx, which can be used by the caller program. Of course, for the caller program to receive RESULT at all, the:

```
OPTIONS RESULTS
```

... statement must be used. In all cases, however, regardless of whether RXFF_RESULT is set, you *must* set the rm_Result1 field to the return code value. This gets copied to the RC variable in ARexx and, depending on the value, the caller's ARexx program might be halted with an error. The rm_Result1 field is a single integer long value, which is the "severity of error" level. If it is zero, then there was no problem, while 5 is a warning, 10 is an error, and 20 is a severe error. In the above fragment, we always return 0. The include file rexx/errors.h defines equates for these common return codes:

COMMON ERROR RETURN CODES

RC_OK 0	
RC_WARN	5
RC_ERROR	10
RC_FATAL	20

... which we can use in our program as long as this file has been included. For example:

```
msg->rm_Result1 = RC_ERROR;
/* Return an error */
```

And that's about all there is to it for simple command hosts. Function hosts operate in a similar fashion, except we process the RXFUNC rm_Action code instead of RXCOMM:

```
  if ( (msg->rm_Action & RXCODEMASK) == RXFUNC)
    {
    /* Process the function: */
    }
```

The function name is contained in rm_Arg[0] – you identify if it's an acceptable value and, if so, you can then fetch the arguments themselves from rm_Arg[1], rm_Arg[2] etc, up to rm_Arg[15]. As you can see from this the maximum number of arguments that can be passed to a function host is 15, but this is unlikely to be a serious limitation – if you need more than 15 arguments to a function, you're definitely doing something wrong!

As well as identifying the function, rm_Action also contains the number of valid arguments. We can get it by masking the rm_Action field with RXARGMASK:

```
  number_of_args = msg->rm_Action & RXARGMASK;
```

As for command hosts, when we have finished we set the rm_Result1 field to be the RC return code (error severity level), and rm_Result2 to the result string if required (we check rm_Action & RXFF_RESULT to see if this is required).

This simple example in "C" creates an ARexx port, and responds to the single command "QUIT" which makes the program exit. It simply shows any other possible commands on the screen. It is easy to call from ARexx – this brief ARexx program, for example allows us to send commands to our host:

```
/* Talk to our host! */

IF ~SHOW(ports, "example") THEN
  DO
  SAY "The command host example is not running"
  EXIT 10
  END

ADDRESS "example"

SAY "What would you like to send to the host?"
PARSE PULL send

send

EXIT
```

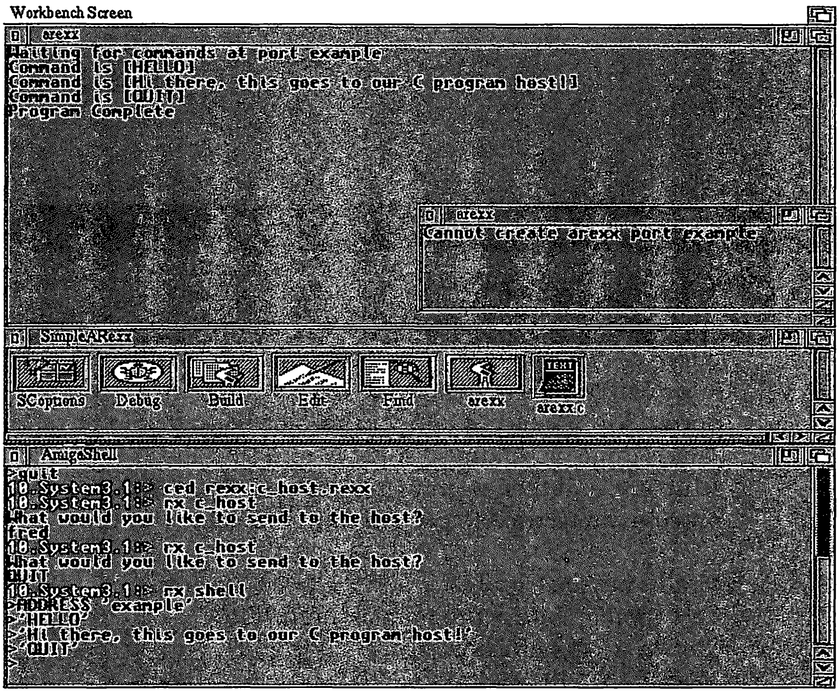
This listing was typed in and tested on SAS/C 6.51 but it should work on DICE and other "C" compilers with very little alteration. Consult the documentation that came with your "C" compiler for more information if you have difficulties compiling it.

```
/* Simple Passive ARexx Command Host in "C". */
/* Compiled and tested under SAS/C 6 */

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

#include <exec/exec.h>
#include <rexx/errors.h>
```

```
#include <rexx/rexxio.h>
#include <rexx/rxslib.h>
#include <rexx/storage.h>

#include <clib/exec_protos.h>
#include <clib/rexxsyslib_protos.h>
#include <clib/alib_protos.h>

/* Defines: */

#define AREXX_PORT_NAME "example"

/* Our global variables, the ARexx Port, and the
rexxsyslib.library library base: */

struct MsgPort        *arexx_port = NULL;
struct Library        *RexxSysBase;

/* Prototypes for functions in this module: */

void ProcessARexxMessages(void);

/* Main entry point: */
void main(void)
{
  struct RexxMsg     *msg;

  /*Open the rexxsyslib.library, required for
  CreateArgstring: */
  if (!(RexxSysBase = ¬
  OpenLibrary("rexxsyslib.library", 0L)))
    {
    printf("Can't open rexxsyslib.library.\n");
    return;
    }

  /* Open our port: */
```

```
Forbid();
if (FindPort(AREXX_PORT_NAME) == NULL)
  arexx_port = CreatePort(AREXX_PORT_NAME, 0);
else
  arexx_port = NULL;

Permit();

if (!(arexx_port))
  {
  printf("Cannot create arexx port %s\n",¬
  AREXX_PORT_NAME);
  return;
  }

printf("Waiting for commands at port %s\n",¬
AREXX_PORT_NAME);
ProcessARexxMessages();

/* All done, free resources and exit: */
CloseLibrary(RexxSysBase);

/* This next bit deletes our ARexx port now we're
done. Note that it stops multi-tasking, and
replies to any unanswered messages with an
RC_FATAL return code, then deletes the port, and
then restarts multi-tasking. This is to prevent
any messages not been dealt with, and any calling
applications from sending messages to a port
which has just closed, or is closing. */
Forbid();
while ((msg = (struct RexxMsg *)¬
GetMsg(arexx_port)) != NULL)
  {
  msg->rm_Result1 = RC_FATAL;
  msg->rm_Result2 = NULL;
  ReplyMsg((struct Message *) msg);
```

```
    }

  DeletePort(arexx_port);
  Permit();

  printf("Program Complete\n");

  return;
}

/*Process ARexx messages */

void ProcessARexxMessages(void)
{
  struct RexxMsg     *msg;
  BOOL   quit_flag   = FALSE;
  char   *command;

  while (!quit_flag)
    {
    WaitPort(arexx_port);

    while ( (msg = (struct RexxMsg *)¬
    GetMsg(arexx_port)) != NULL)
      {
      /* Got one, check if its a RXCOMM message */
      if ( (msg->rm_Action & RXCODEMASK) == RXCOMM)
        {
        /* Process our command now! */
        command = msg->rm_Args[0];
        printf("Command is [%s]\n", command);

        /* Deal with the quit command */
        if (!(strcmp(command, "QUIT")))      quit_flag¬
        = TRUE;

        /* Now deal with results */
```



```
Waiting for commands at port example
Command is [HELLO]
Command is [Hi there, this goes to our C program host!]
Command is [QUIT]
Program Complete
```

```
Cannot create arexx port example
```

```
>quit
10.System3.1:> ced rexx:c_host.rexx
10.System3.1:> rx c_host
What would you like to send to the host?
fred
10.System3.1:> rx c_host
What would you like to send to the host?
QUIT
10.System3.1:> rx shell
>ADDRESS 'example'
>'HELLO'
>'Hi there, this goes to our C program host!'
>'QUIT'
>
```

```
    msg->rm_Result1 = RC_OK;

    if (msg->rm_Action & RXFF_RESULT)
      {
      /* Result string expected: */
      msg->rm_Result2  = (LONG)¬
      CreateArgstring("HELLO!", 6);
      }
    }

/* All done, reply to it */
ReplyMsg((struct Message *) msg);
}
```

```
    }

  return;
}
```

The ARexx Resident Process

The ARexx resident process is very important to applications implementing ARexx support. It provides an interface between the application and a lot of ARexx-specific information. The resident process consists of two ARexx ports, "REXX" and "AREXX", to which requests can be sent. This is done by sending a suitably filled-out RexxMsg structure, and then waiting for the reply. The resident process can be used to perform the following actions;

- Modifying the Clip-List
- Call another ARexx program as a function
- Call an ARexx function
- Modify the Library List
- Change the global tracing status

Further information can be found in the "includes" and autodocs, and in the *Amiga Programmer's Guide to ARexx* (see details above). If you are interested in looking at some of this information a cheap and easy way is to get the current Amiga Developer's Kit (currently version 3.1 and available from Commodore for £23), and comes complete with the very latest disk-based reference for every library function, heaps of example code, utilities, debugging tools and so on – essential for every developer. If you're interested, you can send a cheque for £23 pounds, payable to Commodore Business Machines (UK) Ltd, to:

MAKE A NOTE!

Developer Support
Commodore Business Machines (UK) Ltd
Commodore House
The Switchback
Gardner Road
Maidenhead, Berks SL6 7XA

Include a covering letter explaining that the cheque is for the "3.1 Amiga Developer's Upgrade". If you are serious about Amiga development, you might also like to enquire about becoming a registered developer at the same time.

Function Libraries

As well as command and function hosts, programmers can implement function libraries. These act like the rexxsupport.library – a shared Amiga library which exists in the LIBS: drawer. It can be used just like rexxsupport.library, and added to the ARexx library list using the ADDLIB function. They are set up as standard Amiga shared libraries, and they must be fully re-entrant.

A Matter of Style

If you are implementing an ARexx port in your application, then it is worth thinking a little about consistency when it comes to "the naming of parts". Commodore have produced a book called the *User Interface Style Guide* (around £20, ISBN 0-201-57757-7).

This discusses a number of general style issues which are important for a consistent user interface between applications. Following this guide makes things easier for the computer user, who can use the same skills learned on one application when working with another. Among other things the *Style Guide* lists suggested ARexx command names for common commands, such as:

COMMON NAMES FOR COMMON COMMANDS

NEW	Create a new project (a new window in a text editor, new document in a word processor...)
CLEAR	Clear the current project (for instance, erase text, or clear picture).
OPEN	Open a project and load it into the current work area.
SAVE	Save a project
SAVEAS	Provide a requester so the user can pick where to save the current project
CLOSE	Close the current project and window
PRINT	Print the contents of the current project and window
QUIT	Quit the application

These are the items which you would commonly find on the "Project" menu for an application. Obviously not all of them will be relevant to your particular program, but when considering the naming of your ARexx commands, it is worth spending a few minutes browsing through the guide to see if you can use the naming conventions suggested there, because a consistent user interface is in everyone's best interest – both for users, and developers.

Appendix 1
Error Codes

If the interpreter detects an error, an error code is returned, together with a severity level – which is used to indicate what the error was and how serious it is. Errors which occur while your program is running can be trapped using the SIGNAL statement (See Section B, or Section E for information on SIGNAL).

Some errors do not happen within a program, and cannot be trapped using SIGNAL. For example, if you attempt to run an ARexx program which does not exist from the shell, this might happen:

```
8.System3.1:> rx no_such_program.rexx
Command returned 5/1: Program not found
```

When errors like this occur, the first number is the severity level, and the second is the error code itself. As well as a numeric representation of the problem, ARexx also shows us the textual description to make it easier to understand. Errors which are not trapped using SIGNAL and which happen within a running ARexx program will cause the program execution to stop, and an error to be displayed:

```
8.System3.1:> rx test.rexx
+++ Error 21 in line 4: Unexpected ELSE or OTHERWISE
Command returned 10/21: Unexpected ELSE or OTHERWISE
```

From this we know that an error of severity level 10, error code 21, occurred in line 4 of the program. Severity levels are normally 5, 10 and 20, where 5 is the least serious and 20 is very serious – such as "Insufficient Memory".

Two support functions are provided by ARexx for the processing of errors by the program itself. If the program traps errors using the SIGNAL ON SYNTAX, and then provides a function to deal with errors, information about the error can be gained by using these functions:

ERROR SUPPORT FUNCTIONS

result = ERRORTEXT(error code)	Returns error text corresponding to specified code
result = SOURCELINE([line])	Returns the actual line of source corresponding to line parameter

Further information about these functions can be found in Section E. In addition to the functions, the following pair of variables also contain valuable information:

OTHER USEFUL ERROR VARIABLES

RC	The error code which caused the error
SIGL	The program line at which the error occurred

The remainder of this section lists all the error codes, how severe they are, whether they can be trapped with SIGNAL, and give a brief summary of the error itself.

The Trap column contains a single letter which indicates whether the error is trappable using SIGNAL, and if so, which interrupt to use. S equals Syntax, H equals Halt, and the "-" sign means "Cannot be trapped using SIGNAL".

TABLE OF ERROR CODES

Code	Sevr	Trap	Message	Description
1	5	-	Program not found	An attempt was made to run a program which did not exist, or was not a valid ARexx program
2	10	H	Execution Halted	Program execution was halted. This could come from a user CTRL-C or the external halt interrupt (see utility command HI in Section E)
3	20	-	Insufficient memory	Not enough memory was available for the interpreter to function. It may or may not be trappable depending on just how much memory is left
4	10	-	Invalid character	Non-ASCII character found by the interpreter when processing a program line
5	10	-	Unmatched quote	Quote was found without the matching close quote
6	10	-	Unterminated comment	A comment was not terminated. A comment starts with /* and ends with */. This error occurs when the */ cannot be found

Code	Sevr	Trap	Message	Description
7	10	-	Clause too long	A program clause was too long for the ARexx interpreter to process
8	10	-	Invalid token	Interpreter could not identify a token
9	10	S	Symbol or string too long	Symbol or string was longer than the maximum allowed (65,535 bytes)
10	10	-	Invalid message packet	A packet sent to the ARexx resident process contained an invalid action code. The packet is returned without processing
11	10	-	Command string error	Command string could not be processed
12	10	S	Error return from function	Program called an external function which returned an error code (RC > 0)
13	10	S	Host environment not found	The program attempted to call a host address which did not currently exist
14	10	S	Requested library not found	The interpreter tried to open a library which was present in the library list, but could not be opened

Code	Sevr	Trap	Message	Description
15	10	S	Function not found	A function was called, but could not be found in any of the current loaded libraries, or as a program, or in the currently running program
16	10	S	Function did not return value	A called function did not return a result, but generated no errors
17	10	S	Wrong number of arguments	An incorrect number of arguments was passed to a function.
18	10	S	Invalid argument to function	An incorrect argument type was supplied to a function, or required information may have been omitted.
19	10	S	Invalid PROCEDURE	The PROCEDURE statement was misused.
20	10	S	Unexpected THEN or WHEN	The WHEN or THEN statement was misused.
21	10	S	Unexpected ELSE or OTHERWISE	The ELSE or OTHERWISE statement was misused.
22	10	S	Unexpected BREAK, LEAVE or ITERATE	The BREAK, LEAVE or ITERATE statement was misused.

Code	Sevr	Trap	Message	Description
23	10	S	Invalid statement in SELECT	A statement other than WHEN, THEN and OTHERWISE was found in a select range.
24	10	S	Missing or multiple THEN	Misuse, or omission of a THEN statement.
25	10	S	Missing OTHERWISE	None of the conditions within a SELECT range were satisfied, and no OTHERWISE statement was provided.
26	10	S	Missing or unexpected END	Either the program ended before a DO or SELECT statement block's END, or an END was found without a matching DO or SELECT.
27	10	S	Symbol mismatch	A symbol was specified after an END, ITERATE or LEAVE statement, but it could not be matched with a DO statement.
28	10	S	Invalid DO syntax	DO statement misused.
29	10	S	Incomplete IF or SELECT	An IF or SELECT statement block ended prematurely, and some of the required parts were omitted.

Code	Sevr	Trap	Message	Description
30	10	S	Label not found	The program attempted to transfer control to a label which did not exist, usually from a SIGNAL statement.
31	10	S	Symbol expected	Interpreter expected a symbol, but got something else. Statements such as DROP, END, LEAVE and ITERATE require a symbol.
32	10	S	Symbol or string expected	The interpreter expected a symbol or a string but found something invalid.
33	10	S	Invalid keyword	A keyword was not used correctly.
34	10	S	Required keyword missing	The interpreter was executing a statement which required a keyword, but it was not found.
35	10	S	Extraneous characters	Invalid unwanted characters were found at the end of an instruction clause.
36	10	S	Keyword conflict	Keywords were misused in a statement, or included more than once.
37	10	S	Invalid template	Template specified for ARG, PARSE or PULL was not valid.

Code	Sevr	Trap	Message	Description
38	10	S	Invalid TRACE request	TRACE or TRACE() was called with an unrecognised tracing option.
39	10	S	Uninitialised variable	Program used a variable which was not initialised. This error will only happen if the NOVALUE interrupt was enabled.
40	10	S	Invalid variable name	Program attempted to assign a value to a fixed symbol.
41	10	S	Invalid expression	An expression could not be evaluated because of an error.
42	10	S	Unbalanced parentheses	A bracket was used without a matching close bracket.
43	10	S	Nesting limit exceeded	An expression contained more than the maximum number of sub-expressions permitted (32)
44	10	S	Invalid expression result	The result of an expression was invalid in the context in which it was used.
45	10	S	Expression required	An expression was required by a statement but not supplied.

Code	Sevr	Trap	Message	Description
46	10	S	Boolean value not 0 or 1	An attempt was made to use the value of an expression as a Boolean, but it was not 0 or 1.
47	10	S	Arithmetic conversion error	While performing an arithmetic operation the interpreter was unable to convert an operand to a numeric value, the operand may contain non-numeric characters.
48	10	S	Invalid operand	An attempt was made to use an operand where it was not valid. This is generated if a program divides by zero or uses a fractional exponent in an exponential operation.

Appendix 2
Further Reading and References

ARexx programmers who want further information about the langauge – and the Amiga itself – should check out the publications listed below.

Amiga owners with A1200 computers will not have received two of the most useful manuals: the *AmigaDOS Reference* book, and the *ARexx Reference* book. The *AmigaDOS* book is essential reading and explains how to use the shell properly – a skill which is particularly useful for ARexx programmers. The *ARexx* book contains a lot of technical information about the ARexx language specification, and reference material. To get hold of these books you can buy the Amiga Workbench 3.1 upgrade kit. This comes with a complete set of manuals and includes an upgrade to Kickstart 3.1 (if you don't already have it). Or contact Commodore at the address given in Section F.

Further reading

Amiga Rom Kernel Manuals

Published by Addison Wesley, these are essential for any serious Amiga developer and are the official Commodore written documentation for the Amiga. Although their usefulness for ARexx programmers is limited, they do offer a vast wealth of information about what tasks your Amiga can perform – and how to carry them out using programming languages such as C.

Other useful publications

Libraries: Edition 3	ISBN 0-201-56774-1, around £30
Devices: Edition 3	ISBN 0-201-56775-X, around £25
Includes and Autodocs: Edition 3	ISBN 0-201-56773-3, around £25
The REXX Language: A Practical Approach to Programming, by Mike Cowlishaw, published by Prentice Hall.	ISBN 0-137-80651-5

Appendix 3
ASCII and ANSI Codes

This table shows the ASCII (American Standard Code for Information Interchange) character set, showing you how each character is represented in both hexadecimal and decimal. Section B has a more detailed discussion on ASCII codes but, to summarise, printable characters range from 32 to 126, various control codes are given 0 to 31 and 127. On the Amiga, additional codes are at 128 to 255 and these are special characters: some are letters of the Greek alphabet, for example. A few control codes have been omitted from this chart because they are irrelevant to the Amiga.

Amiga ASCII Codes

The control codes are generally only of use if you are working within console windows, such as the AmigaShell, in which case you can use them to tidy up displays or menus you are working on. Most of the control codes can be accessed directly from the keyboard for immediate use, by holding the control (CTRL) key down and pressing another key, then releasing both keys. This is called a control sequence. Control sequences for some of the more useful codes are shown in the key after the reference table.

TABLE OF ASCII CODES

Decimal	Hexadecimal	Character
000	00	NULL[1]
003	03	ETX[2]
007	07	BELL[3]
008	08	BS[4]
009	09	HT[5]
010	0a	LF[6]
011	0b	VT[7]
012	0c	FF[8]
013	0d	CR[9]

024	18	CAN[10]
027	1b	ESC[11]
032	20	SPACE
033	21	!
034	22	"
035	23	#
036	24	$
037	25	%
038	26	&
039	27	'
040	28	(
041	29	)
042	2a	*
043	2b	+
044	2c	'
045	2d	-
046	2e	.
047	2f	/
048	30	0
049	31	1
050	32	2
051	33	3
052	34	4
053	35	5
054	36	6
055	37	7
056	38	8
057	39	9
058	3a	:
059	3b	;
060	3c	<
061	3d	=
062	3e	>
063	3f	?
064	40	@
065	41	A
066	42	B

067	43	C
068	44	D
069	45	E
070	46	F
071	47	G
072	48	H
073	49	I
074	4a	J
075	4b	K
076	4c	L
077	4d	M
078	4e	N
079	4f	O
080	50	P
081	51	Q
082	52	R
083	53	S
084	54	T
085	55	U
086	56	V
087	57	W
088	58	X
089	59	Y
090	5a	Z
091	5b	[
092	5c	\
093	5d	]
094	5e	^
095	5f	_
096	60	`
097	61	a
098	62	b
099	63	c
100	64	d
101	65	e
102	66	f
103	67	g

104	68	h
105	69	i
106	6a	j
107	6b	k
108	6c	l
109	6d	m
110	6e	n
111	6f	o
112	70	p
113	71	q
114	72	r
115	73	s
116	74	t
117	75	u
118	76	v
119	77	w
120	78	x
121	79	y
122	7a	z
123	7b	{
124	7c	|
125	7d	}
126	7e	~
127	7f	DEL[12]
161	a1	¡
162	a2	¢
163	a3	£
164	a4	¤
165	a5	¥
166	a6	¦
167	a7	§
168	a8	¨
169	a9	©
170	aa	ª
171	ab	«
172	ac	¬
173	ad	-

174	ae	®
175	af	¬
176	b0	°
177	b1	±
178	b2	²
179	b3	³
180	b4	´
181	b5	µ
182	b6	¶
183	b7	·
184	b8	¸
185	b9	¹
186	ba	º
187	bb	»
188	bc	¼
189	bd	½
190	be	¾
191	bf	¿
192	c0	À
193	c1	Á
194	c2	Â
195	c3	Ã
196	c4	Ä
197	c5	Å
198	c6	Æ
199	c7	Ç
200	c8	É
201	c9	É
202	ca	Ê
203	cb	Ë
204	cc	Ì
205	cd	Í
206	ce	Î
207	cf	Ï
208	d0	Ð
209	d1	Ñ
210	d2	Ò

211	d3	Ó
212	d4	Ô
213	d5	Õ
214	d6	Ö
215	d7	x
216	d8	Ø
217	d9	Ù
218	da	Ú
219	db	Û
220	dc	Ü
221	dd	Ý
222	de	p
223	df	ß
224	e0	à
225	e1	á
226	e2	â
227	e3	ã
228	e4	ä
229	e5	å
230	e6	æ
231	e7	ç
232	e8	è
233	e9	é
234	ea	ê
235	eb	ë
236	ec	Ì
237	ed	í
238	ee	Î
239	ef	Ï
240	f0	
241	f1	ñ
242	f2	Ò
243	f3	Ó
244	f4	Ô
245	f5	Õ
246	f6	Ö
247	f7	÷

248	f8	ø
249	f9	ù
250	fa	ú
251	fb	û
252	fc	ü
253	fd	ý
254	fe	þ
255	ff	ÿ

Key to control codes:

Note number	**Name**	**Purpose**	**Keystroke(s)**
1	NULL character	often used to mark the end of a string	
2	End of Session	often used to terminate a running program	(CTRL-C)
3	BELL	depending on your preferences, this will flash your screen, make a sound, or both	(CTRL-G)
4	BS. Back Space	this moves the cursor one position to the left.	(CTRL-H or the BACKARROW delete key)
5	HT. Horizontal Tab	moves the cursor one tab position to the right	(TAB key, or CTRL-I)
6	LF. Line feed	this moves the cursor down one line, and on the Amiga also moves the cursor to the far left; effectively moving to the start of the next line on the display	(CTRL-J or RETURN or ENTER)
7	VT. Vertical Tab	moves the cursor up one line. Not much use on the Amiga.!	(CTRL-K)
8	FF. Form Feed	this clears the screen	CTRL-L)
9	CR. Carriage Return	moves the cursor to the start of the current line without moving it to the next or previous line	(CTRL-M)
10	CAN. Cancel	used on the Amiga mostly to abort the current text line. In a shell window, for example, it will clear the line you were typing and allow you to start again	(CTRL-X)
11	ESC. Escape	on the Amiga this starts an escape sequence, allowing you to put colours and other text effects in a console window (ie, a shell)	(ESCAPE or CTRL-[
12	DEL. Delete	deletes a character to the right of the cursor	DELETE)

Amiga ANSI Codes

In addition to the ASCII character set, the functionality of Amiga console windows have been extended by implementing part of the ANSI (American National Standards Institute) escape sequences. For a complete list of the sequences, consult the *Amiga ROM Kernel Reference Manual: Devices* (Edition 3, ISBN 0-201-56775-X).

The chapter on the "console.device" in this book explains the ANSI sequences in much greater detail. In addition, a list of codes is included in your *Workbench* manual, in the section on "Printers". For completeness, however, some of the more common and useful ones are listed here. To use one of these you have to send the ESC character first, then the ANSI sequence. In ARexx this is done like this:

```
/* ANSI Example */
/* Set ESC to be equal to the Escape Sequence Start ¬
Character */
ESC = "1B"XSAY ESC || "[3mThis is in italics!"
SAY ESC || "[33mThis is in colour 3 (and italics!)"
EXIT
```

ANSI Sequences are particularly useful for making ASCII displays look more presentable, because you can change both background and foreground colours, and alter the way in which the text is printed.

TABLE OF ANSI ESCAPE SEQUENCES

Escape Sequence	Result
[0m	Returns to normal character set (Disables italics, bold etc)
[1m	Bold on
[22m	Bold off
[3m	Italics on
[33m	Italics off
[30m to [39m	Set foreground colour
[40m to [49m	Set background colour
[4m	Underline on
[24m	Underline off

Discover the potential of your Amiga's best-kept secret!

The most powerful Amiga language ever?

ARexx could be your Amiga's single most valuable asset. It's a powerful programming language in its own right, it lets you created time-saving automated scripts for the increasing number of commercial programs that support it, and it is purpose-built for multi-tasking operations. Just like the Amiga, in fact.

'ARexx: Your Amiga's Built-in Turbocharger' contains everything you need to know to get the most out of ARexx. It includes help for beginners and authoritative reference material for experienced users.

Toby Simpson is a professional Amiga programmer who's developed a profound respect for ARexx and its abilities. As well as being head programmer at a top UK games development company, he is a regular contributor to *Amiga Shopper*, the UK's premier 'serious' Amiga magazine.

- Running ARexx
- Basic programming
- Function libraries
- Full ARexx reference
- Debugging
- ARexx 'ports'
- Controlling programs
- Automating tasks

£17.95

Future PUBLISHING

Your guarantee of value

ISBN 1-898275-09-2

9 781898 275091